Abstract

The Raspberry Pi is a credit card sized computer with versions costing between £20 and £35. It runs a full version of the Linux Operating System. Its files are held on an SD card typically holding between 2 and 32 Giga-bytes of data. When connected to a power supply, a USB keyboard and mouse, and attached to a TV via an HDMI cable, it behaves like a regular laptop running Linux. Programs for it can be written in various languages such as Python, C and Java, and systems such as Squeak and Scratch are fun to use and well worth looking at. This document is intended to help people with no computing experience to learn to write, compile and run BCPL programs on the Raspberry Pi in as little as one or two days, even if they are as young as 10 years old.

Although this document is primarily for the Raspberry Pi, all the programs it contains run equally well (or better) on any Linux, Windows or OSX system.

Keywords

BCPL, Programming, Raspberry Pi, Graphics.

Acknowledgements

I would particularly like to thank Philip Hazel for his helpful advice on how to improve this document.
# Contents

Preface v

1 Setting up the Raspberry Pi 1  
  1.1 Later versions of the Raspberry Pi ...................... 3

2 SD Card Initialisation 5  
  2.1 A More Recent SD Card Image .......................... 10

3 Introduction to Linux 13  
  3.1 The Filing System .................................... 15  
  3.2 The Desktop ........................................... 17  
  3.3 Midori .................................................. 18  
  3.4 Editing Files .......................................... 19  
  3.5 vi ....................................................... 19  
  3.6 emacs ................................................... 21

4 The BCPL Cintcode System 25  
  4.1 Installation of BCPL .................................. 26  
  4.2 Hello World .......................................... 31  
  4.3 Fibonacci ............................................. 34  
  4.4 Multiplication Table .................................. 42  
  4.5 A Mathematician’s Approach .......................... 43  
  4.6 Numbers ............................................... 47  
  4.7 Applications of XOR and MOD ........................ 50  
    4.7.1 RSA Mathematical Details .......................... 52  
  4.8 Vectors ............................................... 53  
  4.9 Primes .................................................. 57  
  4.10 MANIFEST, GLOBAL and STATIC declarations .......... 58  
  4.11 Functions ............................................. 60  
  4.12 Solving the recurrence relation for C ............... 63  
  4.13 Greatest Common Divisor ............................. 64  
  4.14 Powers ................................................ 65  
  4.15 Compilation ........................................... 66
CONTENTS

4.16 The Collatz Conjecture ........................................ 72
4.17 The Pig Dice Game ............................................. 78
  4.17.1 The Optimum Strategy ...................................... 86
4.18 The Enigma Machine ............................................. 89
  4.18.1 enigma-m3 functions ....................................... 94
4.19 Breaking the Enigma Code ..................................... 121
4.20 The Advanced Encryption Standard ......................... 131
  4.20.1 Final Observation .......................................... 145
4.21 GF(2^8) Arithmetic ............................................. 148
4.22 Polynomials with GF(2^8) Coefficients ...................... 150
4.23 Reed-Solomon Error Correction .............................. 154
4.24 The Queens Problem ............................................ 171
4.25 Sudoku ......................................................... 174
4.26 The Sliding Blocks Puzzle .................................... 182
4.27 The Rubik Cube ................................................ 202
4.28 Simple series .................................................. 240
4.29 e to 2000 decimal places ..................................... 243
4.30 The χ^2 test ..................................................... 247
4.31 e^x ............................................................. 248
4.32 The extraordinary number e^{π√163} ......................... 249
4.33 Digits of π ....................................................... 255
4.34 More commands ................................................ 261
4.35 The VSPL Compiler ............................................. 262
4.36 Summary of BCPL ................................................ 263
  4.36.1 Comments and GET .......................................... 263
  4.36.2 Sections .................................................... 264
  4.36.3 Declarations ................................................ 264
  4.36.4 Definitions ................................................ 264
  4.36.5 Expressions ................................................ 264
  4.36.6 Commands .................................................. 266
  4.36.7 Constant expressions ..................................... 267

5 Interactive Graphics in BCPL using SDL ................................. 245
  5.1 Introduction ................................................... 245
  5.2 The dragon curve .............................................. 250
  5.3 Collatz Revisited .............................................. 253
  5.4 sdlinfo.b ...................................................... 256
  5.5 Graphs ......................................................... 259
  5.6 Gradients ....................................................... 262
  5.7 Events ........................................................ 265
  5.8 e^{ix} and rotation ........................................... 269
  5.9 Polar Coordinates ............................................. 281
  5.10 The Mandelbrot Set .......................................... 281
### 5.11 Ball and Bucket Game ........................................... 293
### 5.12 Robots ....................................................... 323
### 5.13 Moon Lander ................................................ 355
### 5.14 A Library for High Precision Arithmetic .................. 370
  * 5.14.1 A Simple Example ....................................... 397
### 5.15 The Airy Disk .............................................. 403
### 5.16 A Catadioptric Telescope .................................... 415
### 5.17 A 3D Demo .................................................. 461
### 5.18 drawtigermoth.b ............................................ 477
### 5.19 Tigermoth Flight Simulator .................................. 496

### 6 Interactive Graphics in BCPL using OpenGL 526
  * 6.1 Introduction to OpenGL ....................................... 527
  * 6.2 Geometric Transformations .................................... 528
  * 6.3 Viewing the Scene ............................................ 531
  * 6.4 A first OpenGL example ....................................... 535

### A sdl.h 615

### B sdl.b 622

### C Package Installation Details 642
  * C.0.1 Installing BCPL under Linux, the Raspberry Pi and Mac OSX ........................................ 642
  * C.0.2 Installing Emacs under Linux, the Raspberry Pi and Mac OSX ........................................ 643
  * C.0.3 Installing SDL under Linux and the Raspberry Pi ........ 644
  * C.0.4 Installing SDL2 under Linux and the Raspberry Pi ... 645
Preface

When a new programming language is designed it is invariably strongly influenced by languages that preceded it. One thread of related languages is: Algol -> CPL -> BCPL -> B -> C -> C++ -> Java, indicating that BCPL is just a small link in the chain from the development of Algol in the late 1950s to Java in the 1980s. BCPL is particularly easy to learn and is thus a good choice as a first programming language. It is freely available via my home page (www.cl.cam.ac.uk/~mr10) and the only file to download is called bcpl.tgz. This is easy to decompress and install on the Raspberry Pi and so, in very little time, you can have a usable BCPL system running on your machine.

The main topics covered by this document are:

• How to connect the Raspberry Pi to a television, keyboard, mouse, and power supply.

• How to initialise its SD card with a version of the Linux Operating System.

• How to login to the Raspberry Pi followed by a brief description of a few Linux Shell commands.

• How to obtain and install BCPL on the Raspberry Pi.

• Then follows a series of examples showing how to write, compile and run BCPL programs.

• Near the end there are some example programs involving interactive graphics using the BCPL interface to the SDL graphics library.

• Finally, there is a section outlining some of the debugging aid provided by the BCPL system.

Professional computer scientists require a reasonable grounding in mathematics and so some mathematics has been included in this document, but even though some is of university level, the approach taken requires very little mathematical background, and should be understandable by most young people. But if this is not to your taste, skip any sections remotely connected with mathematics.
Chapter 1

Setting up the Raspberry Pi

The Raspberry Pi is a credit card sized computer that runs the freely available Linux Operating System. I recommend using the Model B version, as shown in Figure 1.1, since it is more powerful and not much more expensive than Model A. It is powered by a typical mobile phone charger using a micro USB connector, but be careful to choose a charger that can supply at least 700 milli-amps.

![Raspberry Pi with connectors](image)

**Figure 1.1:** Raspberry Pi with connectors

The Raspberry Pi can be connected to a TV using an HDMI cable although an analogue connection is also available. With some early versions of Linux for the Raspberry Pi, the HDMI connection failed to work properly. Luckily these early problems seem to have gone away with later versions of the software.
A USB keyboard and mouse is required and a combined wireless keyboard and touch pad is particularly convenient since it allows you to sit in the comfort of an armchair with the keyboard on your knee and the Raspberry Pi neatly hidden behind the TV. The lack of unsightly trailing cables is a clear bonus and leaving the second USB socket free is an added advantage. My favourite keyboard is made by Sandstrøm (available from PC World for about £30). A radio keyboard with a separate mouse might be even better. The picture of the Raspberry Pi shows the tiny USB radio dongle for the keyboard to the left, the HDMI cable above and the micro USB connector for the charger to the right.

Figure 1.2 shows the Raspberry Pi fully connected only requiring the HDMI lead to be connected to a TV and the charger plugged into a socket. Notice that at the right side of the machine, you can see part of the blue SD memory card which has to be preloaded with a suitable version of Linux. If you have access to the internet, you can plug a suitable ethernet cable into the Raspberry Pi. This is not absolutely necessary but does have many advantages, particularly for the automatic setting of the date and time, web browsing and downloading software.

The SD card should have a size between 2 and 32 GBytes, although I recommend initially using a card of between 4 and 8 GBytes. Unfortunately some SD cards seem not to work. There are several good web pages supplied by the Raspberry Pi community that describe how to load the Linux image into the SD card. The version of Linux I currently use allows me to login as user **pi** with password **raspberry** leaving me connected to a **bash** shell waiting for Linux commands.
1.1. Later versions of the Raspberry Pi

In early February 2015, a new version of the Raspberry Pi became available. It has 1Gb of RAM, 4 USB sockets and is about six times faster the the earlier version. It uses a micro SD card for its disk memory and the machine still costs about same as the previous version. A major advantage is that its operating system provides full support for floating point machine instructions which is invaluable since BCPL now supports floating point which is used extensively in programs involving OpenGL graphics. I therefore strongly advise you to upgrade to this
version and buy a good quality fast (class 4) micro SD card, typically of size 8Gb. This machine is shown in Figure 1.4.

Figure 1.4: The Raspberry Pi Model B-2
Chapter 2

SD Card Initialisation

The SD memory card must be initialised with a suitable version of Linux and this chapter outlines how this can be done. Since it is potentially a dangerous operation I strongly recommend you look at the various tutorials and videos on the Web supplied by members of the Raspberry Pi community. A good place to start is to do a google search on: Raspberry Pi SD card setup, and also look at the web page: www.raspberrypi.org/downloads.

You will need access to a desktop or laptop computer running some version of Windows, OSX or Linux, and a connection to the internet. I used a laptop computer (called solestreet) running Linux to perform the download and all the operations needed to initialise the SD card. I strongly recommend using Linux and in particular the Wubi version of Ubuntu Linux for many reasons. Firstly, it is easy to install on Windows machines without needing the tricky and potentially dangerous job of repartitioning your hard disc. It allocates one large file on Windows to hold the entire Linux filing system. I would recommend allocating 20 Gbytes if you can spare that much, but less will work. You can uninstall Wubi Linux in exactly the same way you uninstall other Windows programs, and again there is no need to repartition the hard disc. Secondly, it has a lot in common with the Linux system you will be using on the Raspberry Pi, including, for instance, the apt-get mechanism for downloading and installing Linux packages. Finally, it already has most of the commands installed such as ls, cd, df, dd, sudo, parted, e2fsck, fdisk and resize2fs that you will need when setting up the SD card. Even if some are absent, they are easily obtained by commands such as: sudo apt-get install parted. A further advantage is that all the fragments of terminal sessions in this chapter were run using the Wubi version of Linux on my laptop. I believe Wubi Linux already has the Workspace Switcher program which allows to the switch easily between four separate screens. Two other programs I strongly recommend installing are Terminator which is a brilliant terminal program and emacs which is my favourite screen editor for editing text files. If suitably configured, emacs will give different colours to reserved words, strings, comments and other lexical features of BCPL programs making them easier to
read. I would also recommend installing emacs on the Raspberry Pi for the same reason. Details of how to use emacs will be given later.

Using a web browser you should be able to download a suitable Linux image. The recommended Debian "squeeze" is ideal and the one I used was called: `2012-07-15-wheezy-raspbian.zip`, but its name keeps changing as updates are made. The zip file can be expanded to produce the image file called: `2012-07-15-wheezy-raspbian.img`.

I connected a 500 Gbyte external USB disc drive to the laptop and so had plenty of disc space for both the zip and image files. The USB drive turned out to have name `/media/TOSHIBA/EXT` and so I changed to this directory, created a subdirectory directory called `raspi` and made it the current directory. The commands used were:

```
solestreet:$ cd /media/TOSHIBA/EXT
solestreet:$ mkdir raspi
solestreet:$ cd raspi
solestreet:
```

I used a web browser to download `debian6-19-04-2012.zip` into this directory and inspected the result.

```
solestreet:$ ls -lrt
total 453696
-rw------- 1 mr10 mr10 461001289 Jul 23 14:07 2012-07-15-wheezy-raspbian.zip
solestreet:
```

I then checked the checksum using `sha1sum` and expanded the file using `unzip`.

```
solestreet:$ sha1sum 2012-07-15-wheezy-raspbian.zip
3947412bab0bf63f9f022f10b22ea6a308bb630c 2012-07-15-wheezy-raspbian.zip
solestreet:
solestreet:$ unzip 2012-07-15-wheezy-raspbian.zip
Archive: 2012-07-15-wheezy-raspbian.zip
    inflating: 2012-07-15-wheezy-raspbian.img
solestreet:$ ls -lrt
total 2344600
-rw------- 1 mr10 mr10 1939865600 Jul 15 20:45 2012-07-15-wheezy-raspbian.img
-rw------- 1 mr10 mr10 461001289 Jul 23 14:07 2012-07-15-wheezy-raspbian.zip
solestreet:$
```
This takes some time so be patient, but when it completes, it will have created
the file 2012-07-15-wheezy-raspbian.img. The size of this image is very nearly
2 Gbytes which just fits on a 2 Gbyte SD card, but later images are likely to be
bigger so it would be wise to buy SD cards of at least 4 Gbytes.

Now come the tricky and potentially dangerous part. This file represents the
complete image of what must be written to the SD card destroying everything
that was previously on it. If you accidently write it to the wrong place, you may
well make your laptop or desktop unusable, so great care is required.

My laptop has a slot for an SD card and so can be conveniently used to
initialise the card. First, I executed the df -h command producing the following
output.

solestreet:$ df -h
Filesystem  Size  Used  Avail  Use% Mounted on
/dev/sda6   32G   16G   14G  53% /
udev       743M   4.0K  743M  1% /dev
tmpfs      300M  840K  300M  1% /run
none       5.0M   8.0K  5.0M  1% /run/lock
none       750M  344K  750M  1% /run/shm
/dev/sda7   100M   75M   25M  80% /boot
/dev/sda2   4.0G  2.5G  1.6G  63% /dose
/dev/sdb1   466G  25G  441G  6% /media/TOSHIBA EXT

solestreet:$

I then inserted a suitable SD card (a Verbatim 4Gbyte card) and ran the command
again giving:

solestreet:$ df -h
Filesystem  Size  Used  Avail  Use% Mounted on
/dev/sda6   32G   16G   14G  53% /
udev       743M   4.0K  743M  1% /dev
tmpfs      300M  852K  300M  1% /run
none       5.0M   8.0K  5.0M  1% /run/lock
none       750M  344K  750M  1% /run/shm
/dev/sda7   100M   75M   25M  80% /boot
/dev/sda2   4.0G  2.5G  1.6G  63% /dose
/dev/sdb1   466G  25G  441G  6% /media/TOSHIBA EXT
/dev/mmcblk0p1  58M  34M  24M  59% /media/18DA-FFB9
/dev/mmcblk0p2  3.7G  1.5G  2.0G  43% /media/a6b2691a-99d8-47...

solestreet:$

This shows that the SD card was called /dev/mmcblk0 and already had two par-
titions on it, one of size 58 Mbytes and the other of size 3.7 Gbytes. I unmounted
both these two partitions using the `umount` command twice and used the `sudo dd` command to copy the Raspian Linux image to the SD card. This is the command that required special care since mistakes can make your machine unusable. The commands used were as follows:

```
solestreet:$ umount /dev/mmcblk0p1
solestreet:$ umount /dev/mmcblk0p2
solestreet:$
solestreet:$ sudo dd bs=1M if=2012-07-15-wheezy-raspbian.img of=/dev/mmcblk0
[sudo] password for mr10:
1850+0 records in
1850+0 records out
1939865600 bytes (1.9 GB) copied, 468.454 s, 4.1 MB/s
solestreet:$
```

As can be seen, this took 468 seconds or nearly 8 minutes so patience is again required. I then extracted the SD card after issuing the `sync` command to ensure that all disc transfers have completed.

```
solestreet:$ sync
solestreet:$
I re-inserted the SD card to see what it contained.

```
solestreet:$ df -h
Filesystem     Size  Used  Avail Use% Mounted on
/dev/sda6      32G   16G   14G  53%  /
udev           743M   4.0K  743M   1%  /dev
tmpfs          300M  852K  300M   1%  /run
none           5.0M   8.0K   5.0M   1%  /run/lock
none           750M  344K  750M   1%  /run/shm
/dev/sda7      100M   75M   20M  80%  /boot
/dev/sda2      4.0G   2.5G   1.6G  63%  /dose
/dev/sdb1      466G   25G  441G   6%  /media/TOSHIBA EXT
/dev/mmcblk0p1  56M   34M   23M  61%  /media/1AF7-904A
/dev/mmcblk0p2  1.8G  1.3G  439M  75%  /media/8fe3c9ad-c8f5-4b39-aec2-f6e8dba743e0
solestreet:$
```

```
solestreet:$
solestreet:$ cd /media/8fe3c9ad-c8f5-4b39-aec2-f6e8dba743e0
solestreet:$ ls
bin  dev  home  lost+found  mnt  proc  run  selinux  sys  usr
boot  etc  lib  media  opt  root  sbin  srv  tmp  var
solestreet:$
```
Note that the horrible looking `cd` command is easy to type because you only have to input `cd /media/8` and then press the Tab key for `bash` to fill in the rest of the file name automatically.

The directory `home` contains all the home directories of users permitted to use the machine, however at this stage no users are set up. The first time this image is run on the Raspberry Pi, it creates a user called `pi` with password `raspberry`.

Our next job is to extract the SD card from the laptop and insert it into the SD slot on the Raspberry Pi. Assuming a suitable keyboard and mouse is attached and the HDMI lead is connected to a TV or suitable screen, we can plug in the power and watch the Raspberry Pi initialise itself. The first time you use a new image extra initialisation is done and it asks you a few configuration questions. You should agree to let the system expand the root filing system to fill your SD card. If you do not you will be limited to a mere 2 Gbyte of filing system which is unlikely to be enough for your needs. The other options are up to you. You should then let the system reboot. With the default settings, the system will eventually issue a prompt looking something like the following.

```
Debian GNU/Linux wheezy/sid raspberrypi tty1

raspberrypi login:
```

You should respond by typing the user name `pi` remembering to press the Enter key. It will then ask for the password and your response should be: `raspberry`, again remembering to press the Enter key. It will then output about 6 lines ending with a Linux shell prompt such as the following:

```
Debian GNU/Linux 6.0 raspberrypi tty1

pi@raspberrypi:~$ 
```

If you get this far, you are now in business and can begin to use Linux on your Raspberry Pi. Well done!

If your Raspberry Pi was connected to the internet, it will have automatically set the time and date, but if not you should correct the time using the `sudo date` command as shown below.

```
pi@raspberrypi:~$ date
Tue Apr 17 14:15:04 BST 2012
pi@raspberrypi:~$ sudo date --set="2012-04-23 12:27"
Mon Apr 23 12:27:00 BST 2012
pi@raspberrypi:~$ date
Mon Apr 23 12:27:04 BST 2012
pi@raspberrypi:~$
```
2.1 A More Recent SD Card Image

Since the Raspberry Pi SD card image is repeatedly upgraded, I have recently (October 2014) re-installed the wheezy-raspbian image on a 4Gbyte SD card. The console session was as follows. You will see that is close the description above.

solestreet:$ sha1sum 2014-09-09-wheezy-raspbian.zip
951a9092dd160ea06195963d1af4b720558ed84  2014-09-09-wheezy-raspbian.zip
solestreet:$
solestreet:$ unzip 2014-09-09-wheezy-raspbian.zip
Archive:  2014-09-09-wheezy-raspbian.zip
    inflating: 2014-09-09-wheezy-raspbian.img
solestreet:$ ls -lrt *.img
-rw------- 1 mr10 mr10 1939865600 Jul 15 2012 2012-07-15-wheezy-raspbian.img
-rw------- 1 mr10 mr10 3965190144 Feb 13 2013 img130213.img
-rw------- 1 mr10 mr10 1939865600 May 25 2013 2013-05-25-wheezy-raspbian.img
-rw------- 1 mr10 mr10 4031774720 Oct 14 2013 sdcard14-10-13.img
-rw------- 1 mr10 mr10 3276800000 Sep 9 09:42 2014-09-09-wheezy-raspbian.img
solestreet:$
solestreet:$ df
Filesystem  1K-blocks  Used  Available  Use% Mounted on
/dev/sda6  32274308  24446852  6187984  80%  /
udev       760316     4  760312  1%  /dev
tmpfs      153524     880  152644  1%  /run
none       5120       8  5112  1%  /run/lock
none       767600     80  767520  1%  /run/shm
/dev/sda7  100148   53544  41433  57%  /boot
/dev/sda2  4184772  3245360  939412  78%  /dose
/dev/sdb1  488383484 35674332 452709152  8%  /media/TOSHIBA EXT
solestreet:$
solestreet:$ df
Filesystem  1K-blocks  Used  Available  Use% Mounted on
/dev/sda6  32274308  24446900  6187936  80%  /
udev       760316     4  760312  1%  /dev
tmpfs      153524    896  152628  1%  /run
none       5120       8  5112  1%  /run/lock
none       767600     80  767520  1%  /run/shm
/dev/sda7  100148   53544  41433  57%  /boot
/dev/sda2  4184772  3245360  939412  78%  /dose
/dev/sdb1  488383484 35674336 452709148  8%  /media/TOSHIBA EXT
/dev/mmcblk0p1  76186   28089  48097  37%  /media/95F5-0D7A
/dev/mmcblk0p2  3599168 1671940 1744512 49%  /media/18c27e44-ad29-4264-
solestreet:$
solestreet:$ umount /dev/mmcblk0p1
solestreet:$ umount /dev/mmcblk0p2
2.1. A MORE RECENT SD CARD IMAGE

solestreet:
solestreet: $ sudo dd bs=4M if=2014-09-09-wheezy-raspbian.img of=/dev/mmcblk0
[sudo] password for mr10:
781+1 records in
781+1 records out
3276800000 bytes (3.3 GB) copied, 597.799 s, 5.5 MB/s
solestreet:
solestreet:
solestreet: $ sync
solestreet:$
Chapter 3

Introduction to Linux

Assuming that you have successfully logged in to the Raspberry Pi as user pi and have the time and date correctly set you should be looking at a bash prompt such as:

```
pi@raspberrypi:$
```

This line is inviting you to type in a command to the bash shell. If you press the Enter key several times, it will repeatedly respond with the prompt. Shell commands are lines of text with the first word being the command name and later words being arguments supplied to the given command. For instance, if you type `echo hello` the command name is `echo` and its argument is `hello`. If you then press the Enter key, the machine will load and run the `echo` command outputing its argument as shown below.

```
pi@raspberrypi:$ echo hello
hello
pi@raspberrypi:$
```

After doing that, the shell is again waiting for a command.

Errors are common while typing in commands and the shell is helpful in allowing you to correct such mistakes before they are executed. Suppose you typed `echo hello` without a space between the command name and its argument, you could delete the last five characters by pressing the backspace key (often labelled `<-BkSp`) five times then press the space bar followed by `hello`. Alternatively, you could press the left arrow key five times to position the cursor over the `h` of `hello`. Pressing the space bar now will insert a space before the `h` and pressing the Enter key will now cause the corrected command to be executed.

The shell remembers commands you have recently executed and you can search through them using the up and down arrow keys. So rather typing
echo hello again, you can find it by pressing the up arrow key once and execute it by pressing the Enter key. Believe it or not, this is an incredibly useful feature.

We will now look at a few shell commands that you are likely to find useful. Firstly, there is the command date which outputs information you might expect. But if the output is wrong, the time and date should be corrected using the sudo date command shown in the previous chapter.

When you have finished using the computer, it is important to close it down properly by issuing the command sudo shutdown -h now and wait until the machine says it has halted.

There are literally hundreds of shell command available in Linux and many of them are held in the directories /bin and /usr/bin. You can see them by typing ls /bin and ls /usr/bin. But don’t be frightened, you will only need to know about perhaps 10 or 15 of them to make effective use of Linux. Linux is to a large extent self documented, and it is possible to learn what commands do using the man command. This is a rather sophisticated command that will display manual pages describing almost any command available in the system. The output is primarily aimed and professional users and is highly detailed and often incomprehensible to beginners, but you should just try it once to see the kind of information that is available. Try typing man echo. This give a detailed description of the echo command which you can step through using the up and down arrow keys and the space bar. To exit from the man command, type q. As an example of a really long and complicated command description, try man bash and repeatedly press the space bar until you get tired, remembering to press q to return to the shell. Again don’t be frightened by what you have just seen, you will only be using a tiny subset of the features available in bash and this document will show you the ones you are most likely to use.

Sometimes you want to do something but don’t know the name of the command to use. The man -k command can be helpful in this situation, but it is not always as helpful as you would like. When I first started to use Linux, many years ago, I wanted to delete a file. On previous systems I had used, commands such as del, delete or erase had done the job. Typing man -k delete lists about 13 commands that have something to do with deletion but none of the suggested commands would actually delete a file. In Linux deleting a file is called removal and is performed by the rm command. It appears in the rather long list generated by man -k remove.

The whoami outputs your user identifier. On the Raspberry Pi this is likely to be pi.

As has been seen the date command will will either generate the date and time or let you set the date.

The command cal 2012 will output a calendar for the year 2012. As an interesting oddity try cal 1752 since this was the year in which some days in
September were deleted when there was a switch from the Julian to the Gregorian calendar. Type `man cal` for details.

To execute a command that requires special privileges, you should precede it by `sudo`. It will normally require you to type in a password before it will execute the given command.

Many other commands are associated with files and the filing system. Some of these are described in the next section.

## 3.1 The Filing System

As we have seen, the SD card holds the image of the Linux system including the built-in shell commands and much more, but it also holds data that you can create. This data is held in files and continues to exist for use on another day, even after you turn the computer off. Files have names and are grouped in directories (often called folders). They typically contain text that can be output to the screen, but files are frequently used for other purposes. The `echo` command, used above, is a file but not a text file. It is actually a program containing a long and complicated sequence of instructions for the computer to obey in order to output its argument to the screen. At this stage it may seem like magic, but after reading this document you will hopefully have a better understanding of how programs are written and how they work.

Directories can contain other directories as well as files and so it is natural to think of the filing system as a tree of files (the leaves) and directories (the branches). At the lowest level is the root which is referred to by the special name `/`. We can list the contents of this using the command `ls /` as can be seen below.

```
pi@raspberrypi:$ ls /
bin  dev  lib  opt  sbin  srv  usr
boot  etc  media  proc  sd  sys  var
Desktop  home  mnt  root  selinux  tmp
pi@raspberrypi:$
```

It turns out that all the items in the root directory are themselves directories mostly belonging to the system. As can be seen, one is called `home` which contains the so called home directories of all users permitted to use this computer. Currently there is only one user called `pi` setup. We can show this by listing the contents of `home`.

```
pi@raspberrypi:$ ls /home
pi
pi@raspberrypi:$
```
We can also list the contents of the pi directory by the following.

```
pi@raspberrypi:$ ls /home/pi
```

This indicates that it is apparently empty. However, it does contain files whose names start with dots (\`.\`) that are normally hidden. These can be seen using the `-a` option as in:

```
pi@raspberrypi:$ ls -a /home/pi
  .  ..  .bash_history  .config  .lesshst
```

An absolute file name is a sequence of names separated by slashes (\`/\`) and starting with a slash. Such compound names can become quite long. For instance the full file name of the `echo` command is `/usr/bin/echo` as can be found using the `which` command. To reduce the need to frequently have to type long names, Linux has the concept of a current working directory. The absolute name of this directory can be found using the `pwd` command as in:

```
pi@raspberrypi:$ pwd
/home/pi
```

File names not starting with a slash are called relative file names and are interpreted as files within the current working directory. In this case, it is as though `/home/pi/` is prepended to the relative file name. You can change the current directory using the `cd` command, as the following sequence of commands shows.

```
pi@raspberrypi:$ cd /usr/local/lib
pi@raspberrypi:/usr/local/lib$ pwd
/usr/local/lib
pi@raspberrypi:/usr/local/lib$ cd
pi@raspberrypi:/usr/local/lib$ cd /home/pi
```

A few more Linux commands relating to files will be given in the next chapter after you have installed the BCPL system.
3.2 The Desktop

After you have logged in to the Raspberry Pi (typically as user pi with password raspberry), you will probably find yourself connected to a bash shell waiting for you to enter Linux commands. It is normally more convenient to work within a graphics session since this allows you to interact with several programs using separate windows. To start a graphics session type the command startx. After about 10 seconds you will be in a graphics session. You can then use the mouse to move about the screen and press the mouse buttons to cause actions to take place.

At the very bottom of the screen there are some tiny icons that are particularly useful. If you move the mouse pointer over one of them and wait a second, it will probably bring up a tiny message reminding you what the icon is for. The little red icon at the bottom right of the screen allows you to logout of the graphics session, returning to the original bash shell. The reminder message for this icon just says logout. A little further to the left is an icon showing the current time. If you place the mouse pointer over it, it will tell you today’s date. Provided you are connected to the internet or you have set the time and date manually, the displayed date should be correct.

Two icons at the bottom near the left side allow you to quickly switch between two separate desktops (Desktop 1 and Desktop 2). This is particularly useful if you want quick access to many windows. Perhaps, one for editing, one for compilations, one for running compiled programs in, one for web browsing, etc, etc. The icon at the bottom left looks like a white bird with a forked tail. If you click the left mouse button on this, it brings up a menu containing about nine items such as Accessories, Education, internet, Programming, and several others. For many of these, if you place the mouse pointer over them they bring up sub menus. You can explore these menus using either the mouse or the arrow keys. Suppose you highlight the Accessories menu item, pressing Right Arrow will highlight the first item in the Accessories’ sub menu. You can move up and down this sub menu with the Up and Down Arrow keys, and if you select Leafpad, say, and press Enter, a window will appear that allows you to create and edit text files. This is a fairly primitive editor similar to Notepad on computers running Windows.

On the left side of the screen, you should find a column of larger icons for commonly used applications. Probably the most important ones for our purposes are Midori a web browser and LXTerminal which creates a window allowing you to interact with a bash shell. If you place the mouse pointer over the LXTerminal icon and then click the left mouse button twice quickly (within about half a second), a window will appear with a bash shell prompt. You can test it by typing commands such as echo hello or date. The top line of the window is called the Title Bar. At its centre will be the title, typically pi@raspberry:~. If you place the mouse pointer in the title bar and hold down the left button you will find you can drag the window to a new position on the screen. If you place
the mouse pointer carefully at the bottom right corner of the window, the shape of the pointer should change to one looking like an arrow pointing down and to the right. If you now hold down the left button you will be able to drag the bottom right corner of the window to a new position. This allows you to change the size and shape of the window.

Just below the title bar is a menu bar typically holding items like File, Edit, Tabs and Help. If you place the pointer over the Edit item and press the left button, a menu will appear. Select the item named Preferences by highlighting it and press the left button. This will bring up a dialog box that allows you to modify various properties of the window, such as the background and foreground colours. I tend to prefer a background of darkish blue and a foreground of a light blue-green colour. Choose any colours you like but do not make them the same or your text will be invisible!

You can create several LXTerminals by double clicking the LXTerminal icon several times. If you move them around you will find some can be partially obscured by others, just like pages of paper on a desk. To bring a window to the top, just place the mouse pointer anywhere on it and click the left button. This is said to also bring the window into focus which means that input from the keyboard will be directed at it. You can thus have several bash sessions running simultaneously, and you can move from one to another just by moving the mouse and clicking.

3.3 Midori

If you double click on the Midori icon, it will bring up a window containing the Midori web browser. This allows you to follow links to almost any web page in the world. The only problem is to know what to type. If you happen to know the exact name (or URL) of the page you want to display, you can type it in carefully in the main text field just below the Midori title bar. Such URLs normally start with http://www., for instance, try typing http://www.cl.cam.ac.uk/~mr10 and press Enter. This should bring up my Home Page. It is however usually easier to find web pages by giving keywords to a search engine. Such keywords can be typed in the smaller text field to the right of the main URL field in Midori. But first I would suggest you select Google as your search engine since this is my favourite. To do this click on the little icon at the left hand end of the text field for keywords. This will bring up a menu of possible search engines, and you should click on Google. Now typing some keywords such as vi tutorial and press Enter. Google will respond with many links to web pages that relate to the keywords. Clicking on one of these will open that page. This is a good way to find documentation and tutorials on almost any topic you are interested in. This particular request will help you with the vi editor briefly summarised in the next section.
3.4 Editing Files

In order to program you are going to have to input and edit text files representing the programs you are going to write. There are many possible editor programs available for this but I will only mentions three of them. First is Leafpad mentioned above. It is easy to use but rather primitive and I do not recommend it for editing programs. The next is vi which is small, efficient and liked by a surprising number of professional programmers. It has good tutorials on the web, but the version typically installed on the Raspberry Pi has no built in documentation. My favourite text editor is called emacs. It is large and sophisticated and much liked by many professional (just as Linux is). It has plenty of build in documentation and is an effective editor even if you use only a tiny proportion of its facilities. The next two sections will give brief instructions on how to use vi and emacs.

3.5 vi

This section contains only a brief introduction to the vi editor since there are several excellent tutorials on vi some of which are videos. Try doing a web search on vi tutorial.

Although I prefer to use the emacs editor, vi is sometime useful since it is a small program and simple to use. To enter vi, type the command vi filename where filename is the name of a file you wish to create or edit. If you omit the filename, you can still create a file but must give the filename when you write it to disc (using :w filename). When vi is running it displays some of the text of the file being edited in a window with with a flashing character indicating the current cursor position. The cursor can be moved using the arrow keys, or by pressing h, j, k or l to move the cursor left, down, up or right, respectively.

vi has two modes: command and insert. When in insert mode characters typed on the keyboard are inserted into the current file. Pressing the ESC character causes vi to return to command mode. In the description that follows text represents characters typed in in insert mode, ch represents a single character, Esc represents the escape key and Ret represents the Enter key. Some of the vi commands are as follows.
CHAPTER 3. INTRODUCTION TO LINUX

Move the cursor to the first non blank character of the current line.

Move the cursor to the end of the current line.

Insert text just before the cursor.

Insert text just after the cursor.

Create a blank line just after the current line and insert text at its start.

Create a blank line just above the current line and insert text at its start.

Join the current line with the next one.

Delete the character at the current cursor position.

Delete the character before the current cursor position.

Delete the current line putting it in the deletion buffer.

Insert (or paste) the text in the deletion buffer to just before the cursor position.

Undo the last command.

Scan forwards from the current cursor position for the nearest occurrence of text.

Scan backwards from the current cursor position for the nearest occurrence of text.

Repeat the last / or ? command.

Save the current file and exit from vi.

Save the current file and exit from vi.

Exit from vi without saving the file.

Write the current file to disc.

Write the current file to disc using the specified filename.

Substitute all occurrences of pattern in the current line by replace. It g is omitted only the first occurrence is replaced.

Perform the substitution on all lines between line numbers n and m. The last line number can be written as $.

The vi editor has many more features, but the above selection is sufficient for most needs.
3.6 emacs

The emacs editor is highly sophisticated and much loved by many professional programmers and I recommend that you use it. You can use it effectively using a tiny minority of its available commands, and so it should not take long to learn.

It is normally best to use `emacs` once you are in the graphics desktop, i.e., after you have executed the `startx` command immediately after logging in. So from now on I assume that you have started a graphics desktop session (using `startx`) and have opened an `LXTerminal` session, so that you can execute `bash` commands.

The Linux image you copied to your SD card probably did not include the `emacs` editor, so you will have to install it using `apt-get` or `synaptic`. Try typing:

```
sudo apt-get install emacs
```

If this works (and it should), you will be able to enter `emacs` by typing

```
emacs &
```

This will create a new window on the desktop for `emacs` to run in.

Before learning how to use `emacs`, I suggest you move to the next chapter and install the BCPL system. Once that is working come back here to see how to use `emacs` to edit files.

You should first set up some initialisation files so that `emacs` knows about BCPL mode which will automatically colour BCPL reserved words, strings, comments and other syntactic items appropriately. So, after installing BCPL, type:

```
cd
cp -r $BCPLROOT/Elisp .
cp $BCPLROOT/.emacs .
```

The next time you enter `emacs`, it will use BCPL mode when editing BCPL source files with extensions `.b` or `.h`. This makes BCPL source code much more readable.

As I said above, you can create an `emacs` window by typing the `emacs &` command. When the window appears, move the mouse to it and click to bring it into focus. Input from the keyboard will now be directed to `emacs`.

Many `emacs` commands require the Ctrl key to be held down. For instance, holding down Ctrl and pressing e will move the cursor to the end of the current line. We will use the notation C-e to denote this operation. To illustrate what `emacs` can do, we will edit the `hello.b` program in the `~/distribution/BCPL/cintcode/com/` directory. To edit this file, type C-x C-f and then type `~/distribution/BCPL/cintcode/com/hello.b` followed by Enter. This should put the following text (in colour) near the top of the window.

```
As I said above, you can create an `emacs` window by typing the `emacs &` command. When the window appears, move the mouse to it and click to bring it into focus. Input from the keyboard will now be directed to `emacs`.

Many `emacs` commands require the Ctrl key to be held down. For instance, holding down Ctrl and pressing e will move the cursor to the end of the current line. We will use the notation C-e to denote this operation. To illustrate what `emacs` can do, we will edit the `hello.b` program in the `~/distribution/BCPL/cintcode/com/` directory. To edit this file, type C-x C-f and then type `~/distribution/BCPL/cintcode/com/hello.b` followed by Enter. This should put the following text (in colour) near the top of the window.
```
GET "libhdr"

LET start() = VALOF
{ writef("Hello World!*n")
  RESULTIS 0
}

The cursor position will be marked by a small flashing rectangle. The cursor can be moved UP, DOWN, LEFT and RIGHT using the arrow keys. It can also be moved to the end of the current line by typing C-e, and to the beginning of the current line by C-a. Use these keys to position the cursor over the w of writef and press C-k C-k. The first deletes (or kills) the text from the cursor position to the end of the line, and the second kills the newline character at the end of the line. The killed text is not lost but held in a stack of killed items. Type C-y will recover what has just been killed, and typing C-y again will recover it again. The text should now be as follows.

GET "libhdr"

LET start() = VALOF
{ writef("Hello World!*n")
writef("Hello World!*n")
  RESULTIS 0
}

Move the cursor to the w of the second writef and press the space bar twice will correct the indentation. Now move the cursor to the H of the second Hello World! and press C-d 12 times to delete Hello World!. Now insert some text by typing: How are you?. The text should now be as follows.

GET "libhdr"

LET start() = VALOF
{ writes("Hello World!*n")
writes("Hello World!*n")
  RESULTIS 0
}

Now write this back to the file by typing C-s. To test that the editing was successful, click on the LXTerminal window and type: cat com/hello.b. It should output the edited version of the hello.b program. You can now compile and run it by typing:
cintsys
c bc hello
hello

The command c combines the file bc and the argument hello to form a command sequence that invokes the BCPL compiler to translate the source code com/hello.b into a form suitable for execution which it stores in cin/hello. You can inspect the source and compiled forms by typing the commands type com/hello.b and type cin/hello. Although at this stage the compiled form will be unintelligible. The file bc is called a command script and is one of many designed to make the BCPL cintcode system easier to use.

Now return to the emacs window by clicking on it. We can move the cursor to the start of the file by typing C-Shift-Home and the end by C-Shift-End. Now move to the start of the file (C-Shift-Home). If we want to find some text in the file type C-s followed by some characters such as al and observe how the cursor moves. You will see that the match ignores whether letters are in lower or upper case. If you press BkSp the cursor moves back to the a of start and pressing r will highlight the ar of start, and also the ar of are, two lines below. You can move to this word by either typing C-s again, or by increasing the length of our pattern by typing e. Pressing BkSp removes e from our pattern and returns the cursor to the just after the r of start. Just as C-s performs a forward search, C-r performs a backwards search. Practice using these commands until you are satisfied you can easily find anything you want in the file. To leave this interactive searching mode press Enter.

Suppose we wished to change every occurrence of writef to writes. We could do this by pressing C-Shift-Home to get to the top of the file. Then press Esc followed by % to enter the interactive replacement command. It will invite you to type in the text you wish to replace, namely writef. You terminate this by pressing Enter. It then invites you to give the replacement text, to which you type writes followed by Enter. This causes the first occurrence of writef to be highlighted, waiting for a response. If you press the Space Bar it will replace this occurrence and move on to the next. If you press BkSp it will just move on to the next, and if you press Enter it will leave interactive replace mode.

The command C-g aborts whatever you were doing and returns you to the normal editing state. This turns out to be more useful that you might imagine.

A log of changes is kept by emacs and this is used by C-_ to undo the latest change. Multiple C-_s can undo several changes.

If you want to close the emacs window, type C-x C-c.

Splitting the screen is useful if you want to edit two files at the same time. To do this type C-x 2 and to return to a single screen type C-x 1. C-x 3 will split the screen vertically putting the sub-windows side by side.

There is a sophisticated online help facility. Type C-h to enter it. To find out what to do next, type ?. This will split the window into two parts filling the lower
half with a decription of the possible help commands that are available. You can move the cursor into this sub-window by pointing the mouse into it and clicking. Alternatively, you can type C-x o. Once there, you can navigate through the help text using the same commands you use when editing a file.

To obtain a list of all key bindings type C-h b. If you scroll down to C-x C-f (or search for it) you will find it is bound to the find-file command. C-h f find-file will output a description of the command.

Although the commands I have described so far allows you to create and edit files, you will find exploring the emacs help system will allow you to use emacs even more effectively.
Chapter 4

The BCPL Cintcode System

The quick way to install the BCPL system is to download `bcpl.tgz` into your home directory (`/home/pi`) and then type the following sequence of commands.

```bash
cd
mkdir distribution
cd distribution
tar zxf ../bcpl.tgz
cd BCPL/cintcode
  . os/linux/setbcplenv
make clean
make -f MakefileRaspi
c compall

cp -r Elisp $HOME -- to configure emacs
cp .emacs $HOME -- to configure emacs
```

But if you wish to understand what is going on, you should read the next section. But, while you are here, you might as well install the BCPL Cintpos systems as well. To do this, download `cintpos.tgz` into your home directory and then type the following.

```bash
cd
mkdir distribution
cd distribution
tar zxf ../cintpos.tgz
cd Cintpos/cintpos
make clean
make -f MakefileRaspi
c compall
```
This is an interpretive implementation of the Tripos Portable Operating System which is described in the BCPL manual available from my home page.

4.1 Installation of BCPL

To install the BCPL System on the Raspberry Pi you must first obtain a copy of the file bcpl.tgz which is available via my home page (www.cl.cam.ac.uk/users/mr). Near the top of this page, under the heading “Shortcut to the main packages”, you will find a link to bcpl.tgz. Right clicking on this link should bring up a menu one of whose items will save bcpl.tgz as a file on your computer. If your Raspberry Pi is connected to the internet, you can do this using the Midori web browser and save to file in your home directory (/home/pi). Failing that, find a computer that has an SD card slot and is connected to the internet, and copy bcpl.tgz into /home/pi on your SD card. When you next login to the Raspberry Pi you will find bcpl.tgz in your home directory. To check it is there, run the following commands.

```
pi@raspberrypi:~$ cd
pi@raspberrypi:~$ pwd
/home/pi
pi@raspberrypi:~$ ls -l
-rwxrwx--- 1 pi pi 10300397 Apr 23 15:20 bcpl.tgz
pi@raspberrypi:~$
```

You can install BCPL anywhere you like but I would strongly recommend that the first time you install it you place it in exactly the same location that I use on my laptop since this will allow you to set the system up without having to edit any of the configuration files. I therefore suggest you follow the next few steps exactly.

1) Create a directory called distribution, make it the current directory and decompress the tgz file into it.

```
pi@raspberrypi:~$ mkdir distribution
pi@raspberrypi:~$ cd distribution
pi@raspberrypi:~/distribution$ tar zxvf ..bcpl.tgz
--- Lots of output showing the names of all files of the BCPL system
pi@raspberrypi:~/distribution$
```

2) List the contents of the current directory, the BCPL directory and BCPL/cintcode.
4.1. INSTALLATION OF BCPL

pi@raspberrypi:~$ ls
BCPL
pi@raspberrypi:~$ ls BCPL

bcplprogs cintcode Makefile natbcpl README TGZDATE xfiles
pi@raspberrypi:~$ ls BCPL/cintcode
--- Lots of files and directories including
g com sysb sysc os
pi@raspberrypi:~$ cd BCPL/cintcode

3) Now change to directory BCPL/cintcode and type the following commands.

pi@raspberrypi:~/distribution$ cd BCPL/cintcode
pi@raspberrypi:~/distribution/BCPL/cintcode$ . os/linux/setbcplenv
pi@raspberrypi:~/distribution/BCPL/cintcode$ make clean
pi@raspberrypi:~/distribution/BCPL/cintcode$ make -f MakefileRaspi
--- Lots of output showing the BCPL system being built
--- ending with something like:
bin/cintsys

BCPL Cintcode System (24 Jan 2012)
0.000>

The file os/linux/setbcplenv is a shell script that sets up BCPL environment variables such as BCPLROOT and BCPLPATH telling the system where BCPL has been installed. The important part of setbcplenv is as follows.

export BCPLROOT=$HOME/distribution/BCPL/cintcode
export BCPLPATH=$BCPLROOT/cin
export BCPLHDERS=$BCPLROOT/g
export BCPLSCRIPTS=$BCPLROOT/s

export POSROOT=$HOME/distribution/Cintpos/cintpos
export POSPATH=$POSROOT/cin
export POSHDERS=$POSROOT/g
export POSSCRIPTS=$POSROOT/s

export PATH=$PATH:$BCPLROOT/bin:$POSROOT/bin

When run using the dot (.) command, it defines the required shell environment variables and updates the PATH variable to include the bin directories where cintsys and cintpos live. Cintpos is a portable operating system implemented
in BCPL but not covered by this document. You can test whether the script has run correctly by typing `echo $BCPLROOT` or `printenv`.

You need to run this script every time you login to the Raspberry Pi if you want to use BCPL. It would therefore be useful for this to happen automatically every time you login. The `bash` shell runs some initialising shell scripts when it starts up, as is described in the manual pages generated by the `man bash` commands. Some of the scripts are provided by the system and live in the `/etc` directory but others live in the user's home directory. The possible file names are `.bash_profile`, `.bash_login`, `.profile` and possibly `.bashrc`. You can see which of these dot files are in your home directory by typing:

```
cd
ls -a
```

You should add the following line onto the end of one of these files.

```
. $HOME/distribution/BCPL/cintcode/os/linux/setbcplenv
```

On the version of Linux I am using on the Raspberry Pi, the script `.profile` calls `.bashrc`, and so I added the line to the end of the file `.bashrc`. To do this, I typed

```
cd
vi .bashrc
```

This caused me to get into the `vi` editor editing the file `.bashrc`. Now using the down-arrow key several times I got to the last line of the file and typed the lowercase letter o. This got me into input mode allowing me to add text to the end of the file. I then typed the line

```
. $HOME/distribution/BCPL/cintcode/os/linux/setbcplenv
```

terminated by pressing both the Enter and Esc keys. This returned me to edit mode. Finally I typed: `:wq` and pressed Enter, to write the edited file back to the filing system. To check that I edited the file correctly, I typed `cat .bashrc` and looked carefully at its last line.

After making this change to an appropriate script file, you should test it by logging out of the Raspberry Pi and login again. To logout, type

```
sudo shutdown -h now
```
But, if you are in the graphics environment, you should leave this first by clicking on the little red icon at the bottom right hand corner of the screen.

The next time you login to the Raspberry Pi, you should find that the BCPL environment variables have been defined automatically. To make sure, type: echo $BCPLROOT.

The commands make clean and make -f MakefileRaspi remove unwanted files and causes the entire BCPL Cintcode System to be rebuilt from scratch. This involves the compilation of several C programs and the BCPL compilation of every BCPL program in the system. The last line 0.000> is a prompt from the BCPL Command Language Interpreter inviting you to type a command. If this all works you will now be in business and can begin to use BCPL.

As confirmation that the system really is working, type in the following commands.

0.000> echo hello
hello
0.000> type com/echo.b
SECTION "ECHO"

GET "libhdr"

LET start() = VALOF
{ LET tostream = 0
  LET toname = 0
  LET appending = ?
  LET nonewline = ?
  LET text = 0
  LET argv = VEC 80

  IF rdargs("TEXT,TO/K,APPEND/S,N/S", argv, 80)=0 DO
    { writes("Bad argument for ECHO*n")
      RESULTIS 20
    }

  IF argv!0 DO text := argv!0 // TEXT
  IF argv!1 DO toname := argv!1 // TO/K
  appending := argv!2 // APPEND/S
  nonewline := argv!3 // N/S

  IF toname DO
    { TEST appending
      THEN tostream := findappend(toname)
      ELSE tostream := findoutput(toname)
    }
UNLESS tostream DO
{ writef("Unable to open file: %s*n", toname)
  result2 := 100
  RESULTIS 20
}
selectoutput(tostream)

IF text DO writes(text)
UNLESS nonewline DO newline()

IF tostream DO endstream(tostream)
RESULTIS 0

0.260> bcpl com/echo.b to junk

BCPL (1 Feb 2011)
Code size 244 bytes

0.130> junk hello
hello
0.020> bcpl com/bcpl.b to junk

BCPL (1 Feb 2011)
Code size 22156 bytes
Code size 12500 bytes

1.210> junk com/bcpl.b to junk

BCPL (1 Feb 2011)
Code size 22156 bytes
Code size 12500 bytes

1.210> logout

pi@raspberrypi:/distribution/BCPL/cintcode$

The echo command just outputs its argument. The type command outputs the BCPL source code of the echo command and the bcpl command compiles it into a file called junk. This is then executed as the junk command, demonstrating that it behaves exactly as the echo command did. Next we use the bcpl command to compile the BCPL compiler whose source code is in com/bcpl.b. This overwrites the file junk which is then used to compile the compiler again with identical effect. The prompt contains the time in seconds of the previous command, so we see that compiling the BCPL compiler takes a mere 1.2 seconds. The logout command
leaves the BCPL system and returns to the `bash` shell. To re-enter the BCPL system type the command `cintsys`.

If you plan to use the `emacs` editor (which I recommend) you should set up its initialisation files so that it knows about BCPL mode which will automatically colour BCPL reserved words, strings, comments and other syntactic items appropriately. To do this type:

```bash
cd
cp -r $BCPLROOT/Elisp .
cp $BCPLROOT/.emacs .
```

The next time you enter `emacs` it will used BCPL mode when editing BCPL source files with extensions `.b` or `.h`. This makes editing such files much more friendly.

We will now look at a few more Linux commands. The `bash` program looks up commands in a sequence of directories called a path. This sequence can be inspected by looking at the value of the `PATH` environment variable as shown by:

```
pi@raspberrypi:~$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:
/home/pi/distribution/BCPL/cintcode/bin:
/home/pi/distribution/Cintpos/cintpos/bin:
```

You can output an entire file to the screen by commands such as `cat com/echo.b` or you can display it one page at a time using `more` as in `more com/type.b`. The `more` program can be controlled using the Space bar, Enter key, the arrow key, `p` and `b` and many others. To quit the program type `q`.

The `cp` command copies files. For instance, `cp com/abort.b prog.b` will copy the source of the `abort` command into the current directory as file `prog.b`. You can also use `cp` to copy complete directory trees using the `-r` argument, as in `cp -r g myg`. You can test it worked by typing `ls myg`.

The `rm` command removes files as in `rm myg/libhdr.h`. It can also remove complete directory trees using the `-r` argument, as in `rm -r myg`.

We are now ready to learn how to program in BCPL and this will be done in a gentle way exploring the simple programs presented below.

## 4.2 Hello World

The BCPL system contains a huge number of BCPL programs that can be found in directories such as
You are certainly free to look at these, but it is probably best to start with some simple examples. Ever since Brian Kernighan wrote the first Hello World program in an internal Bell Laboratory memorandum about B in the mid 1970s, it has become the standard first program used in the description of most programming languages. The version for BCPL is \texttt{com/hello.b} and is as follows:

\begin{verbatim}
GET "libhdr"

LET start() = VALOF
{ writef("Hello World!*n")
  RESULTIS 0
}
\end{verbatim}

The line \texttt{GET "libhdr"} inserts a file declaring all sorts of library functions, variables and constants needed by most programs. The actual file inserted is \texttt{cintcode/g/libhdr.h} but there is no need to look at it yet. The next line is the heading of a function called \texttt{start} which, by convention, is the first function of a program to be executed. The body of \texttt{start} is a \texttt{VALOF} block that contains commands to be executed terminated by a \texttt{RESULTIS} command that specifies the result. In this case a result of zero indicates that the \texttt{hello} program terminated successfully. But before returning, it executes \texttt{writef("Hello World!*n")} which output the characters \texttt{Hello World!} followed by a newline (represented by the escape sequence \texttt{*n}).

This program can be compiled using the \texttt{bcpl} command to form a compiled program called \texttt{junk} which is then executed.

\begin{verbatim}
0.000> bcpl com/hello.b to junk

BCPL (1 Feb 2011)
Code size = 60 bytes
0.100>
0.000> junk
Hello World!
0.020>
\end{verbatim}

Compiled commands are normally placed in a directory called \texttt{cin}, and, for convenience, there is a script called \texttt{bc} to simplify the compilation of such commands. If we regard \texttt{hello.b} as a command, it can be compiled using the \texttt{c bc} \texttt{hello} command as follows.
The hello command can now be executed.

```
0.000> hello
Hello World!
0.020>
```

The script file `bc` is as follows

```bash
#!/home/mr/distribution/BCPL/cintcode/cintsys -s
.k file/a, arg
echo "bcpl com/<file>.b to cin/<file> hdrs BCPLHDRS <arg>"
bclpl com/<file>.b to cin/<file> hdrs BCPLHDRS <arg>
```

But at this stage there is no need to understand how it works.

For convenience, all the BCPL programs covered in this document can be found in the directory `BCPL/bcplprogs/raspi` of the standard BCPL distribution. If you make this your current directory, you can inspect, compile and run these programs using commands such as the following.

```
pi@raspberpi:~$ cd ~/distribution/BCPL/bcplprogs/raspi
pi@raspberpi:~/distribution/BCPL/bcplprogs/raspi$ cintsys
```

```
0.000> type hello.b
GET "libhdr"

LET start() = VALOF
{ writef("Hello World!*n")
  RESULTIS 0
}
0.020> c b hello
bcpl hello.b to hello hdrs BCPLHDRS
```

BCPL (1 Feb 2011)
The command script `b` used here is similar to `bc` used earlier by \texttt{expects} the source program to be in the current directory and place the compiled version in the same directory.

The next program we will study concerns the Fibonacci sequence of numbers.

### 4.3 Fibonacci

Leonardo Fibonacci lived in Italy near Pisa dying in about 1250 AD aged around 80. He is regarded by some as “the most talented western mathematician of the Middle Ages”. He is perhaps best known for the sequence of numbers named after him. This sequence has some extraordinary properties and has excited mathematicians ever since. The sequence starts as follows: 0, 1, 1, 2, 3, 5, 8, 13, 21,... with every number being the sum of the preceding two. For instance 2+3 gives 5, and 3+5 gives 8 etc. These numbers can be given positions with the convention that the first in the sequence is at position zero. The following table shows the positions and values of the first few numbers in the sequence.

<table>
<thead>
<tr>
<th>position</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>13</td>
<td>21</td>
</tr>
</tbody>
</table>

A program to print out the positions and values of some numbers in this sequence is called \texttt{fib1.b} and is shown in Figure 4.1. Text between `//` and the end of the line is called a comment and is designed to help the reader understand what is going on. Comments have no effect on the meaning of a program and are ignored by the compiler. This program can be compiled and run as follows.

```
0.020> c b fib1
bcpl fib1.b to fib1 hdrs BCPLHDRS
```

BCPL (1 Feb 2011)

Code size = 168 bytes

```
0.030> fib1
Position 0 Value 0
Position 1 Value 1
Position 2 Value 1
0.010>
```
GET "libhdr"

LET start() = VALOF
{ LET a = 0  // a and b hold two consecutive Fibonacci numbers
  LET b = 1
  LET c = a+b // c holds the Fibonacci number after b, namely a+b
  LET i = 0  // The position of the Fibonacci number held in a

  writef("Position %n Value %n*n", i, a)
  a := b
  b := c
  c := a+b
  i := i+1

  writef("Position %n Value %n*n", i, a)
  a := b
  b := c
  c := a+b
  i := i+1

  writef("Position %n Value %n*n", i, a)
  a := b
  b := c
  c := a+b
  i := i+1

  RESULTIS 0
}

Figure 4.1: The file fib1.b

At the beginning of the body of the function start we see the declaration LET a = 0. This allocates space in the memory of the computer which you can think of as a pigeon hole which can hold a number. It has the name a and is initialised with the number zero. Similarly, LET b = 1 allocates a pigeon hole for b initialised to 1. The third declaration LET c = a+b allocates a pigeon hole for c initialising it to the sum of the numbers in a and b. From now on, rather than talking about pigeon holes, we will usually describe them as variables with names a, b and c. They are called variables because, during the execution of the program, their values change. Indeed, as this program progresses, they are going to be successively set to three consecutive Fibonacci numbers further down the sequence. Initially, they hold the first three Fibonacci numbers (0, 1, 1)
with a holding the number at position zero. The declaration \texttt{LET i = 0} declares variable \texttt{i} to hold the position of the Fibonacci number in \texttt{a}. The statement

\begin{verbatim}
   writef("Position %n Value %n*n", i, a)
\end{verbatim}

outputs a line with the substitution items \texttt{%n} replaced by the numbers in variables \texttt{i} and \texttt{a}. It thus outputs the following.

Position 0  Value 0

We now want to move on to the next position in the sequence, and so we set \texttt{a} and \texttt{b} to the values currently in \texttt{b} and \texttt{c}. This is done by the assignments \texttt{a := b} and \texttt{b := c}, being careful to do these assignments in that order. We then compute the new value of \texttt{c} using \texttt{c := a+b} which essentially says: take the numbers in variables \texttt{a} and \texttt{b}, add them together and put the result in \texttt{c}. The numbers now in \texttt{a}, \texttt{b} and \texttt{c} are the three consecutive Fibonacci numbers starting at position 1. To set \texttt{i} to this new position number, we execute the statement \texttt{i := i+1} which increments \texttt{i} changing it from zero to one.

The program then executes exactly the same code two more times, outputting the following:

Position 1  Value 1
Position 2  Value 1

Finally, it executes \texttt{RESULTIS 0} causing the program to return from \texttt{start} successfully.

This program is not well written and can be improved in many ways. Its most obvious problem is that part of the program is written out three times and we should be able to find a way of writing this part once, and somehow arrange for it to be executed three times. The following code does just this.

\begin{verbatim}
GET "libhdr"

LET start() = VALOF
{ LET a = 0 // a and b hold two consecutive Fibonacci numbers
  LET b = 1
  LET c = a+b // c holds the Fibonacci number after b, namely a+b
  LET i = 0 // The position of the Fibonacci number held in a

  WHILE i<=2 DO
  { writef("Position %n Value %n*n", i, a)
    a := b
    b := c
  }

\end{verbatim}
Here the WHILE statement repeatedly executes its body so long as the value of i remains less than or equal to 2. This kind of loop is so common that many languages allow it to be coded even more compactly. Such as the following.

\begin{verbatim}
{ LET a = 0 // a and b hold two consecutive Fibonacci numbers
  LET b = 1
  LET c = a+b // c holds the Fibonacci number after b, namely a+b

  FOR i = 0 TO 2 DO
  { writef("Position %n Value %n*n", i, a)
    a := b
    b := c
    c := a+b
  }

  RESULTIS 0
}
\end{verbatim}

The FOR loop declares \texttt{i} with initial value 0, and then it repeatedly executes its body, incrementing \texttt{i} each time. This version is both more concise and more understandable.

Finally, the variable \texttt{c} is only needed very briefly when we are calculating the new value of \texttt{b}. We do not need to remember its value between iterations of the body, and so it can be declared inside the FOR loop. At the same time we can replace the separate declarations of \texttt{a} and \texttt{b} by a single simultaneous declaration. The resulting program is as follows.

\begin{verbatim}
GET "libhdr"

LET start() = VALOF
{ LET a, b = 0, 1 // a and b hold two consecutive Fibonacci numbers

  FOR i = 0 TO 2 DO
  { LET c = a+b  // c holds the Fibonacci number after b, namely a+b
    writef("Position %n Value %n*n", i, a)
  }

  RESULTIS 0
}
\end{verbatim}
a := b
b := c
}

RESULTIS 0
}

The declaration LET c = a+b is placed at the head of the block (enclosed within
{ } brackets) since such declarations are only permitted at the start of a block. An obvious advantage of this form of the program is that we can now easily change it to output the sequence up to, say, position 20.

GET "libhdr"

LET start() = VALOF
{ LET a, b = 0, 1 // a and b hold two consecutive Fibonacci numbers

FOR i = 0 TO 20 DO
{ LET c = a+b // c holds the Fibonacci number after b, namely a+b
  writef("Position %n Value %n", i, a)
  a := b
  b := c
}

RESULTIS 0
}

This gives the following output.

0.010> c b fib4
bcpl fib4.b to fib4 hdrs BCPLHDRS

BCPL (1 Feb 2011)
Code size = 92 bytes
0.020> fib4
Position 0 Value 0
Position 1 Value 1
Position 2 Value 1
Position 3 Value 2
Position 4 Value 3
Position 5 Value 5
...
4.3. FIBONACCI

Position 15 Value 610
Position 16 Value 987
Position 17 Value 1597
Position 18 Value 2584
Position 19 Value 4181
Position 20 Value 6765

The final improvement could be to arrange that the position numbers are printed in a field width of 2 and the values in a field width of, say, 12. We do this by changing the `writef` statement from

```
writef("Position %n Value %n*n", i, a}
```

to

```
writef("Position %2i Value %12i*n", i, a}
```

The effect is as follows.

Position 0 Value 0
Position 1 Value 1
Position 2 Value 1
Position 3 Value 2
Position 4 Value 3
Position 5 Value 5

... Position 15 Value 610
Position 16 Value 987
Position 17 Value 1597
Position 18 Value 2584
Position 19 Value 4181
Position 20 Value 6765

We have just seen that we can perform quite complicated calculations just using simple variables, assignments, the plus operator and `WHILE` loops. If we allow subtraction as well, we can calculate almost anything we like, such as, for example, the $n^{th}$ prime number. A prime number is only divisible by 1 and itself. The first few primes are 2, 3, 5, 7, 11 and 13. The following program outputs the 100th prime.

GET "libhdr"

LET start() = VALOF
{ LET n = 100   // The number of the prime we want
LET p = 2     // The current number we are looking at
LET count = 0 // The count of how many primes we have found

{ // Start of the main loop
  // Test whether p is prime
  // Let us assume it is prime unless proved otherwise
  LET p_is_prime = TRUE
  // Try dividing it by all numbers between 2 and p-1
  FOR d = 2 TO p-1 DO
    { // d is the next divisor to try
      // We test to see if d divides p exactly
      LET r = p // Take a copy of p
      // Keep subtracting d until r is less than d
      UNTIL r < d DO r := r - d
      // If r is now zero, d exactly divides p
      // and so p is not prime
      IF r=0 DO
        { p_is_prime := FALSE
          BREAK // Break out of the FOR loop
        }
    }
  IF p_is_prime DO
    { // We have found a prime so increment the count
      count := count + 1
      IF count = n DO
        { // We have found the prime we were looking for,
          // so print it out,
          writef("The %nth prime is %n*n", n, p)
          // and stop.
          RESULTIS 0
        }
    }
  // Test the next number
  p := p+1
} REPEAT
}

This program uses special numbers TRUE (=1) and FALSE (=0) to represent truth values. It uses an IF statement to conditionally execute some code, and it uses a BREAK command to break out of the FOR loop. The word REPEAT causes
the preceding command to be executed repeatedly. In this program the loop is terminated by \texttt{RESULTIS 0} after the \textit{n}\textsuperscript{th} prime has been output. It is terribly inefficient but it does compute the correct result on the Raspberry Pi in very little time, as can be seen below.

\begin{verbatim}
0.000> c b prime1
bcpl prime1.b to prime1 hdrs BCPLHDRS

BCPL (1 Feb 2011)
Code size = 124 bytes
0.110> prime1
The 100th prime is 541
0.080>
\end{verbatim}

If you successively change \texttt{n} to 1000, 2000 and 4000 you will find the time to compute these primes increases by nearly a factor of 5 each time. It seems to grow faster than \(n^2\) (this stands for \(n \times n\), so when \(n\) doubles the cost goes up by a factor of 4) but less fast than \(n^3\) (this stands for \(n \times n \times n\), so every time \(n\) doubles the cost goes up by a factor of 8). Such programs are said to have \textit{polynomial complexity}, and one of the challenges in programming is to find ways of computing the required result much more efficiently.

If you think polynomial complexity is bad, \textit{exponential complexity} is far worse (but sometimes useful). This is when the computation time grows at a rate of similar to \(k^n\) (every time \(n\) is increased by 1 the cost goes up by a factor of \(k\)). One problem that is thought to have exponential complexity is the following. Given an \(n\) digit decimal number, \(x\) say, that is known to be the product of two primes, find them. In a sense this is easy – just try dividing by every number between 2 and \(x - 1\). Unfortunately, there are roughly \(10^n\) to try and if \(n\) is more than about 500 it is likely to take longer than the life time of the universe to solve.

Coming back to our \(n^{th}\) prime program, we can speed it up quite a bit using additional operators available in BCPL, in particular the \texttt{MOD} operator that computes the remainder after division of one number by another. For instance \(13\ \texttt{MOD} 5 = 3\). Using the \texttt{MOD} operator the program becomes:

\begin{verbatim}
GET "libhdr"

LET start() = VALOF
{ LET n = 100    // The number of the prime we want
  LET p = 2     // The current number we are looking at
  LET count = 0  // The count of how many primes we have found
\end{verbatim}
{ // Start of the main loop
// Test whether p is prime
// Let us assume it is prime unless proved otherwise
LET p_is_prime = TRUE
// Try dividing it by all numbers between 2 and p-1

FOR d = 2 TO p-1 DO
{ // d is the next divisor to try
    // We test to see if d divides p exactly
    LET r = p MOD d
    // If r is zero, d exactly divides p
    // and so p is not prime
    IF r=0 DO
    { p_is_prime := FALSE
        BREAK // Break out of the FOR loop
    }
}

IF p_is_prime DO
{ // We have found a prime so increment the count
    count := count + 1
    IF count = n DO
    { // We have found the prime we were looking for,
        // so print it out,
        writef("The \%nth prime is \%n*\%n", n, p)
        // and stop.
        RESULTIS 0
    }
}
// Test the next number
p := p+1
} REPEAT

4.4 Multiplication Table

The following simple program (bcplprogs/raspi/multab.b) outputs the 12x12 multiplication table.

GET "libhdr"
LET start() = VALOF
{} FOR x = 1 TO 12 DO
{} newline()
{} FOR y = 1 TO 12 DO writef(" %i3", x*y)
} newline()
RESULTIS 0
}

The output it generates is as follows

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>10</td>
<td>12</td>
<td>14</td>
<td>16</td>
<td>18</td>
<td>20</td>
<td>22</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>9</td>
<td>12</td>
<td>15</td>
<td>18</td>
<td>21</td>
<td>24</td>
<td>27</td>
<td>30</td>
<td>33</td>
<td>36</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>24</td>
<td>28</td>
<td>32</td>
<td>36</td>
<td>40</td>
<td>44</td>
<td>48</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>15</td>
<td>20</td>
<td>25</td>
<td>30</td>
<td>35</td>
<td>40</td>
<td>45</td>
<td>50</td>
<td>55</td>
<td>60</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>12</td>
<td>18</td>
<td>24</td>
<td>30</td>
<td>36</td>
<td>42</td>
<td>48</td>
<td>54</td>
<td>60</td>
<td>66</td>
<td>72</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>14</td>
<td>21</td>
<td>28</td>
<td>35</td>
<td>42</td>
<td>49</td>
<td>56</td>
<td>63</td>
<td>70</td>
<td>77</td>
<td>84</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>16</td>
<td>24</td>
<td>32</td>
<td>40</td>
<td>48</td>
<td>56</td>
<td>64</td>
<td>72</td>
<td>80</td>
<td>88</td>
<td>96</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>18</td>
<td>27</td>
<td>36</td>
<td>45</td>
<td>54</td>
<td>63</td>
<td>72</td>
<td>81</td>
<td>90</td>
<td>99</td>
<td>108</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>20</td>
<td>30</td>
<td>40</td>
<td>50</td>
<td>60</td>
<td>70</td>
<td>80</td>
<td>90</td>
<td>100</td>
<td>110</td>
<td>120</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>22</td>
<td>33</td>
<td>44</td>
<td>55</td>
<td>66</td>
<td>77</td>
<td>88</td>
<td>99</td>
<td>110</td>
<td>121</td>
<td>132</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>24</td>
<td>36</td>
<td>48</td>
<td>60</td>
<td>72</td>
<td>84</td>
<td>96</td>
<td>108</td>
<td>120</td>
<td>132</td>
<td>144</td>
<td></td>
</tr>
</tbody>
</table>

Many will recognise this as the horrendous collection of 144 numbers one had to learn, often by rote, at school. Some readers will still be in the process of learning them. I have two reasons for giving this example. The first is that this program can be easily modified to output tables for other expression operators. For instance, try replacing the expression \(x \times y\) in the `writef` statement by each of \(x/y, x \mod y, x+y, x-y, x\&y, x|y, x \ XOR y\), and even \(x=y\) or \(x<y\). All these operators are described later. The second reason is that learning 144 numbers can be boring and there are a whole collection of simple tricks that help you work out the answer to any of these multiplications.

### 4.5 A Mathematician’s Approach

This section is entirely optional but the mathematics is contains is both simple and useful, so I recommend you only skip this section when you have had enough.

Rather than remembering a multitude of results, mathematicians tend to like to work things out from first principles. We all know that \(5 \times 9 = 45\), but our memory is not always perfect and we might accidentally think \(5 \times 9 = 54\) and have little to help us recognise that we have the wrong answer. A mathematician
looking 5 × 9 thinks of the cunning ways of multiplying by 5 and by 9. For instance, 9 = 10 − 1, so 5 × 9 = 5 × (10 − 1) = 50 − 5 = 45. Since multiplication by 10 is easy as is subtracting 5, there can be little chance of error. Another thought is that 5 = \( \frac{10}{2} \), so 5 × 9 = 5 × (8 + 1) = 5 × 8 + 5 = 10 × 4 + 5 = 45. These are applications of two rules that I have named X9 and X5 and there are many other helpful rules as shown in Figure 4.2.

![Multiplication Table](image)

**Figure 4.2: Multiplication Table**

The rules are as follow.

**Sym**

We all know that 2 × 3 = 3 × 2 and 5 × 4 = 4 × 5, that is we can swap the order of the operands of the multiplication without changing the result. This rule can be stated algebraically as follow.

\[
x \times y = y \times x
\]

where \( x \) and \( y \) can be replaced by any numbers we like. The immediate effect of this rule is that we do not need to learn the 66 values in the bottom left triangle since they all appear in the upper right hand triangle.

**X1**

The top row of the table is trivial since it corresponds to the one times table. Its entries, such as 1 × 5 = 5, are so obvious they hardly need to be learnt. The algebraic rule is as follows.

\[
1 \times x = x
\]
4.5. A MATHEMATICIAN’S APPROACH

X2
This corresponds to the two times table. It is easy to remember that \(2 \times 2 = 4\). We have 5 fingers on each hand making 10 in all, so \(2 \times 5 = 10\) is not a problem. We can surely remember that \(2 \times 10 = 20\) and there are rules (X9, X11 and X12) to help with multiplication by 9, 11 and 12. So we really only have to learn \(2 \times 3 = 6\), \(2 \times 4 = 8\), \(2 \times 6 = 12\), \(2 \times 7 = 14\) and \(2 \times 8 = 16\). The result of multiplying by two is called an even number and always has a 0, 2, 4, 6 or 8 in the units position, and so is easy to recognise.

X10
Multiplication by ten is easy since it just requires a zero to placed on the end of the number, as is \(10 \times 6 = 60\) or \(10 \times 12 = 120\). We could possibly write this rule as follows.

\[
10 \times x = x0
\]

X11
Multiplication by eleven can be simplified by observing that \(11 = (10 + 1)\), so that, for instance, \(11 \times 6 = (10 + 1) \times 6 = 60 + 6 = 66\). The rule is thus:

\[
11 \times x = 10x + x
\]
Notice that when \(x\) is a single digit, it is duplicated, as in \(11 \times 4 = 44\), but when it is 10, 11 or 12 a simple addition is required, as in \(11 \times 10 = 100 + 10 = 110\), \(11 \times 11 = 110 + 11 = 121\) and \(11 \times 12 = 120 + 12 = 132\). These are easy since no carries are required.

X9
Multiplication by nine can be simplified by observing that \(9 = (10 - 1)\), so that, for instance, \(9 \times 6 = (10 - 1) \times 6 = 60 - 6 = 54\). The rule is thus:

\[
9 \times x = 10x - x
\]

X12
Multiplication by twelve can be simplified by observing that \(12 = (10 + 2)\), so that, for instance, \(12 \times 6 = (10 + 2) \times 6 = 60 + 12 = 72\). The rule is thus:

\[
12 \times x = 10x + 2x
\]
Multiplying \(x\) by ten and two are trivial and adding the two results is easy because the units digit will be the units digit of \(2x\) and the senior two digits will be the result of adding 0, 1 or 2 into the ten position of \(10x\), as in \(12 \times 7 = 70 + 14 = 84\) or \(12 \times 9 = 90 + 18 = 108\).

X5
Computing \(5 \times x\) can be simplified by observing that \(5 = \frac{10}{2}\). The rule has two versions depending on whether \(x\) is even or odd.

If \(x\) is even it can be written as \(2n\) and the rule is

\[
5 \times x = \frac{10}{2} \times 2n = 10 \times n
\]
For example, $5 \times 8 = 10 \times 4 = 40$

If $x$ is odd it can be written as $2n + 1$ and the rule is

$$5 \times x = 5 \times (2n + 1) = 10 \times n + 5$$

For example, $5 \times 7 = 5 \times 6 + 5 = 30 + 5 = 35$

**Sq**

Perfect squares are important and should be learnt. All except $3^2$, $4^2$, $6^2$, $7^2$ and $8^2$ have been covered by rules given above. $3^2 = 9$ is easy to remember since it is just three groups of three as in 123 456 789. $4 \times 4 = 2 \times 8$ which equals 16 from the two times table. Observing that $6 = (5 + 1)$ suggests the $6 \times 6 = (5 + 1) \times 6 = 5 \times 6 + 6 = 30 + 6 = 36$. $7 \times 7$ is a problem. Perhaps we should just remember that is is 49, or observe that $7 \times 7 = 6 \times 7 + 7 = 42 + 7 = 49$. Finally $8 \times 8 = 2 \times 4 \times 8 = 2 \times 32 = 64$. Since 8 is $2^3$, $8^2 = 2^6$ and so is a power of two. Powers of two (1, 2, 4, 8, 16, 32, 64, 128, 256, ... ) are important to computer scientists since computers use the binary system. These powers are etched into most computer scientist’s brains, as are $2^{10} = 1024$, $2^{12} = 4096$, $2^{20}$ is about a million and $2^{30}$ is about a thousand million.

**S1**

If you stare at the multiplication table long enough you will notice that

$$4 \times 6 = 24 = 5^2 - 1$$
$$5 \times 7 = 35 = 6^2 - 1$$
$$6 \times 8 = 48 = 7^2 - 1$$
$$7 \times 9 = 63 = 8^2 - 1$$

and so on. This is no accident because it follows from

$$(x - 1) \times (x + 1) = (x - 1) \times x + (x - 1) = x^2 - x + x - 1 = x^2 - 1$$

ie

$$(x - 1) \times (x + 1) = x^2 - 1$$

So the product of two numbers that differ by two is one less that the square of the number between them.

**S4**

The **S1** rule can easily be generalised to

$$(x - y) \times (x + y) = x^2 - y^2$$

If we set $y = 2$ this becomes

$$(x - 2) \times (x + 2) = x^2 - 4$$

as in

$$3 \times 7 = 5^2 - 4 = 25 - 4 = 21$$
$$4 \times 8 = 6^2 - 4 = 36 - 4 = 32$$
This rule is not particularly useful but it does lead to one observation. The larger the value of $y$ the smaller the product. So if you knew that $7 \times 8$ and $6 \times 9$ were 56 and 54, or possibly the other way round. Since $6 \times 9$ must be smaller than $7 \times 8$, $6 \times 9$ must have the smaller value, namely 54.

4.6 Numbers

The programs we have looked at so far involved numbers that were held in variables or named pigeon holes. This section explores how such numbers are represented within the computer.

Humans have always used numbering systems based on 10, presumably because we have 10 fingers. Even in the roman numbering system, 10 is special. For instance, single letters are used for 10 (X), 100 (C) and 1000 (M). Although the Roman numbering system is rather elegant and often used on clock faces (I, II, III, IV, V, VI, VII, VIII, IX, X, XI and XII) it is not convenient for numerical calculation. Consider, for example, adding 16 to 57. In roman numerals we would have to add XVI to DVII giving DXXIII (or 73). In China, India and the Arab world the advantages of multiple digits to represent numbers were well known 3000 years ago but not used in the west until much later. They also discovered the need for the digit zero which had previously not existed. Arithmetic calculations were sometimes done using pebbles placed in holes in the ground and the symbol 0 used to represent zero is thought to be a picture of a hole containing no pebbles.

Fibonacci was one of the first mathematicians in the west to study the advantages of the system we now use. We all know how to add 16 to 57. We first add 6 to 7 giving the answer 3 in the units position and carry of 1 to the tens position. We then add this carry to 1 and 5 giving 7, resulting in the answer 73. Humans are happy with the idea of 10 digits (0 to 9) but computers are much easier to design if only two digits (0 and 1) are available. Typically, in electronic circuits, 0 is represented by a low voltage possibly about 0 volts, and one is represented by a higher voltage of possibly about 3 volts. Numbers using only the digits zero and one are binary numbers. They are like decimal numbers but their digit positions correspond to powers of 2 (1, 2, 4, 8, 16,...) rather powers of 10 (1, 10, 100, 1000,...) used in the decimal system. Using three digit binary numbers, we can count from 0 to 7 as follows: 000, 001, 010, 011, 100, 101, 110, 111. In BCPL, on the Raspberry Pi, numbers are represented using 32 binary digits (or bits) rather than the three just shown. So rather than just eight different numbers, a BCPL variable can have huge number of different values (actually rather more the 4000 million of them). This sounds like a lot and usually causes no problems. But if you write a program that requires numbers outside this range, unexpected things happen. For instance, if we modify the Fibonacci program
above to output Fibonacci numbers up to position 50 and modify the `writef` statements to be:

```c
writef("Position %2i  Value %12u  %32b*n", i, a, a);
```

The `%12u` substitution item outputs the Fibonacci number as an unsigned (i.e., \( \geq 0 \)) number in a field width of 12 characters and `%32b` outputs it as a 32-bit binary number. The resulting output is:

<table>
<thead>
<tr>
<th>Position</th>
<th>Value</th>
<th>Binary Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>00000000000000000000000000000000</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>00000000000000000000000000000001</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>00000000000000000000000000000001</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>00000000000000000000000000000010</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>00000000000000000000000000000011</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>00000000000000000000000000000101</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td>00000000000000000000000000001000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>45</td>
<td>1134903170</td>
<td>0100011110100100111111110000010</td>
</tr>
<tr>
<td>46</td>
<td>1836311903</td>
<td>011010101111001110010101011111</td>
</tr>
<tr>
<td>47</td>
<td>2971215073</td>
<td>10110001110001001001001001100010</td>
</tr>
<tr>
<td>48</td>
<td>5125595680</td>
<td>00111110100001101100001010100000</td>
</tr>
<tr>
<td>49</td>
<td>3483774753</td>
<td>110011110100110011110100100001</td>
</tr>
<tr>
<td>50</td>
<td>3996334433</td>
<td>1110111000110011100101100000001</td>
</tr>
</tbody>
</table>

Notice that the value at position 6 is 8 which is the sum of 3 and 5. In binary, the calculation is `0011+0101` giving `1000`. The value at position 47 is correct, but after that the Fibonacci numbers are too large to be represented with just 32 bits, and digits off the left hand end are lost. This unfortunate effect is called overflow and some languages generate a warning when this happens, but not BCPL. BCPL assumes that programmers are really clever and careful and don’t need such warnings which, in any case, greatly complicates the definition of the language.

We have seen that decimal constants such as 2 and 100 can be written in the normal way, but BCPL also allows binary constants by prefixing a string of binary digits with `#b`, as in `#b0011` and `#b0101`. It is sometimes helpful to put underscores in long numbers to make them more readable. For instance, the binary representation of the Fibonacci number at position 47 could be written as:

```
#b1011_0001_0001_1001_0010_0100_1110_0001
```

This can also be written as a more concisely using the hexadecimal digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E and F, as follows:

```
#xB11924E1
```
4.6. NUMBERS

Each hexadecimal digit represents 4 binary digits, so, for instance, #xB means #b1011 and #xB1 means #b10110001, etc.

In binary numbers the values associated with the digits, taken from the right (or least significant end) are 1, 2, 4, 8, 16, ... or $2^0, 2^1, 2^2, 2^3, 2^4$, ... Following this convention the left most bit of a 32-bit binary number corresponds to the value $2^{31}$ which is, of course, a positive number. Unsigned numbers use this convention, but if we want to represent positive and negative numbers, the normal convention to use is to assign a value of $-2^{31}$ to the left most bit. This allows us to have numbers roughly in the range -2000 million to +2000 million. Notice that #x80000000 represents the largest negative number, #xFFFFFFFE represents the number -1 and #x7FFFFFFF represents the largest positive number.

The representation of -1 perhaps needs some explanation. With a decimal numbers such as 9999, we all know how to increment it by one. During the calculation there is a cascade of carries before producing the answer 10000. So a string of consecutive nines on the right are converted to zeroes. A similar cascading effect happens when we increment a binary number having a sequence of ones on the right. Just as nine is the largest decimal digit, one is the largest binary digit, so when incrementing the digit one it turns into a zero and generates a carry. If we add one to the binary number 1111, there is a cascade of carries before giving the result 10000. If we add one to the binary number consisting of a zero bit followed by 31 ones (#x7FFFFFFF) we get a one followed by 31 zeroes (#x80000000). In unsigned arithmetic this correctly represents the value $2^{31}$.

In signed arithmetic, this result represents $-2^{31}$ and so the calculation has overflowed, so #x7FFFFFFF must be the largest positive number than can be represented. If we increment a bit pattern of 32 ones (#x7FFFFFFF), using signed arithmetic, all the least significant ones are turn to zeroes and the left most bit also changes from a one to a zero. This gives the correct answer since the carry into the left most bit represents $2^{31}$ and this cancels the one that is there representing $-2^{31}$ correctly giving a zero bit in this position. Thus adding one to #x7FFFFFFF gives zero, and so #xFFFFFFFE must represent $-1$.

We have already seen the operators +, - and MOD used in programs given above, but several other expression operators available. The operator * will multiply its operands together as in 3*7 gives 21. The operator / divides its left hand operand by the one on the right, as in 13/5 gives 2. Notice that the result is a whole number and the remainder, if any, is thrown away. The remainder after division can be obtained using the MOD operator, as in 13 MOD 5 which gives 3. If we do ordinary arithmetic using operators like +, - and * but always return the remainder after division by some number, often called the modulus, then we are doing what is called modulo arithmetic. We will see useful applications of modulo arithmetic later.

A value can be negated using - as a monadic operator, as in -$x$. If $x$ was 1000 then the result would be -1000. The monadic operator ABS negates its operand if it was negative, but leaves it unchanged if it was positive. Thus, ABS (-1000)
and ABS 1000 both give 1000.

There are various operators that manipulate bit patterns directly. For instance, $x \ll n$ will shift the value of $x$ left by the number of bits specified by $n$. Bits are lost off the left hand end and vacated positions on the right are filled with zeroes. The expression $x \gg n$ similarly computes $x$ shifted right by $n$ bit positions, filling vacated positions with zeroes. The operators & and | perform the logical bit-wise operations of and and or. For and, the $n^{th}$ bit of the result is only a one if the $n^{th}$ bit of both operands are ones, as in $\texttt{#b0011 & #b1010}$ gives $\texttt{#b0010}$. For or, the $n^{th}$ bit of the result is only a zero if the $n^{th}$ bit of both operands are zeros, as in $\texttt{#b0011 | #b1010}$ gives $\texttt{#b1110}$. The monadic operator ~ complements each bit of its operand to give the result. You might like to convince yourself that $(\neg x)+1 = -x$. The XOR operator computes a result in which the $n^{th}$ bit is only a one if the corresponding bits of its two operands are different, as in $\texttt{#b0011 XOR #b1010}$ gives $\texttt{#b1001}$.

Two little tricks are worth noting. If we subtract one from a variable $x$ we get a bit pattern identical to $x$ except the consecutive zero bits on the right have all changed to ones, and the rightmost occurring one has changed to a zero. If we then and this with the original value of $x$ we obtain a bit pattern with the right most occurring one removed. For example:

\[
\begin{array}{c|c}
  x & 0101_1101_0011_1010_0000_0110_0000_0000 \\
  x-1 & 0101_1101_0011_1010_0000_0101_1111_1111 \\
  x \& (x-1) & 0101_1101_0011_1010_0000_0100_0000_0000 \\
\end{array}
\]

Similarly, if we compute $x \& (-x)$, we obtain a bit pattern which is all zeroes except for a one in the position of the right most one in $x$. For example:

\[
\begin{array}{c|c}
  x & 0101_1101_0011_1010_0000_0110_0000_0000 \\
  -x & 1010_0010_1100_0101_1111_1010_0000_0000 \\
  x \& (-x) & 0000_0000_0000_0000_0000_0010_0000_0000 \\
\end{array}
\]

Many other bit manipulations require cunning to do them efficiently. For instance, how can we find the most significant occurring one, or count the number of ones in a bit pattern. If you are interested in these kinds of problems look at the programs in bcplprogs/bits.

### 4.7 Applications of XOR and MOD

If you do not feel up it skip this section and the next, but, trust me, you might find it interesting.

Cryptography is the science of encoding secret messages is a way which allows only the intended recipient to decode them. Many methods involve
the use of a shared secret key known by both the sender and receiver but unknown to everyone else. Suppose the sender and receiver agree that the shared secret key is the 32 bit word \#x87654321 and the message to be sent is \#x0ABCDEF0. The sender could encode the message using the XOR operator to combine the key with the message to give the encrypted message \#x8DD99DD1 (= \#x87654321 XOR \#x0ABCDEF0). This has complemented some of the bits in the binary representation of the message, and the receiver can complement the same bits by computing \#x87654321 XOR \#x8DD99DD1, giving back the original message \#x0ABCDEF0. To anyone not knowing the secret key, the encoded message \#x8DD99DD1 is meaningless. This is potentially the basis of an excellent encryption technique but it suffers the major problem of how we setup the secret keys between everyone who wishes to encrypt their messages. You cannot send a key unencrypted since an eavesdropper will be able to see it, and you cannot send it encrypted because we have assumed you have no secret key already set up. You could possibly hand it over in person, by telephone or by post, but these methods take time a may be inconvenient. A better solution must be found.

It was not until 1978 that a suitable mechanism, called RSA public-key encryption, was invented (named after the developers Rivest, Shamir and Adleman). The idea is simple. The receiver publishes a key that everyone can read. The sender uses this key to encode the message and sends it to the receiver. The way the message is encoded is such that it cannot be decoded using the public key but requires an additional secret known only by the receiver, the person that published the public key. The public key consists of two carefully chosen random numbers \( r \) and \( e \). To encode a message \( M \), assumed to be less than \( r \), we compute \( M^e \) (ie 1 multiplied by \( M \), \( e \) times) and then take the remainder after division by \( r \). If we call this encrypted value \( C \), then

\[
C = M^e \mod r
\]

Although this calculation looks horrendous, it is, in fact, quite easy to do, as shown in page 65. Knowing the public key is not enough to decode the encrypted message. However, there is a decoding exponent \( d \) that was calculated and kept secretly by the receiver when the public key of \( r \) and \( e \) was chosen. This can be used to decode the encrypted message \( M \) by evaluating the following:

\[
C^d \mod r
\]

As an example, if the receiver chose a public key of \( r=1576280161 \) and \( e=10000691 \), and a decoding exponent of \( d=899015831 \), the calculations would be as follows.

\[
\#x0ABCDEF0^{10000691} \mod 1576280161 \text{ gives } \#x5AF3EBFE
\]

and

\[
\#x5AF3EBFE^{899015831} \mod 1576280161 \text{ gives } \#x0ABCDEF0
\]
This gives the correct result, and since only the receiver knows the decoding exponent, no one else can (easily) decode the message.

To see how the above calculations were done, look at the file bcplprogs/crypt/rsa.b. The next section (which may be skipped) gives a brief introduction to the underlying mathematics associated with RSA encryption.

4.7.1 RSA Mathematical Details

This section is entirely optional and should only be read by those who are interested. It shows how the public key and decoding exponent can be chosen, but does not go into the details of why the mechanism works. In practice, the public key should be rather large, perhaps 2000 bits in length or more. So all arithmetic must be done using numbers of this size rather than the 32 bits used in the previous section.

To create a new public key, first think up two large prime numbers \( p \) and \( q \) that are roughly equal and whose product is about 2000 bits long. Unfortunately finding such large primes is out of the scope of this document. Now multiply \( p \) by \( q \) to give the first component of the public key. Next choose a number \( e \) that is about the same size as \( p \), and check that it has no factors in common with \( (p-1)*(q-1) \). This is extremely likely to be true if \( e \) is a prime. If the test succeeds \( e \) is the second component of the public key, otherwise keep trying other values for \( e \). Now find the decoding exponent by finding \( d \) such that

\[
(e * d) \equiv 1 \pmod{(p-1)*(q-1)}
\]

This amounts to calculating \( d = 1/e \) using arithmetic modulo \( (p-1)*(q-1) \). This can be done using a program related to Euclid’s greatest common divisor (GCD) algorithm.

The public key used in the previous section was based on the prime numbers \( p=45007 \) and \( q=35023 \). Their product was 1576280161 and the chosen encoding exponent was 10000691. The expression \( (p-1)*(q-1) \) evaluates to 1226540484, and \((1/e) \mod 1226540484\) gives 899015831, the decoding exponent.

Notice that if you can factorise the first component of the public key into its two prime factors \( p \) and \( q \), you would be able to calculate the decoding exponent \( d \) and so would be able to decode any message using this public key. Luckily factorising such large numbers is thought by most mathematicians to be unfeasible.

This is only the germ of the idea of public key encryption. For a professional version much attention must be paid to subtle details of the implementation and use.
4.8 Vectors

We have already seen that variables are like named pigeon holes that contain numbers, and that they can be declared by declarations such as

\[
\text{LET } x, y, z = 5, 36, 1004
\]

To implement this declaration, BCPL finds three pigeon holes that are currently free, labels them with the names \(x, y\) and \(z\), and puts the numbers 5, 36, 1004 into them. The BCPL Cintcode system normally has about 4 million pigeon holes to choose from, and each is labelled with an identifying number, similar to the way houses have numbers. Such numbers help postmen deliver letters, and pigeon hole numbers turn out to be fantastically useful in BCPL programs. The pigeon hole numbers of variables \(x, y\) and \(z\) can be found using the \@ operator, as in the following program.

```
GET "libhdr"

LET start() = VALOF
{ LET x, y, z = 5, 36, 1004
  writef("@x=%n @y=%n @z=%n+n", @x, @y, @z)
  RESULTIS 0
}
```

The following shows this program being compiled and run.

```
0.000> c b vec1
bcpl vec1.b to vec1 hdrs BCPLHDRS

BCPL (1 Feb 2011)
Code size = 80 bytes
0.030>
0.000> vec1
@x=12156 @y=12157 @z=12158
0.000>
```

Notice that the pigeon hole numbers for variables \(x, y\) and \(z\) are consecutive. This is no accident since BCPL always allocates consecutive pigeon holes to variables declared by simultaneous declarations. Pigeon hole numbers are normally called addresses and the symbol \@ was chosen because it looks like an \(a\) inside an \(o\) standing for address of.
Instead of using the name \( x \) to access the contents of its pigeon hole we can use the indirection operator \( ! \) applied to the pigeon hole number. So if \( \&x \) evaluates to 12156, then \( !12156 \) would behave exactly like \( x \).

We cannot tell in advance what the address of \( x \) will be, so it would be better to declare another variable \( p \), say, to hold this value. The expressions \( !p \), \( !(p+1) \) and \( !(p+2) \) are now equivalent to \( x \), \( y \) and \( z \). Since expressions like \( !(p+1) \) and \( !(p+2) \) are so useful, a dyadic version of the \( ! \) operator is provided allowing these expressions to be written as \( p!1 \) and \( p!2 \), as is shown in the following example.

```
GET "libhdr"

LET start() = VALOF
{ LET x, y, z = 5, 36, 1004
  LET p = \&x
  p!2 := p!0 + p!1 // Equivalent to \( z := x + y \)
  writef("x=%n y=%n z=%n\n", x, y, z)
  RESULTIS 0
}
```

The output from this program is as follows.

\[ x=5 \quad y=36 \quad z=41 \]

Collections of consecutive pigeon holes are called vectors in BCPL. In other languages, they are often called one dimensional arrays. They are sometimes used to represent values that are too large to fit into a single BCPL word. An example is BCPL’s representation of the current time and date as shown in the following program (vec3.b).

```
GET "libhdr"

LET start() = VALOF
{ LET days, msecs, filler = 0, 0, 0
datstamp(@days)
  writef("days=%n msecs=%n filler=%n\n", days, msecs, filler)
  // Output the time in hh:mm:ss.mmm format
  writef("The time is %2i:%2z:%2z.%3z\n",
         msecs/(60*60*1000), // The hours
         msecs/(60*1000) MOD 60, // The minutes
         msecs/1000 MOD 60, // The seconds
         msecs MOD 1000) // The milli-seconds
  RESULTIS 0
}
```
4.8. VECTORS

We can run this program vec3 immediately followed by the command dat msecs separating by a semicolon (;) giving the following output.

```
0.010> vec3; dat msecs
days=15502 msecs=38273016 filler=-1
The time is 10:37:53.016
    Monday 11-Jun-2012 10:37:53.020
0.000>
```

The argument given to the library function datstamp is the address of the first of three consecutive variables named days, msecs and filler to hold a representation to the current time and date. After the call, days holds 15502 being the number of days since 1 January 1970, and msecs holds 38273016 being the number of milli-seconds since midnight. To demonstrate this number is correct, it has been converted to hours, minutes and seconds and compared with the output of the dat command. By the way, dat stands for date and time.

Historically, datstamp was defined when BCPL was typically used on 16-bit computers such as the PDP-11, Data General Nova or the Computer Automation LSI-4. When BCPL words were only 16 bits long three words were need to represent the date and time. For compatibility with the past three words have been retained with the convention that -1 in filler indicates that the new representation is being used.

It is all very well declaring vectors using simultaneous declarations, but this method is not feasible if we wish to declare a vector containing 1000 elements, or if we do not know how many elements we need until the program is running. The declaration LET v = VEC 10 declares a variable v initialised with the address of 11 consecutive pigeon holes. They can be accessed by expressions such as v!0, v!1 up to v!10. The operand of VEC, in this case 10, is the upperbound of the vector and must be a compile time constant. The elements of v are unnamed and so can only be accessed using the subscription operator (!). Vectors declared using = VEC are allocated from and area of memory called the run time stack which is of limited size (typically 50000 words), so if you require vectors larger than about 1000 elements, or if you do not know how large they should be until the program is running, you should allocate them using getvec. This function has one argument which is the upperbound of the vector required and it returns the address of its zeroth element, or zero if insufficient space is available.

Vectors allocated by getvec should be freed by calls of freevec otherwise space will be permanently lost. This is often called a space leak as illustrated by the following program (vec4.b).

GET "libhdr"
LET start() = VALOF
\{ LET v1, v2 = 0, 0
  v1 := getvec(100_000)
  writef("getvec(100_000) => %n", v1)
  v2 := getvec(3_000_000)
  writef("getvec(3_000_000) => %n", v2)
  IF v1 DO freevec(v1)
    //IF v2 DO freevec(v2) // Forget to free v2
  RESULTIS 0
\}

The effect of running this is as follows.

0.030> vec4
  getvec(100_000) => 62171
  getvec(3_000_000) => 162181
0.010>

The state of memory can be inspected using the command map pic, as follows:

0.010> map pic
Largest contiguous free area: 837810 words
Totals: 4000000 words available, 3012122 used, 987878 free
This shows that the 3 million words allocated for v2 have not been freed, so the next time vec4 is executed it is unable to allocate v2.

0.000> vec4
getvec(100_000) => 62171
getvec(3_000_000) => 0
0.010>

An advantage of declaring a vector using = VEC is that it is automatically freed when execution leaves the block in which it was declared.

On page 38 we saw how to write out some Fibonacci numbers. We will now look at a program fills a vector with them.

GET "libhdr"

LET start() = VALOF
{ LET f = VEC 50 // A vector to hold Fibonacci numbers from 0 to 50
  f!0 := 0 // Fill in the first two Fibonacci number
  f!1 := 1
  // Now fill in the others
  FOR i = 2 TO 50 DO f!i := f!(i-1) + f!(i-2)

  // Now write out the result
  FOR i = 0 TO 50 DO
    writef("Position %2i Value %12u %32b*n", i, f!i, f!i)

  RESULTIS 0
}

It produces exactly the same output that we saw on page 48.

4.9 Primes

As another example of the use of vectors, we will look a program that finds all prime numbers less than a million. The program is as follows.
GET "libhdr"

LET start() = VALOF
{ LET upb = 1_000_000
  LET isprime = getvec(upb)

  FOR i = 2 TO upb DO isprime!i := TRUE // Until proved otherwise.

  FOR p = 2 TO upb IF isprime!p DO
    { LET i = p*p // First non prime to be crossed out
      // Cross out all multiples of p
      IF i>upb BREAK
      { isprime!i := FALSE; i := i + p } REPEAT UNTIL i>upb
    }

  // Output some primes near the end
  FOR p = upb-100 TO upb IF isprime!p DO writef("%6i*n", p)

  freevec(isprime)
  RESULTIS 0
}

This program outputs the primes between 999900 and a million.

0.000> vec6
999907
999917
999931
999953
999959
999961
999979
999983
0.200>

4.10 MANIFEST, GLOBAL and STATIC declarations

We have already seen how to declare local variables and vectors using LET, but there other ways to declare variables. The first of these is the MANIFEST declaration as in:
4.10. MANIFEST, GLOBAL AND STATIC DECLARATIONS

MANIFEST {
    col_red = #xFF0000
    col_green = #x00FF00
    col_blue = #x0000FF

    n_op=0 // The operator field of a node
    n_r1 // The first operand field of a node
    n_r2 // The second operand field of a node

    // List of node operators
    s_num=1 // A number node
    s_mul // A multiply node
    s_div // A divide node
    s_add // An add node
    s_sub // A subtract node
}

This declaration declares various named constants such as col_red and n_op. If
the name being declared is followed by an equal sign (=) then its value is that of
the constant following the equals, otherwise its value is one larger than that of
the previous name declared. Thus n_r1 and n_r2 have values 1 and 2.

The GLOBAL vector is a area of memory that is allocated when a program starts
and usually has an upperbound of 1000. It is possible to give names to particular
elements of the global vector and this is done using a GLOBAL declaration. The
following example is a modification of part of the standard library header file
g/libhdr.h.

GLOBAL {
    globsize: 0
    start: 1
    stop: 2
    sys: 3 //SYSLIB MR 18/7/01
    clihook: 4
    muldiv: 5 //SYSLIB changed to G:5 MR 6/5/05
    changeco: 6 //SYSLIB MR 6/5/04
    currco: 7
    colist: 8
    rootnode: 9 // For compatibility with native BCPL
    result2
    returncode
    cis
It declares that `globsize` is a variable at position zero of the global vector. By convention it holds the upper bound of the global vector which is usually 1000. This can be confirmed by executing
\[
\text{writef("globsize=%n*n", globsize)}
\]
The next variable is called `start` and is by convention is the first function of a program to be called.

The variables `result2`, `returncode`, `cis` and `cos` are not followed by colons (`:`) and so are given successively the next available global positions, namely 10, 11, 12 and 13.

The main advantage of global variables is that they provide a means of communication between separately compiled parts of the system. For instance, there is a precompiled library module called `blib` that contains the definitions of functions like `writef` that we have used in all the example programs so far. The entry point to `writef` actually resides in global 94 and is initialise at the moment a program starts.

\textbf{STATIC} declarations have a similar syntax to \textbf{MANIFEST} declarations but declare initialised variables rather than constants. Unlike manifest constants they can be updated using assignment statements. An example is as follows:

\[
\text{STATIC} \{ \\
\quad a=1 \\
\quad b \\
\quad c \\
\}
\]

This will declare three static variables `a`, `b` and `c` initialised to 1, 2 and 3. In general static variables should not be used unless absolutely necessary. They are usually better placed in the global vector.

4.11 Functions

We have already used functions several times. For instance, we have defined the function `start` in every program and we have used functions such as `writef`, `datstamp`, `getvec` and `freevec` several times. In this section we examine functions in more detail.

Sometimes we have a fragment of code that we would like to use in several different places. It would therefore be good to have a simple way on executing that code without having to write the entire fragment on each time. In most programming languages this can be done by wrapping up the code in something called a function. As an example we will look at the definition of the library
4.11. FUNCTIONS

function randno which generates a sequence of pseudo random numbers. Its definition is as follows.

LET randno(upb) = VALOF
{ // Return a random number in the range 1 to upb
    randseed := randseed*2147001325 + 715136305
    RESULTIS (ABS(randseed/3)) MOD upb + 1
}

This declares the function randno whose entry point is held in global variable 34 as declared in libhdr.h. Within its body it refers to randseed which is declared as global 35. The function is an implementation of what is called a congruential random number generator with carefully chosen constants 2147001325 and 715136305 to cause it to cycle though a huge number of apparently random values. The use of ABS, division by 3, MOD and +1 remove some of the deficiencies of the randseed sequence and restrict the resulting numbers to the required range of 1 to upb. Each value in this range should occur with equal likelihood.

There are two things to note about function definitions. Firstly, if the name of the function is already declared as a global then its entry point becomes the initial value of that global. Secondly, every variable used inside a function must either be declared inside that function or be declared by a function, MANIFEST, GLOBAL or STATIC declaration. Thus so called dynamic free variables are not allowed. To avoid this problem, never define a function inside another. (This is enforced syntactically in languages like C).

You can pass a collection of values to a function when you call it. These are called arguments and they are enclosed in round brackets (‘(‘ and ‘)’). We have already seen this done in calls like writef("x=%n y=%n z=%n*n", x, y, z). Here we are calling the function writef giving it four arguments. The first is a string (actually represented by a pointer to the characters of the string), and the remaining ones are the values of x, y and z. When a function is declared it is given a list of names enclosed in round brackets and separated by commas. These names behave just like local variables that have been initialised from left to right with the argument values. The declaration of writef is in the file sysb/blib.b and its first line is:

LET writef(format,a,b,c,d,e,f,g,h,i,j,k,l,m,
n,o,p,q,r,s,t,u,v,w,x,y,z) BE

As can be seen, its first argument is called format to hold the format string given in the call. The remaining 26 arguments are initialised to as many arguments as were supplied in the call. Hopefully no one will call writef with more than this number of arguments. If they do the later arguments will be lost. Just
as simultaneously declared local variables live in adjacent pigeon holes, the same
applies to function arguments. So, for instance, the arguments a to z can thought
of as a vector of 26 elements pointed to by @a, and so can be accessed conveniently
as needed within the declaration of writef. Functions taking variable numbers
of arguments are often called variadic functions. They are clearly useful but often
difficult to implement sensibly in other languages.

The word BE in the declaration of writef indicates that its result is undefined
and that its body is not an expression but a command or command sequence.
After all, writef is not designed to compute a value since its purpose is to output
some formatted text.

Functions designed to compute results are declared using = in place of BE, and
after the equal sign there is an expression (not a command). A simple example
is the definition of the factorial function that computes $1 \times 2 \times 3 \ldots \times n$
for a given argument $n$. Its definition is as follows:

\[
\text{LET } \text{fact}(n) = n=0 \rightarrow 1, \ n*\text{fact}(n-1)
\]

The expression $n=0 \rightarrow 1, n*\text{fact}(n-1)$ is an IF-THEN-ELSE construct for
expressions. It computes the condition, in this case $n=0$, and if the result is non
zero (representing TRUE) it returns the first alternative namely 1, otherwise it
returns the result of evaluating $n*\text{fact}(n-1)$. The interesting thing about this
definition is that it is recursive, defining fact in terms of itself, based on the idea
that factorial 0 is 1 and for non zero $n$ factorial of $n$ is $n \times$ factorial of $n-1$.

Another example is a rather beautiful definition of a function to compute
Fibonacci numbers. The following program outputs them up to position 50.

\[
\text{GET "libhdr"}
\]

\[
\begin{align*}
\text{LET } & \text{fib}(n) = n=0 \rightarrow 0, \\
& \quad n=1 \rightarrow 1, \\
& \quad \text{fib}(n-1) + \text{fib}(n-2)
\end{align*}
\]

\[
\text{LET start() = VALOF}
\]

\[
\{ \text{FOR } i = 0 \text{ TO 50 } \text{DO} \\
\quad \text{writef("Position }\%2i\text{ Value }\%12u*n\", i, \text{fib}(i)) \}
\]

\[
\text{RESULTIS 0}
\]

When you run this program it takes longer and longer to output each line, and if
you time it with a stopwatch, each line take a time approximately proportional
to the value of the Fibonacci number it is printing. On my laptop it takes about
4.12. SOLVING THE RECURRENCCE RELATION FOR $C$

2 hours to output all 51 Fibonacci numbers and, although I have not tried, I would expect it to take about 8 times longer on the Raspberry Pi. It is perhaps interesting to explore why this wonderfully elegant little program is so inefficient.

Let us try and define a cost function $C(n)$ that is the cost (in time) of computing $\text{fib}(n)$. When $n$ is 0 or 1 computing $\text{fib}(n)$ is very cheap. Let us arbitrarily say the cost of computing $\text{fib}(0)$ is so small it can be zero and the cost of computing $\text{fib}(1)$ is one unit. For larger values of $n$ the cost is dominated by the cost of computing $\text{fib}(n-1)$ and $\text{fib}(n-2)$ giving a total of $C(n-1) + C(n-2)$.

So we have defined the cost function $C$ to have the following properties.

\[
\begin{align*}
C(0) &= 0 \\
C(1) &= 1 \\
C(n) &= C(n-1) + C(n-2) \quad \text{when } n > 1
\end{align*}
\]

This recurrence relation gives us exactly the same sequence of values as the Fibonacci sequence itself which explains why the time to output each line is approximately proportional to the Fibonacci number being written. In the next section (which is entirely optional) we will obtain a simple formula for $C$ (and indeed $\text{fib}(n)$).

### 4.12 Solving the recurrence relation for $C$

In this section we explore the peculiar way in which mathematicians think. They are typically extremely optimistic, thinking they can solve apparently unsolvable problems. They are persistent, repeatedly trying different approaches when all earlier attempts have failed, and they have usually acquired reasonable skill in algebraic manipulation.

To solve this problem, a mathematician checks whether $C(n)$ grows as fast as $n^2$ or $n^3$ but soon discovers that it grows much faster. Indeed it looks as if it grows faster than $n^k$ for any $k$. Oh dear, we must find a formula that grows faster than any of these. How about $X^n$? So lets try $C(n) = X^n$. This clearly is not right, but lets try it all the same. When $n$ is large, substituting this in our definition of $C(n)$ gives us $X^n = X^{n-1} + X^{n-2}$. Assuming $X$ is not zero we can divide both sides of the equation by $X$ giving $X^{n-1} = X^{n-2} + X^{n-3}$ and if we repeatedly divide by $X$ we eventually get the beautifully simple equation $X^2 = X + 1$. If we rearrange this to be $X^2 - X = 1$ and then add 1/4 to both sides we get $X^2 - X + 1/4 = 1 + 1/4 = 5/4$. We can now take the square root of both sides giving $X - 1/2 = \sqrt{5}/2$. So possible values of $X$ are $(1 + \sqrt{5})/2$ and $(1 - \sqrt{5})/2$. The first of the has a value of about 1.618 and is so famous it is called the Golden Ratio. Look it up on the Web to see why it is so important. The second value is approximately -0.618. If we call these two values $\alpha$ and $\beta$, we can convince ourselves that a mixture of the two such as $A\alpha^n + B\beta^n$ also satisfies
the relation, and by choosing suitable values for $A$ and $B$, we can make a simple formula match $C(n)$ exactly. Substituting $n$ equals 0 and 1 in our definition of $C(n)$ we get $C(0) = A\alpha^0 + B\beta^0 = A + B = 0$ and $C(1) = A\alpha + B\beta = 1$. The first equation tells us that $B = -A$, and substituting this in the second equation gives $A(\alpha - \beta) = 1$. Remembering that $\alpha = (1 + \sqrt{5})/2$ and $\beta = (1 - \sqrt{5})/2$ we can easily deduce that $A = 1/\sqrt{5}$. The formula for $C(n)$ is thus

$$C(n) = (\alpha^n - \beta^n)/\sqrt{5}.$$  

or

$$C(n) = \frac{(1+\sqrt{5})^n - (1-\sqrt{5})^n}{2^n\sqrt{5}}.$$  

As a challenge, convince yourself that this yields a whole number for every $n$ even though this formula contains $\sqrt{5}$ three times.

### 4.13 Greatest Common Divisor

The greatest common divisor (the GCD) of two positive numbers is the largest number that exactly divides into both of them. For instance the GCD of 18 and 30 is 6. In roughly 200 BC, Euclid divided an efficient way of computing it. It is essentially as follows. If they are equal that is the answer, otherwise replace the larger number by the remainder of dividing it by the smaller number, repeating the process until both numbers are equal. A BCPL implementation of this is as follows:

GET "libhdr"

LET gcd(a, b) = VALOF
{ LET r = a MOD b // r will be less than b
  IF r=0 RESULTIS b // b exactly divides a so is the gcd
  // r and b have the same gcd as a and b
  a := b
  b := r // a is greater than b
} REPEAT

LET try(a, b) BE
{ LET res = gcd(a, b)
  writef("gcd(%n, %n) = %n*n", a, b, res)
}
\[ \text{gcd}(18, 30) = 6 \]
\[ \text{gcd}(1000, 450) = 50 \]
\[ \text{gcd}(1576280161, 1226540484) = 1 \]

Notice that if \( b \) is greater than \( a \) initially, then the first iteration of the \texttt{REPEAT} loop just swaps these variables.

### 4.14 Powers

Another example worth looking at is how to raise a number to a large power using modulo arithmetic. That is how can we calculate \( x^n \) modulo \( m \) efficiently as is required by the RSA mechanism described above.

Two ideas come to mind. One is that when we want to calculate, say, \( 1234 \times 5678 \) modulo 100, we need only consider the two least significant digits of each number, since the others cannot affect the answer. So calculating \( 34 \times 78 \) modulo 100 gives the same result. This generalises to \( a \times b \) modulo \( m \) gives the same result as \(( (a \text{ modulo } m) \times (b \text{ modulo } m) ) \) modulo \( m \). The other idea is to consider the binary representation of the exponent. For instance, if we want to calculate \( 7^{25} \), we observe that 25 is 11001 in binary corresponding to \( 16 + 8 + 1 \) so multiplying 1 by 7, 25 times is the same as multiplying 1 by 7, 16 times, then multiplying by 7, 8 times and finally multiplying by 7 once more. In mathematical notation this is just saying

\[
7^{25} = 7^{16+8+1} = 1 \times 7^{16} \times 7^{8} \times 7.
\]

We can easily calculate \( 7^2, 7^4, 7^8 \) and \( 7^{16} \) since \( 7^2 = 7 \times 7, 7^4 = 7^2 \times 7^2, 7^8 = 7^4 \times 7^4 \), etc. Based on these ideas we can construct an elegant program that compute \( x^n \) modulo \( m \), such as the following.

\[ \text{LET powmod}(x, n, m) = \text{VALOF} \]
\[ \{ \text{LET res = 1} \]
\[ \quad \text{LET p = x \text{ MOD } m} \]
\[ \quad \text{WHILE n DO} \]
\[ \quad \{ \text{IF (n & 1)=0 DO res := (res * p) \text{ MOD } m} \]
\[ \quad \quad \text{n := n\text{>>}1} \]
This program has two disadvantages. One is that it is using signed arithmetic and secondly it has a problem with overflow and so only works with quite small numbers. A version using full 32-bit unsigned numbers is as follows.

GET "libhdr"

LET add(x, y, m) = VALOF
{ LET a = x+y
   IF x<0 & y<0 & a>0 RESULTIS a-m
   IF a-m<0 RESULTIS a // Unsigned comparison
   RESULTIS a-m
}

AND mul(x, y, m) = y=0 -> 0,
   (y&1)=0 -> mul(add(x,x,m), y>>1, m),
   add(x, mul(add(x,x,m), y>>1, m), m)

AND pow(x, y, m) = y=0 -> 1,
   (y&1)=0 -> pow(mul(x,x,m), y>>1, m),
   mul(x, pow(mul(x,x,m), y>>1, m), m)

LET start() = VALOF
{ LET a, n, m = 7, 25, 19
   writef("%n****%n modulo %n = %n*n", a, n, m, pow(a, n, m))
   a, n, m := #x0ABCDEF0, 10000691, 1576280161 // Should give #x5AF3EBFE
   writef("%8x****%n modulo %8x*n", a, n, m, pow(a, n, m))
   RESULTIS 0
}

4.15 Compilation

So far we have looked at a few BCPL programs and invoked the BCPL compiler before running them. In this section we explore what the BCPL compiler actually does and how the compiled code is executed. To illustrate what is going on we will consider the following simple program (in bcplprogs/raspi/demo.b).
```plaintext
GET "libhdr"

LET start() = VALOF
{ LET n = 7
  LET count = 0

  { count := count+1
    IF n=1 RESULTIS count
    TEST n MOD 2 = 0
    THEN n := n/2
    ELSE n := 3*n+1
  } REPEAT
}

This program declares two variables \( n \) and \( count \) initialised to 7 and zero. It then enters a REPEAT loop in which it increments \( count \) before testing to see if \( n \) is one. If it is, it returns from \texttt{start} with the current value of \( count \). By convention, a non zero result is treated as an error causing its value to be output, as in:

0.010> c b demo
bcpl demo.b to demo hdrs BCPLHDRS

BCPL (24 July 2012)
Code size = 68 bytes
0.020> demo
demo failed returncode 17 reason -1
0.010>

This indicates that when it detects that \( n \) equals to 1, \( count \) equals to 17. The TEST statement causes \( n \) to be set to \( n/2 \) if \( n \) was even or \( 3*n+1 \) if \( n \) was odd. These operations are repeated until the program is terminated by the RESULTIS statement. With \( n \) initially set to 7, the sequence of values of \( n \) has length 17 and is as follows:

\[
7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1
\]

Before running \texttt{demo} we have to compile it using a command such as \texttt{c b demo}. The effect of this is to read the file \texttt{demo.b} and output a file called \texttt{demo}. This file can be displayed using the \texttt{type} command as follows:
At first sight this compiled code does not look very comprehensible. It basically consists of a sequence of 32-bit words given in hexadecimal. The first \(000003E8\) indicates that this is a hunk of compile code whose length is given by the next value \(00000011\). The rest of the file gives the actual data that must be loaded into memory before the demo program can be run. This code is much easier to understand if we use the \(d1\) option when invoking the compiler. The output this generates is as follows:

```
0.000> c b demo d1
bcpl demo.b to demo hdrs BCPLHDRS d1
```

```
BCPL (24 July 2012)
 0:  DATAW 0x00000000
  4:  DATAW 0x0000DFDF
   8:  DATAW 0x6174730B
  12:  DATAW 0x20207472
  16:  DATAW 0x20202020
// Entry to:  start
 20:   L1:
   20:    L7
   21:    SP3
   22:    L0
   23:    SP4
   24:   L3:
   24:    L1
   25:    AP4
   26:    SP4
   27:    L1
   28:    LP3
   29:    JNE  L4
   31:    LP4
   32:    RTN
   33:   L4:
   33:    LP3
   34:    L2
   35:    REM
```
4.15. COMPILATION

The word at position zero will hold the length of the compiled code when it is known, and this if followed by four words that indicate that the function named start follows at byte position 20 in this module. The compiler kindly comments this position to make the code more readable.

The compiled code consists of a sequence of 8-bit bytes in a language called Cintcode (Compact Interpretive Code) that was specifically designed for BCPL. Most Cintcode instructions occupy just one byte and correspond to simple operations performed on the Cintcode Abstract Machine. This machine has some central registers, the most important being PC, the program counter, that points to the next Cintcode instruction to execute, and A and B that are used during the evaluation of expressions. To see how Cintcode works we will execute this program one Cintcode instruction at a time. We can do this by typing the following piece of magic.

0.000> abort

!! ABORT 99: User requested
* x

0.000> demo

!! BPT 9: clihook

A= 0 B= 0 20092: K4G 1
* \ A= 0 B= 0 48532: L7
*
The `abort` command enters an interactive debugger and the debugging command `x` sets a break point just before `start` is entered. When we try to execute the `demo` command, we immediately hits this break point just as it is about to execute the Cintcode instruction `K4G 1` to enter the function `start`. The debugger issues the prompt * inviting us to type a debugging command. We then press the \\ key to cause one Cintcode instruction to be executed leaving the system about to execute `L7` at byte address 48532. We can see that both registers A and B contain zero.

The compiled code for `LET n = 7` is `L7` to load 7 into A followed by `SP3` to store A in the memory location whose address is `P+3` where P is another central register of the Cintcode Machine. At this moment P points to an area of memory used to hold local variables belonging to the function `start`, and the compiler has chosen to allocate the location at offset 3 to hold the variable n. Pressing \\ twice performs these two instructions, as follows:

```
*    A= 0   B= 0  48532:   L7
*    A= 7   B= 0  48533:   SP3
*    A= 7   B= 0  48534:   L0
```

Initialising `count` can be performed by pressing \\ twice more as follows:

```
*    A= 7   B= 0   48534:   L0
*    A= 0   B= 7  48535:   SP4
*    A= 0   B= 7  48536:   L1
```

Notice that when a value is loaded into A, the previous content is copied into B. We have now entered the `REPEAT` loop and are about to execute the compiled code for `count := count + 1` as can be seen by pressing \\ three more times.

```
*    A= 0   B= 7  48536:   L1
*    A= 1   B= 0  48537:   AP4
*    A= 1   B= 0  48538:   SP4
*    A= 1   B= 0  48539:   L1
```

L1 loads 1, AP4 adds the value in P4 (=count) and SP4 stores the result back in P4. The next three instructions test whether n equals 1.
L1 and LP3 load \( n \) and 1 in A and B, and the JNE 48545 instruction sets PC to 48545, if \( n \) is not equal to 1. Although the destination of the jump (48545) is too large to fit into an 8-bit byte, it is actually encoded as an 8-bit signed relative address in Cintcode. So jump instructions only occupy 2 bytes. Cintcode has a cunning mechanism to deal with jumps over large distances. The next four instructions test whether \( n \) is even.

The REM instruction sets A to the remainder after dividing \( n \) by 2, and the JNE0 48556 instruction sets PC to 48556 if this remainder is not zero, ie if \( n \) is odd. So rather than halving \( n \) we now compute \( n := 3n + 1 \) as follows:

LP3 L3 MUL multiplies \( n \) by 3 giving 21, A1 increments the result giving 22, and SP3 updates \( n \) with this new value. The next instruction J 48536 jumps us back to the start of the REPEAT loop.

We can remove the break point using the debugging command 0b9 and continue normal execution by typing c.

While in the debugger, pressing ? gives a useful summary of the possible debugging commands. For more information about Cintcode and the debugger see the BCPL manual (bcplman.pdf) available via my home page.
CHAPTER 4. THE BCPL CINTCODE SYSTEM

4.16 The Collatz Conjecture

The previous section contained a program that computed a sequence of numbers from a given starting value using a simple rule to determine whether to replace \( n \) by \( n/2 \) or \( 3n+1 \). Collatz conjectured in 1937 that the sequence always reaches 1 for every starting value. Surprisingly, no one has yet been able to prove this. You can learn all about the Collatz Conjecture by searching the web using the keyword Collatz.

If the conjecture is false, either there will be a starting value that generates a sequence either ending in a loop not containing one, or generating larger and larger numbers indefinitely. The following simple program (colllatz0.b) generates Collatz sequences from a given starting value.

GET "libhdr"

LET start() = VALOF
{ LET n = 7
  LET count = 0
  \{ count := count+1
    writef("%5i: %10i*n", count, n)
    IF n=1 BREAK
    TEST n MOD 2 = 0
    THEN n := n/2
    ELSE n := 3*n+1
  } REPEAT

  RESULTIS 0
}

In this program the starting value is held in \( n \). It outputs \( n \) and its position in the sequence before updating \( n \) with the next value. The test \( n \) MOD 2 = 0 determines whether \( n \) is even, replacing \( n \) by \( n/2 \) if it was, otherwise setting \( n \) to \( 3n+1 \). The program breaks out of the REPEAT loop if \( n \) reaches one, otherwise it goes on for ever outputing more and more numbers in the sequence. You can easily test a different starting value by modifying the declaration of \( n \). For instance, if the declaration was replaced by LET \( n = 123456789 \) you will find the sequence terminates at position 178.

An imperfection of this program is that it may suffer from overflow. The following program (collatz1.b) corrects this fault stopping with a message when it discovers that the next value will be too large to hold in a BCPL variable. This can only happen when \( n \) is odd and \( 3n+1 \) is greater than the largest number maxint that can be represented. So if \( n>(maxint-1)/3 \) the next number in the sequence will be too large.
GET "libhdr"

LET start() = VALOF
{ LET n = 123456789
  LET count = 0
  LET lim = (maxint-1)/3

  { count := count+1
    writef("%5i: %10i*n", count, n)
    IF n=1 BREAK
    TEST n MOD 2 = 0
    THEN { n := n/2
    }
    ELSE { IF n > lim DO
      { writef("Number too big*n")
        BREAK
      }
      n := 3*n+1
    }
  } REPEAT

RESULTIS 0
}

A variant of this program is given in Section 5.3 on page 253 that plots the relationship between sequence lengths and starting values.

Even with the program given above you will not be able to find a starting value that disproves the Collatz Conjecture since it has already been tested for all starting values up to $5 \times 2^{60}$. So if we are going to disprove the conjecture we must modify the program to use numbers of higher precision. The following program (collatz2.b) uses numbers with up to about one million binary digits. It starts as follows:

GET "libhdr"

MANIFEST {
  upb = (1<<20)-1 // ie about 1 million digits max
  mask = upb
  countt=10000 // count at start of test loop
  looplen=541 // Length of test loop
}

GLOBAL {
  digv:ug // digv is a circular buffer holding a number with up
           // to upb binary digits, with one digit per element.
CHAPTER 4. THE BCPL CINTCODE SYSTEM

\[
\begin{align*}
digp & \quad \text{Position of the least significant binary digit of the number.} \\
digq & \quad \text{Position of the most significant digit of the number.} \\
count & \quad \text{Position of the number in digv in the sequence} \\
digvc & \quad \text{Copy of the number at last checkpoint} \\
digcs & \quad \text{Count of digits in digvc.} \\
countchk & \quad \text{Count at last checkpoint} \\
digvt & \quad \text{Digits of the number at the start of the test loop} \\
digits & \quad \text{Count of digits in digvt} \\
eq1 & \quad \text{Returns TRUE if the number in digv is 1, ie digp=digq and digv!digp=1} \\
divby2 & \quad \text{Function to divide the number in digv by 2} \\
mulby3plus1 & \quad \text{Function to replace the number in digv by 3*n+1} \\
tracing & \quad \text{=TRUE causes the numbers to be output} \\
looptest & \quad \text{If TRUE, a loop of values is created} \\
\end{align*}
\]

The binary digits of the number are held in consecutive elements of the circular buffer \texttt{digv}, ordered from least to most significant digit. The least and most significant digits have subscripts \texttt{digp} and \texttt{digq}. If the number has only one digit \texttt{digp} will equal \texttt{digq}. \texttt{count} holds the position of the number in \texttt{digv} in the sequence. In order to detect a loop the number in \texttt{digv} is copied into \texttt{digvc} every time \texttt{count} is a power of two. Every time the next number is generated it is compared with the number in \texttt{digvc}. If there is a loop this test will eventually yield \texttt{TRUE}. To test that the loop detection mechanism works, the variable \texttt{looptest} is set to \texttt{TRUE}. This causes the number at position \texttt{count} (currently equal to 10000) to be copied into \texttt{digvt}, and every time \texttt{count} advances by \texttt{looplen} (currently 541) the number in \texttt{digv} is replaced by the number in \texttt{digvt}. The loop detection mechanism should detect this loop. Normally the program just output the position of each number in the sequence and its bit length, but if \texttt{tracing} is \texttt{TRUE} it also outputs the binary digits of each number.

The main program is as follows:

\[
\begin{align*}
\text{LET start()} & = \text{VALOF} \\
\{ \text{LET len } & = 5 \\
\quad \text{LET seed } & = 12345 \\
\quad \text{LET argv } & = \text{VEC 50} \\
\}
\text{UNLESS rdargs("len/n,seed/n,t/s,loop/s", argv, 50) DO} \\
\{ \text{writef("Bad args for collatz2\times n")} \\
\quad \text{RESULTIS 0} \\
\}
\end{align*}
\]
4.16. **THE COLLATZ CONJECTURE**

IF argv!0 DO len := !(argv!0)  // LEN/N
IF argv!1 DO seed := !(argv!1)  // SEED/N
tracing := argv!2  // T/S
looptest := argv!3  // LOOP/S

setseed(seed)

UNLESS 0<len<upb DO
{ writef("len must be in range 1 to \%n*n", upb)
  RESULTIS 0
}

digv := getvec(upb)
digvc := getvec(upb)
UNLESS digv & digvc DO
{ writef("upb too large -- more space needed\n")
  RESULTIS 0
}

digvt := 0

IF looptest DO
{ digvt := getvec(upb)
  UNLESS digvt DO
  { writef("upb too large -- more space needed\n")
    RESULTIS 0
  }
}

// Initialise digv with a random number of length len
digp := 0
FOR i = 0 TO len-2 DO digv!i := randno(2000)/1000
digv!(len-1) := 1  // Plant a most significant 1
digq := len-1  // Set position of the most significant digit
digcs := -1
count := 0

{ LET digs = ((digq+mask+1-digp) & mask) + 1

  count := count+1
  writef("\%9i \%6i: ", count, digs)
  IF tracing DO prnum()
  newline()
}
// Check whether the current number has been seen before
IF digs = digcs DO
  { // Numbers are the same length so check the digits
    writef("Checking the digits\n", digs)
    FOR i = 0 TO digs-1 UNLESS digvc!i=digv!((digp+i)&mask) GOTO notsame
    writef("*nLoop of length \%n found at count = \%n*\n", count-countchk, count)
  }
GOTO fin
}

notsame:
IF (count&(count-1))=0 DO
  { // Set new check value in digvc
    FOR i = 0 TO digs-1 DO digvc!i := digv!((digp+i)&mask)
    digcs := digs
    countchk := count // Remember the position of the check value
    writef("%9i %6i: Set new check value\n", count, digs)
  }
IF looptest DO
  { IF count=countt DO
    { // Create a loop starting here
      FOR i = 0 TO digs-1 DO digvt!i := digv!((digp+i)&mask)
      digts := digs
      writef("%9i: Save start of loop number\n", count)
    }
    IF count>countt & (count-countt) MOD looplen = 0 DO
      { // Return to start of test loop
        FOR i = 0 TO digs-1 DO digv!i := digvt!i
digp, digq := 0, digts-1
        writef("%9i: Restore start of loop number\n", count)
      }
  }
}

IF eq1() BREAK
TEST digv!digp=0 // Test for even
THEN divby2()
ELSE mulby3plus1()
} REPEAT

fin:
IF digv DO freevec(digv)
IF digvc DO freevec(digvc)
4.16. THE COLLATZ CONJECTURE

IF digvt DO freevec(digvt)
RESULTIS 0
}

The argument `len` specified the length in binary digits of the initial number in the sequence. This length must be between 1 and about one million. The digits of the starting value are chosen using a random number generator whose initial seed can be specified by the `seed` argument. If no seed is specified a seed of 12345 is initially chosen but then updated to a value depending on the current time of day. If no specific seed is chosen, it might happen that a random starting value of say 900000 digits was found that proved the conjecture false by ending with a loop not containing one, but not knowing the seed you would not be able to reproduce your fantastic discovery. Such a situation would be unimaginably annoying. If the argument `t` is given `tracing` will be set to `TRUE` and if `loop` is given `looptest` will be set to `TRUE` to test the loop detection mechanism.

The code is fairly self explanatory. It contains the loop detection mechanism and the code to generate a loop if `looptest` is `TRUE`. The call `eq1()` return `TRUE` if the current value in `digv` represents one. The current value in `digv` is even if its least significant digit is zero, that is if `digv!digp=0`. The call `divby2` divides the value in `digv` by 2, and `mulby3plus1()` multiplied the number in `digv` by three and adds one. These functions are defined below.

AND eq1() = digp=digq & digv!digp=1 -> TRUE, FALSE

AND divby2() BE
{ TEST digp=digq
  THEN digv!digp := 0
  ELSE digp := (digp+1)&mask
}

AND mulby3plus1() BE
{ // Calculate 3*n+1 eg
  // 1 +
  // 1011 +
  // 10110 =
  // ------
  // 100010
  LET carry = 1
  LET prev = 0
  LET i = digp

  { LET dig = digv!i
    LET val = carry+dig+prev
    digv!i := val&1
  }
carry := val>>1
prev := dig
IF i=digq DO
{ IF prev=0=carry RETURN // No need to lengthen the number
  i := (i+1)&mask
digv!i := 0
digq := i
  LOOP
}
i := (i+1)&mask
} REPEAT

AND prnum() BE
{ LET i = digp
  { LET dig = digv!i
    wrch('0'+dig)
    IF i=digq RETURN
    i := (i+1)&mask
  } REPEAT
}

The final function prnum() just outputs the digits of the number in digv.

Using this program you can test random starting values with lengths up to about one million binary digits, and if there is a value that disproves the Collatz Conjecture you might be lucky enough to find it. But I think that unlikely since I am convinced the conjecture is true.

### 4.17 The Pig Dice Game

This is a two player game that uses a six sided die, first described by John Scarne in 1945. It is an example of a *jeopardy race game* in which players have to repeatedly choose between making a small gain with high probability or possibly making a large loss with small probability. As the game proceeds the probabilities change. Each player has a current score. The players take turns with the die. The player with the die repeatedly throws it until either a one is thrown or the player decides to terminate the turn by saying “hold”. If a one is thrown the player’s score in left unchanged, but if the player holds, the numbers thrown during the turn are added to his score. In either case the die is given to the other player. The first player to reach a score of 100 wins.

The optimum choice of whether to roll the die or hold depends on the current scores of each player and the score accumulated in the current turn. The optimum choice turns out to be counter intuitive and complicated.
This program takes several numeric arguments: \( a_1, b_1, c_1, a_2, b_2 \) and \( c_2 \). If the \( a_1 \) is zero, player 1 is a user controlled by input from the keyboard. When it is player 1’s turn, pressing \( P \) causes the die to be thown and pressing \( H \) terminates the turn. If either a one is thrown or \( H \) is pressed the die is passes to the other player. If \( a_1 \) is non zero, player 1 is played by the computer using a strategy specified by \( a_1, b_1 \) and \( c_1 \). If \( a_1 \) is negative, player 1 is played by the computer using the optimum strategy based on data in the file \texttt{pigstrat.txt} , but if \( a_1 \) is greater than zero the computer uses a playing strategy defined by \( a_1, b_1 \) and \( c_1 \). You can think of the game state as a point \((\text{my}, \text{op}, \text{ts})\) in a 3D cube where \text{my} and \text{op} are player 1 and player 2’s scores and \text{ts} is player 1’s current turn score. If we assume that the \text{ts} axis is vertical, the coordinates \((\text{my}, \text{op})\) identify a point on a horizontal square. We can think of this square as the floor of a shed. The strategy is based on a sloping plane that can be thought of as the shed’s roof. If \text{ts} is less than the height of the roof at floor position \((\text{my}, \text{op})\) the strategy is to play the die, otherwise player 1 should hold. The orientation of the roof is defined by its height \( a_1 \) at the origin \((0,0)\), \( b_1 \) at position \((99,0)\) and \( c_1 \) at position \((0,99)\), and so, if \( \text{ts} \leq a_1 + (b_1-a_1)\times \text{my}/99 + (c_1-a_1)\times \text{op}/99 \), the strategy is to throw the die. The default settings for \( b_1 \) and \( c_1 \) are both set to \( a_1 \). This, of course, represents a horizontal roof of height \( a_1 \).

Player 2’s strategy is specified similarly using arguments \( a_2, b_2 \) and \( c_2 \). It is thus possible to cause the computer to play itself with possibly different strategies. A new game can be started by pressing \( S \), and the program can be terminated by pressing \( Q \). After each game, the tally of wins by each player is output. This is useful when comparing the effectiveness of different playing strategies. The program starts by declaring globals as follows.

GET "libhdr"

GLOBAL {
    stdin:ug
    stdout
    ch
    a1; b1; c1 // Player1's strategy parameters
    a2; b2; c2 // Player2's strategy parameters
    score1; score2 // The players' scores
    player // =0 if game ended,
             // =1 if it is player 1's turn,
             // =2 if it is player 2's turn.
    wins1; wins2 // Count of how often each player has won
    quitting // =TRUE when Q is pressed
    newgameP // The longjump arguments to
    newgameL // start a new game
    strategybytes; strategybytesupb; strategystream
}
Next is the definition of the main function `strategyrdch`.

```
LET strategyrdch() = VALOF
{ LET ch = rdch()
  UNLESS ch='(' RESULTIS ch
  // Ignore text enclosed within parentheses
  { ch := rdch()
    IF ch=endstreamch RESULTIS endstreamch
  } REPEAT UNTIL ch=''
} REPEAT
```

This function is used to read characters from the file `pigstrat.txt` when loading the optimum strategy. It behaves like `rdch` but skips over text enclosed in parentheses. The definition of `start` then follows.

```
LET start() = VALOF
{ LET days, msecs, filler = 0, 0, 0
  LET argv = VEC 50

  UNLESS rdargs("a1/n,b1/n,c1/n,a2/n,b2/n,c2/n",
             argv, 50) DO
  { writef("Bad argument(s) for pig*n")
    RESULTIS 0
  }

  a1, b1, c1 := 0, 0, 0 // Player1’s strategy
  a2, b2, c2 := -1, 0, 0 // Player2’s strategy
  wins1, wins2 := 0, 0
  quitting := FALSE

  IF argv!1 DO a1 := !(argv!0)
  b1, c1 := a1, a1
  IF argv!1 DO b1 := !(argv!1)
  IF argv!2 DO c1 := !(argv!2)
  IF argv!3 DO a2 := !(argv!3)
  b2, c2 := a2, a2
  IF argv!4 DO b2 := !(argv!4)
  IF argv!5 DO c2 := !(argv!5)

  newgameP, newgameL := level(), newgame
  datstamp(@days)
  setseed(msecs)

  The program first reads the command arguments, if any, that specify whether
  the two players are users, the computer or one of each. For the computer players
4.17. THE PIG DICE GAME

the value of the arguments specifies which strategy the computer will use. By
default, a1=0 causing player 1 to be the user and a2=-1 causing player 2 is the
computer playing the optimum strategy. Unless b1 and c1 are explicitly given
they are set equal to a1. The same convention applies to b2 and c2.

The variables newgameP and newgamel are set so the call
longjump(newgameP,newgamel) in function userplay will cause jump back into
start where a new game can be be started. Finally the random number seed
is set to a value based on the current time od day. The program continues as
follows.

strategybytes := 0
strategybytesupb := 100*100-1
strategystream := 0

IF a1<0 | a2<0 DO
{ // Load the optimum strategy data from file pigstrat.txt
strategybytes := getvec(strategybytesupb/bytesperword)
UNLESS strategybytes DO
{ writef("Unable to allocated strategybytes\n")
  GOTO fin
}

strategystream := findinput("pigstrat.txt")
UNLESS strategystream DO
{ writef("Unable to open pigstrat.txt\n")
  GOTO fin
}

selectinput(strategystream)

{ LET i, ch = 0, 0

{ LET x = 0

  ch := strategyrdch() REPEATUNTIL '0'<=ch<='9' | ch=endstreamch
  IF ch=endstreamch BREAK

  WHILE '0'<=ch<='9' DO
  { x :=10*x + ch - '0'
    ch := strategyrdch()
  }
  IF i <= strategybytesupb DO strategybytes%i := x
  i := i+1
  } REPEAT
  UNLESS i = 100*100 DO
newgame:
  score1, score2 := 0, 0

  writef("*nNew Game*n")

  If either player 1 or 2 is the computer playing the optimum strategy, one
  or both of a1 and a2 will be negative. The effect is to allocate an array,
  strategybutes, of 10,000 bytes and initialise it with the values specified in
  file pigstrat.txt. These values correspond to the smallest ts value for each
  (op,my) position where the optimum strategy is to hold.

  The program continues as follows.

  UNTIL quitting DO
  { play(1, a1, b1, c1)
    IF quitting BREAK
    play(2, a2, b2, c2)

    IF score1>=100 DO
      { wins1 := wins1 + 1
        writef("*nPlayer 1 wins*n")
      }
    IF score2>=100 DO
      { wins2 := wins2 + 1
        writef("*nPlayer 2 wins*n")
      }
    IF score1>=100 | score2>=100 DO
      { writef("Player1 scored %i3 games won %i3*n", score1, wins1)
        writef("Player2 scored %i3 games won %i3*n", score2, wins2)

        { writef("*nPress S or Q ")
          deplete(cos)
          ch := rch()
          IF ch='Q' | ch=endstreamch DO
            { newline()
              RESULTIS 0
            }
        }
      }
  }
4.17. THE PIG DICE GAME

        IF ch='S' GOTO newgame
    } REPEAT
    }
}

fin:
    IF strategybytes DO freevec(strategybytes)
    IF strategystream DO endstream(strategystream)
    RESULTIS 0
}

This part of the program causes players 1 and 2 to take turns alternately until one of them wins, at which time it outputs which player won, what their scores were and how many times each player has won. Pressing Q will terminate the program and pressing S will start a new game.

Input from the keyboard is read using the function rch which returns the next key as soon as it is pressed. The call writes("*b *b") erases the character that sardch echoed. The call deplete(cos) causes the buffered output to the currently selected output stream to be flushed, typically to the screen.

AND rch() = VALOF
{ LET c = capitalch(sardch())
  writes("*b *b")
  deplete(cos)
  RESULTIS c
}

The function play performs a player’s turn. It is defined as follows.

AND play(player, a, b, c) BE UNLESS score1>=100 | score2>=100 DO
{ LET turnscore = 0
 LET done = FALSE
 LET throws = 0
 LET turnv = VEC 100

UNLESS a DO writef("Press P, H or S*n")

{ LET score = score1
 LET opponent = score2

 IF player=2 DO score, opponent := score2, score1

 writef("*cPlayer%n: %i3 opponent %i3 turn %i3=",
     player, score, opponent, turnscore)
 IF throws>0 DO writef("%n", turnv!0)
FOR i = 1 TO throws-1 DO writef("+%n", turnv!i)

IF done DO
{ newline()
  TEST player=1
  THEN score1 := score1 + turnscore
  ELSE score2 := score2 + turnscore
  RETURN
}

IF strategy(turnscore, score, opponent, a, b, c) DO
{ // Throw
  LET n = randno(6)
  turnv!throws := n
  throws := throws+1
  turnscore := turnscore+n
  IF n=1 DO
  { turnscore := 0
    done := TRUE
  }
  UNLESS score+turnscore >= 100 LOOP
}
// Hold
done := TRUE
} REPEAT

If either player has already won, play returns immediately. Otherwise, it declares some local variables including the vector turnv which will hold all the values thrown in the current turn. The variable throws holds the number of times the die has been thrown in this turn. The choice of whether to hold or play is computed by the function strategy which defined below. As each decision is made it then outputs a line such as the following.

Player1: 14 opponent 23 turn 14=5+3+6

inviting the player to choose between another throw or holding. If done=TRUE the decision to hold has already been made and so the player’s score is updated and play returns. The strategy function is defined as follows.

AND strategy(turnscore, myscore, opscore, a, b, c) = VALOF
{ // Return TRUE to throw die, otherwise return FALSE.
  UNLESS a RESULTIS userplay()

  UNLESS turnscore RESULTIS TRUE // m/c always throws first time
4.17. THE PIG DICE GAME

// If a<0 use the optimum strategy based on data in pigstrat.txt
IF a<0 RESULTIS turnscore < strategybytes%(opscore*100+myscore)

RESULTIS turnscore < a + (myscore*(b-a) + opscore*(c-a))/99
}

If a is zero, the function userplay is called to let the user decide whether to throw or hold. If a is negative, the computer used the optimum strategy based on data in pigstrat.txt. Otherwise, a machine strategy is chosen based on the parameters a, b and c.

The next function reads the user's choice of whether to throw or play. It switches on the next character of input and takes appropriate action.

AND userplay() = VALOF
{ ch := rch()
  SWITCHON ch INTO
  { DEFAULT: LOOP
    CASE 'P': RESULTIS TRUE
    CASE endstreamch:
    CASE 'Q': quitting := TRUE
    CASE 'H': RESULTIS FALSE
    CASE 'S': longjump(newgameP, newgameL)
  }
} REPEAT

A typical run causing the computer to play itself is as follows. Here, strategies a1=20 and a2=27 are being compared. Repeatedly pressing S shows that the limit of 20 is better than 27.

0.010> pig a1 20 a2 27
New Game
Player1: 0 opponent 0 turn 0=4+3+6+1
Player2: 0 opponent 0 turn 21=5+3+3+4+3
Player1: 0 opponent 21 turn 0=4+2+6+1
Player2: 21 opponent 0 turn 20=6+2+4+6+2
Player1: 0 opponent 41 turn 0=4+1
Player2: 41 opponent 0 turn 0=1
Player1: 0 opponent 41 turn 21=2+3+3+6+2+5
Player2: 41 opponent 21 turn 20=5+4+3+6+2
Player1: 21 opponent 61 turn 0=1
Player2: 61 opponent 21 turn 22=6+4+4+5+3
Player1: 21 opponent 83 turn 20=3+5+5+3+2+2
Player2: 83 opponent 41 turn 20=6+5+3+6
CHAPTER 4. THE BCPL CINTCODE SYSTEM

Player 2 wins
Player1 scored  41 games won  0
Player2 scored 103 games won  1

Press S or Q

4.17.1 The Optimum Strategy

As mentioned above the optimum strategy for the pig dice game is complicated and counter intuitive. It is also quite hard to discover. The optimum strategy can be represented by a $100 \times 100 \times 100$ cube of values indicating whether it is best to hold or play the die for each state of the game. The program pigstrategy.b is my attempt to calculate the optimum strategy, leaving the result in the file pigcube.txt.

A point in the cube can be given coordinates $(op, my, ts)$ representing the opponent’s score, the player’s score and the current turn score, respectively. So for each position $(op, my, ts)$ we need a flag to specify whether to hold or play. It is also helpful for each position to hold the probability of winning. We can represent the cube by an array called cube with one million $(=100 \times 100 \times 100)$ elements. The element $\text{cube}[i]$ will hold $(\text{prob} \ll 1 | \text{flag})$, where $i=op*100*100+my*100+ts$. prob holds the probability of a win represented as a scaled number with 8 decimal digits after the decimal point and flag=1 indicates that the best strategy at this position is to hold. The setting of $\text{cube}[i]$ depends on the settings of other elements of $\text{cube}$, so we essentially have one million simultaneous equations to solve. Using a simple recursive function will fail because the equation for $\text{cube}[i]$ often depends on its own value, and this will cause a recursive loop that is hard to avoid. So we probably have to resort to a so called relaxation method, in which we make an initial guess for each element of $\text{cube}$ and then repeatedly update each $\text{cube}[i]$ with a new estimate based on the previous elements of $\text{cube}$. In general there is no guarantee that relaxation will converge, but luckily for this problem it seems to work and converges to a reasonable looking answer reasonably rapidly.

Once the answer has been found two files are written. The first, called pigcube.txt holds the resulting winning probability and flag for every element of the cube. This file is about 13 million bytes long. The second file, called pigstrat.txt holds a sequence of 10,000 numbers giving the lowest turn score for which holding is the best strategy for each $(opponent \ score, \ player \ score)$ pair. This is read by the pig.b program to allow it to play using the optimum strategy.

A few lines of pigcube.txt are as follows:

\[
\begin{align*}
(21 & 25 & 0): 0.56765260P & 0.57086016P & 0.57421506P & 0.57772383P & 0.58139457P \\
(21 & 25 & 5): 0.58523565P & 0.58925465P & 0.59346038P & 0.59785324P & 0.60244720P
\end{align*}
\]
4.17. THE PIG DICE GAME

... (25 21 10): 0.54151407P 0.54654803P 0.55182253P 0.55733909P 0.56310562P
(25 21 15): 0.56908934P 0.57538043P 0.58202627P 0.58986137P 0.59620498P
(25 21 20): 0.60368852P 0.61128274P 0.61977279H 0.62886120H 0.63796413H
(25 21 25): 0.64700159H 0.65618399H 0.66533814H 0.67442849H 0.68337390H
...

(31 25 0): 0.48526691P 0.48858882P 0.49206411P 0.49569825P 0.49949962P
(31 25 5): 0.50347724P 0.50763977P 0.51199559P 0.51655001P 0.52130443P
...

If you run pigstrategy with the trace option (-t) specified, it will generate considerable output including the following lines.

... (31 25 0): 0.48526691P
(21 25 0): 0.56765260P (25 21 12): 0.55182253P (25 21 13): 0.56733909P
(25 21 14): 0.56310562P (25 21 15): 0.56908934P (25 21 16): 0.57538043P
(25 21 10): 0.51473309H 0.54151407P => (25 21 10): 0.54151407P diff=0.00000000
...

These lines were generated when pigstrategy was computing a new setting for position (25 21 10) of the cube, that is when the opponent score was 25, the player’s score was 21 and the current turn score was 10. The first line indicates that the opponent will win with a probability 0.48526691 if the player holds. Note that 31 is the sum of the player’s score and the current turn score. This becomes the opponent score when the opponent begins to play. If the player chooses to play the die, we must take the average of six probabilities corresponding to the possible throws of the die. If the number one is thrown, the opponent gains the die and has a winning probability of 0.56765260 corresponding to position (21 25 0). Otherwise, the player accumulates in the turn score a value between 2 and 6 with varying probabilities held in positions (25 21 12) to (25 21 16). When computing the average, we add 3 before dividing by 6 so as to round the result properly. The last line shows the probability of winning if holding (0.51473309) or continuing to play (0.54151407). The best strategy for this state is therefore to play. This last line also indicates that the new estimate is the same as the previous one.

A few lines of pigstrat.txt is as follows.

... (25 0): 23 23 23 23 23 23 23 23 23 23
(25 10): 23 23 22 22 22 22 22 22 22 22
(25 20): 22 22 22 22 22 22 21 21 21 20
(25 30): 20 20 20 20 20 20 20 19 19 19
(25 40): 19 19 18 18 18 18 18 19 19 19
...
This indicates that when the opponent score is 25 and the player’s score is 21, the lowest turn score for which hold is the best choice is 22. You will notice that this is compatible with the line starting (25 21 20) from the file pigcube.txt where the entry for turn score 22 is 0.61977279H.

A pictorial representation of the optimum strategy is shown in Figure 4.3. The red and green axes identify player1 and player2’s current scores and the blue axis holds player1’s current turn score. The solid material in the cube represents all the games states where player1’s best strategy is to throw the die. Notice that the surface is quite complex and contains some overhangs. The image is based on data in pigcube.txt which can be read by a program called prepPicFic.b to generate data in the file cubpic.txt. This is subsequently read by plotPicFic.b to generate the 3D image shown in Figure 4.3. The image is drawn using the SDL Graphics Library and so you should read the next chapter before trying to understand how plotPicFic.b works.

Figure 4.3: The Optimum Strategy for the Pig Dice Game
4.18 The Enigma Machine

Having recently visited Bletchley Park with my young grandson, I was pleased to see how fascinated he was with the German Enigma Machine used between 1939 and 1945 to encipher messages that were typically transmitted by radio using morse code. Since a program to simulate the machine is quite simple, it is a good programming example with some added interest.

The Allies could easily read the enciphered text so it was necessary to use a cipher code that was impossible to break. The method chosen was to use the Enigma Machine which could translate plain text into enciphered text depending on how the machine was initially set up, and since the machine could be set up in more than 1000 million million ways each generating completely different translations, it was thought to be unbreakable. The machine was battery operated and small and light enough to be used in aircraft, submarines and on the battle front.

The program described in this section simulates the M3 version of the Enigma machine, and its implementation was influenced by a C program written by Fauzan Mirza, and the excellent document and Enigma Machine simulator written by Dirk Rijmenants. For more information, I strongly recommend you visit the following web sites:

http://users.telenet.be/d.rijmenants
http://www.rijmenants.blogspot.com

The machine details and example message have been taken from Rijmenants’ document with permission.

The Enigma machine has a keyboard with keys labelled from A to Z and 26 lights labelled A to Z. When a key was pressed one of the lights will turn on indicating the translated letter. The electrical path from the key to the light is complex. It first passes through a plug board which can be set up to swap typically 10 pairs of letters. For instance, one cable could cause A to be turned into J and J to be turned into A. After the plug board, the signal then enters a sequence of three rotors. Each rotor has 26 spring loaded terminals to the right pressing a plate with 26 contacts arranged in a circle. To the right of the rightmost rotor the contact plate is fixed and connected to the 26 wires from the left side of the plug board. The left side of each rotor has a similar circular contact plate that either makes contact with the terminals of the rotor on its left, or, for the left most rotor, the spring loaded terminals of a reflector plate. The reflect connects the letter positions in pairs in an essentially random fashion. The wiring of each rotor is also essentially random. Once the signal from the pressed key has passed through the plug board and three rotors to the reflector, it returns back through the rotors and plug board to provide power for one of the lights, giving the translated letter. Notice that, because of the way the machine works pressing A, say, will never translate into A. Note also that if pressing A
translates to J, say, then, from the same initial setting, pressing J will translate
to A. This property allows the machine to be used both to encode messages and
decode them.

There is a choice of 5 differently wired rotors (named I, II, III, IV and V)
which can be placed in the machine in any order, and there are two possible
reflectors named B and C. Before translating a message the correct rotors must
be selected and placed in the machine in the required order and each be set to one
of 26 initial positions. Each rotor has a small window displaying a letter giving
its current position. But this is complicated by the fact that the ring of letters
for each rotor can be in any one of 26 positions relative to the its wiring core.
These ring settings have to be done before the rotors are placed in the machine.

Every time a key is pressed one or more of the rotors advance by one position
completely changing the translation of each letter. So pressing Q, say, repeatedly
will generate a seemingly random sequence of letters.

The program for this simulator is in bcplprogs/raspi/enigma-m3.b, and
since it is quite long and it will be described in small chunks. With some com-
ments removed, it starts as follows.

GET "libhdr"

GLOBAL
{ newvec:ug
  spacev; spacep; spacet

  inchar   // String of input characters
  outchar  // String of output characters
  len      // Number of characters in the input string
  ch       // Current keyboard character

  stepping  // =FALSE to stop the rotors from stepping
  tracing   // =TRUE causes signal tracing output

  rotorI;   notchI
  rotorII;  notchII
  rotorIII; notchIII
  rotorIV;  notchIV
  rotorV;   notchV
  reflectorB
  reflectorC

  rotorLname; rotorMname; rotorRname
  reflectorname

  // Ring and notch settings of the selected rotors}
4.18. THE ENIGMA MACHINE

ringL; ringM; ringR
notchL; notchM; notchR

// Rotor start positions at the beginning of the message
initposL; initposM; initposR
// Rotor current positions
posL; posM; posR;

// The following vectors have subscripts from 0 to 25
// representing letters A to Z
plugboard
rotorFR; rotorFM; rotorFL
reflector
rotorBL; rotorBM; rotorBR // Inverse rotors

// Variables for printing signal path
pluginF
rotorRinF; rotorMinF; rotorLinF
reflin
rotorLinB; rotorMinB; rotorRinB
pluginB; plugoutB

// Global functions
newvec; setvec
pollrdch; rch; rdlet
rdrotor; rdringsetting
setplugpair; prplugboardspairs; setrotor
step_rotors; rotorfn; encodestr; enigmafn
prsigwiring; prsigreflector; prsigrotor; prsigplug; prsigkbd
prsigline; prsigpath
}

This inserts the library declarations from libhdr and then declares the global variables required by this program. The first few newvec, spacev, spacep and spacet are used in connection with allocation of space. The variables inchar, outchar and len hold the string of message letters, the enciphered translation and the message length. The variable ch normally holds the latest character typed by the user.

Two debugging aids are available controlled by stepping and tracing. If stepping is FALSE the rotors remain fixed and do not step as each message character is typed. If tracing is TRUE, when each message character is typed, the program outputs a diagram showing the signal path within the machine between the pressed key and the resulting light. For instance, with the program’s default settings, a Q translates to D and the output as shown in Figure 4.4.
Notice that the keyboard, plug board, rotors and reflector appear in rectangles with sides composed of horizontal and vertical lines (– and |). The signal path is represented by horizontal (< and >) and vertical (^ and v) arrows, using an asterisks (*) whenever the path turns a right angle. The current letter positions of
the three rotors are enclosed in square brackets ([ and ]). The current positions of the three rotor notches are shown by equal signs to the left of each rotor and the ring setting for each rotor is shown by an asterisk (*) between the ring letter and the letter A on the left side of the wiring core.

The globals rotorI to rotorV hold strings of length 26 giving the wiring of each of the available rotors. The string for rotor I is "EKMFPLGQDQVZNTOWYHXUSPAIBRCJ", indicating that the terminal at position A on the right hand side of the rotor is connected to the contact at position E on the left side. Similarly terminal B is connected to contact K.

Each rotor has a circular disc on its left size containing a notch. It is a fixed position relative to the rotor’s ring of letters, but this position is different for each rotor. If a rotor has its notch at the A position of the machine then it and the one to its left will both advance by one letter position the next time a key is pressed. This mechanism is covered in more detail on page 106 when the function step_rotors is described. The notch positions of each rotor are held in notchI to notchV. These are given as ASCII characters, for instance notchI is set to 'Q'.

The strings representing the wirings of reflectors B and C are held in reflectorB and reflectorC. The names of the left, middle and right and rotors are held as strings in rotorLname, rotorMname and rotorRname, and the name of the current reflector is held in reflectorname.

The ring settings and notch positions of the left, middle and right hand rotors are held in ringL, notchL, ringM, notchM, ringR and notchR. These are all numbers in the range 0 to 25 representing A to Z.

The initial position of the left hand rotor (just before the message in inchar is processed) is held in initposL as a number in the range 0 to 25 representing A to Z, and initposM and initposR hold the corresponding positions of the middle and right hand rotors. These are needed every time the entire input message is re-enciphered, for instance, whenever one of the machine settings is changed by the user. The current positions of the rotors are held in posL, posM, posR.

For convenience the wiring of the plug board, the rotors and the reflector are held in the vectors plugboard, rotorFR, rotorFM, rotorFL, reflector, rotorBL, rotorBM and rotorBR. Their subscripts range from 0 to 25 corresponding to positions A to Z, and their elements are in the same range. For instance, if the plug board maps letter A to B, then plugboard!0 will equal 1. Since the plug board is its own inverse, plugboard!1 will equal 0. The vector rotorFR holds the mapping (in the forward direction) of the letter as it passes through the right hand rotor from right to left. If the right hand rotor is V, it maps B to Z, so rotorFR!1 is equal to 25. For the return (backward) path from left to right through this rotor, the letter W maps to R. This is implemented using a second vector called rotorBR. Note that rotorBR!22 will equal 17.

When a key is pressed, the signal path through the plug board, rotors and reflector is computed and recorded in the global variables pluginF, rotorRinF,
rotorMinF, rotorLinF, reflin, rotorLinB, rotorMinB, rotorRinB, pluginB and plugoutB. These all have values in the range 0 to 25 corresponding to positions A to Z, and are used by the functions that draw the diagram representing the signal path from the pressed key to the corresponding light.

Although not strictly necessary, all the functions in this program are given global locations. This is primarily to aid debugging, since, for instance, it simplifies the setting of break points.

### 4.18.1 enigma-m3 functions

In this section the functions defined in enigma-m3.b are described in turn.

```bcpl
LET newvec(upb) = VALOF
{ LET p = spacep - upb - 1
  IF p<spacev DO
    { writef("More space needed\n")
      RESULTIS 0
    }
    spacep := p
  RESULTIS p
}
```

A reasonably sized area of memory is allocated using getvec in the main function start. The base and limit of this memory are placed in spacev and spacet. The function newvec sub-allocates vectors from this memory by decrementing spacep by an appropriate amount each time. The advantage of this scheme is that we can allocate all the memory we need by one call of getvec and then return it all by one call of freevec just before the program terminates. There is no need to return all the sub-allocated vectors separately.

```bcpl
LET setvec(str, v) BE
  IF v FOR i = 0 TO 25 DO v!i := str%(i+1) - 'A'

LET setrotor(str, rf, rb) BE
  IF rf & rb FOR i = 0 TO 25 DO
    { rf!i := str%(i+1)-'A'; rb!(rf!i) := i }
```

These two functions convert the character string versions of rotor and reflector wiring strings to the integer vector form as required by the program. Notice that setrotor initialises both the forward and backward wiring vectors for the rotors.

```bcpl
LET pollrdch() = VALOF
{ LET ch = sys(Sys_pollsardch)
```
4.18. THE ENIGMA MACHINE

UNLESS ch=-3 RESULTIS ch
    delay(100) // Wait 100 msecs and try again
} REPEAT

This function uses the call sys(Sys_pollsardch) to attempt to read the latest character typed on the keyboard. If no character is available, represented by -3, it waits a tenth of a second before trying again. The main reason for using polled input is to get instant response to each character typed on the Enigma Machine.

The next function, start, is quite long and so its description is broken into smaller pieces.

LET start() = VALOF
{ LET argv = VEC 50

    UNLESS rdargs("-t/s", argv, 50) DO
    { writef("Bad arguments for enigma-m3*n")
        RESULTIS 0
    }

    writef("*nEnigma M3 simulator*n")
    writef("Type ? for help*n*n")

    tracing := TRUE // Default setting of tracing
    IF argv!0 DO tracing := ~tracing // -t/s

    spacev := getvec(1000)
    spacet := spacev+1000
    spacep := spacet

    When enigma-m3 is called, it can be given a switch argument -t which toggles the tracing option. Currently the default setting is to have tracing enabled. The last three lines allocate some memory, initialising spacev, spacet, spacep appropriately.

    // Set the rotor and reflector wirings
    // and the notch positions.

    // Input  "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    rotorI := "EKMFLGDQVZNTOWYHXUSPAIBRCJ"; notchI := 'Q'
    rotorII := "AJDKSIRUXBLHWTMCQGZNPYFVOE"; notchII := 'E'
    rotorIII := "BDFHJLCPRTXVZNYEIWGAKMUSQO"; notchIII := 'V'
    rotorIV := "ESOVPZJAYQUIRHXLNFTGKDCMWB"; notchIV := 'J'
    rotorV := "VZBRGITYUPSDNHLXAWMJQOFCK"; notchV := 'Z'

    reflectorB := "YRUHQSLDPXNGOKIEBFZCWJAT"
    reflectorC := "FVPJIAOYEDRZXWQGCTKUXSBHNML"
These assignments set the wiring strings of the five rotors and their corresponding notch positions, together with the wiring of the two reflectors.

```plaintext

// Allocate several vectors
rotorFL := newvec(25)
rotorFM := newvec(25)
rotorFR := newvec(25)
rotorBL := newvec(25)
rotorBM := newvec(25)
rotorBR := newvec(25)
plugboard := newvec(25)
reflector := newvec(25)
inchar := newvec(255)
outchar := newvec(255)

UNLESS rotorFL & rotorFM & rotorFR &
    rotorBL & rotorBM & rotorBR &
    plugboard & reflector &
    inchar & outchar DO
    { writef("*nMore memory needed*n")
    GOTO fin
    }

This code allocates all the vectors needed by the program and places them in their global locations. It checks that they have all been allocated successfully.

// Set default encryption parameters, suitable for the // example message.

setvec(reflectorB, reflector)
reflectormap := "B"
setrotor(rotorI, rotorFL, rotorBL)
rotorLname, notchL := "I ", notchI - 'A'
setrotor(rotorII, rotorFM, rotorBM)
rotorMname, notchM := "II ", notchII - 'A'
setrotor(rotorV, rotorFR, rotorBR)
rotorRname, notchR := "V ", notchV - 'A'

ringL := 06-1; ringM := 22-1; ringR := 14-1
initposL := 'X'-'A'; posL := initposL
initposM := 'W'-'A'; posM := initposM
initposR := 'B'-'A'; posR := initposR

FOR i = 0 TO 25 DO plugboard!i := i
```
4.18. THE ENIGMA MACHINE

// Perform +PO+ML+IU+KJ+NH+YT+GB+VF+RE+DC
// to set the plug board.

setplugpair('P', 'O')
setplugpair('M', 'L')
setplugpair('I', 'U')
setplugpair('K', 'J')
setplugpair('N', 'H')
setplugpair('Y', 'T')
setplugpair('G', 'B')
setplugpair('V', 'F')
setplugpair('R', 'E')
setplugpair('D', 'C')

// writef("Set the example message string\n")

{ LET s = "QBLTWLDAHHYEEOEFPTWYBLENDPMKOXLDFAMUDWIJDXRJZ"
  len := s%0
  FOR i = 1 TO len DO inchar!i := s%i
}

This code initialises the Enigma Machine in the way required to decode the following encrypted message.

U6Z DE C 1510 = 49 = EHZ TBS =

TVEXS QBLTW LDAHH YEDEF
PTWYB LENDP MKOXL DFAMU
DWIJD XRJZ=

It was sent on the 31st day of the month from C to U6Z at 1510 and contains 49 letters. The recipient had the secret daily key sheet containing the following line for day 31:

31 I II V 06 22 14 PO ML IU KJ NH YT GB VF RE DC EXS TGY IKJ LOP

This shows that the enigma machine must be set up with rotors I, II and V in the left, middle and right positions with ring settings 6, 22 and 14, respectively. The plug board should be set with the 10 specified connections.

The rotor start positions should be set to EHZ then the three letters TBS should be typed. This generates XWB which is the start positions of the rotors for the body of the message. The first group TVEXS is not enciphered and just confirms we have the right daily key since it contains EXS which appears in the daily key sheet, together with two random letters. Decoding begins at the second group QBLTW. To decode the example message using this program type the following:
This generates the following decrypted text (with spaces added).

**DER FUEHRER IST TOD X DER KAMPF GEHTWEITER X DOENITZ X**

```
len := 0
stepping := TRUE
ch := ’*n’
encodestr()
```

These four lines complete the initialisation of the program. Setting `len` to zero sets the machine to encode letters typed from the keyboard, but if the assignment is commented out the program will decode the example message. The call `encodestr()` encodes all the letters in `inchar` placing their translations in `outchar`.

Now follows the main loop of the simulator. It starts as as follows.

```
{ // Start of main input loop
  IF ch=’*n’ DO { writef("*n> "); deplete(cos); ch := 0 }

  UNLESS ch DO rch()

  SWITCHON ch INTO
  { DEFAULT:
    CASE ’*s’: ch := 0 // Cause another character to be read.
    CASE ’*n’: LOOP

    CASE endstreamch:
    CASE ’.’: BREAK

  }

  IF ch=’?’ THEN newline()
  writef("? Output this help info*n")
  writef("#rst Set the left, middle and *
          *right hand rotors to r, s and t where*n")
  writef(" r, s and t are single digits *
          *in the range 1 to 5 representing*n")
```

It outputs a prompt, if necessary, and reads the next character from the keyboard unless one is already available. It then switches on this character. The character is ignored if it is a space or has no CASE label provided. Dot (.) and the end-of-stream character both cause the program to terminate.
writef("rotors I, II, ..., V.*n")
writef("!abc Set the ring positions for the *
  *left, middle and right rotors where*n")
writef("a, b and c are letters or numbers *
  *in the range 1 to 26 separated*n")
writef("=abc Set the initial positions of the *
  *left, middle and right hand rotors*n")
writef("/B Select reflector B*n")
writef("/C Select reflector C*n")
writef("+ab Set swap pairs on the plug board, *
  *a, b are letters.*n")
writef("Setting a letter to itself removes *
  *that plug*n")
writef("[ Toggle rotor stepping*n")
writef("", Print the current settings*n")
writef("letter Add a message letter*n")
writef("- Remove the latest message *
  *character, if any*n")
writef(". Exit*n")
writef("space and newline are ignored*n")
ch := 'n'
LOOP

This causes some help information to be output when the user types a question
mark.

CASE '#': // Select the rotors, eg #125
{ LET str, name, notch = 0, 0, 0
  ch := 0
  rd rotor(@str)
  set rotor(str, rotorFL, rotorBL)
  rotorLname, notchL := name, notch-'A'
  rd rotor(@str)
  set rotor(str, rotorFM, rotorBM)
  rotorMname, notchM := name, notch-'A'
  rd rotor(@str)
  set rotor(str, rotorFR, rotorBR)
  rotorRname, notchR := name, notch-'A'
  writef("nRotors: %s %s %s notches %c%c%c*n", 
    rotorLname, rotorMname, rotorRname, 
    notchL+'A', notchM+'A', notchR+'A')
  encode str()
  ch := 'n'
  LOOP
}
This reads a command of the form \#abc where \(a, b\) and \(c\) are digits in the range 1 to 5 representing rotor numbers. It specifies which rotors should be placed in the left, middle and right hand positions. Note that the assignment \(\text{ch}:=0\) forces \text{rdrotor} to call \text{rch} to read the next keyboard character. The call \text{rdrotor('@str')} reads the next rotor number and sets the local variables \text{str}, \text{name} and \text{notch} to the wiring string, the rotor name and its notch letter, respectively. Three calls of \text{rdrotor} are made to obtain the appropriate settings for the three rotors.

```plaintext
CASE '!' : // Set ring positions, eg !6 22 14 or !fvn
  ch := 0
  ringL := rdringsetting()
  ringM := rdringsetting()
  ringR := rdringsetting()
  printf("*nRing settings: %c%c%c*n",
         ringL+'A', ringM+'A', ringR+'A')
  encodestr()
  ch := '*n'
LOOP
```

This reads a command of the form \!abc where \(a, b\) and \(c\) are ring positions given as letters or numbers in the range 1 to 26 separated by spaces. They correspond to the ring settings of the rotors in the left, middle and right hand positions.

```plaintext
CASE '=' : // Set the rotor positions
  ch := 0
  initposL := rdlet() - 'A'
  initposM := rdlet() - 'A'
  initposR := rdlet() - 'A'
  printf("*nRotor positions: %c%c%c*n",
         initposL+'A', initposM+'A', initposR+'A')
  encodestr()
  ch := '*n'
LOOP
```

This reads a command of the form \=abc where \(a, b\) and \(c\) are rotor positions given as letters. They correspond to the positions of the left, middle and right hand rotors.

```plaintext
CASE '/ ' : // Set reflector B or C
  if ch = 'B'
  { setvec(reflectorB, reflector)
```

```plaintext
```
4.18. THE ENIGMA MACHINE

```plaintext
reflectorname := "B"
BREAK
}
IF ch = 'C' DO
{ setvec(reflectorC, reflector)
  reflectorname := "C"
  BREAK
}
writef("*nB or C required*n")
} REPEAT

writef("*nReflector %s selected*n", reflectorname)
encodestr()
ch := '*n'
LOOP

The commands /B and /C select which reflector to use.

CASE '+':// Set a plug board pair
{ LET a, b = ?, ?
  rch()
  a := ch
  rch()
  b := ch
  IF 'A'<=a<='Z' & 'A'<=b<='Z' DO
  { setplugpair(a, b)
    BREAK
  }
  writef("*n+ should be followed by two *
    *letters, eg +AB*n")
} REPEAT

encodestr()
ch := '*n'
LOOP

A command of the form +ab where a and b are letters sets a cable between letters a and b. But if a and b are the same letter, any cable between a and another letter is removed. It calls setplugpair to deal with these cases.

CASE '|': // Toggle rotor stepping
    stepping := ~stepping
    TEST stepping
    THEN writef("*nRotor stepping enabled*n")
```
This case outputs the current settings of the machine, namely which rotors have been selected, what their notch and ring positions are, what the initial and current rotor positions are, what the plug board connections have been made, what the current message is and its encoding. Typical output is as follows:

```
> ,
Rotors: I II V
Notches: Q E Z
Ring setting: F-06 V-22 N-14
Initial positions: X W B
Current positions: X W D
Plug board: BG CD ER FV HN IU JK LM OP TY
in: QQ
out: DJ
```
CASE '−': // Remove one message character
    IF len>0 DO len := len-1
    encodestr()
    ch := '∗n'
    LOOP

The command minus (−) removes one letter from the input message and then re-encode the entire message just in case tracing was enabled.

CASE '~': // Toggle signal tracing
    tracing := ~tracing
    TEST tracing
    THEN writef("∗nSignal tracing now on∗n")
    ELSE writef("∗nSignal tracing turned off∗n")
    ch := '∗n'
    LOOP

The twiddles (~) command toggles the tracing option.

    CASE 'A':CASE 'B':CASE 'C':CASE 'D':CASE 'E':
    CASE 'F':CASE 'G':CASE 'H':CASE 'I':CASE 'J':
    CASE 'K':CASE 'L':CASE 'M':CASE 'N':CASE 'O':
    CASE 'P':CASE 'Q':CASE 'R':CASE 'S':CASE 'T':
    CASE 'U':CASE 'V':CASE 'W':CASE 'X':CASE 'Y':
    CASE 'Z':
        IF len<255 DO len := len + 1
        inchar!len := ch
        encodestr()
        ch := '∗n'
        LOOP

If a letter is typed, it is added to the end of the message string and then the entire message re-encoded by a call of encodestr. Notice that the message cannot grow to a length greater than 255 letters.

    }
    } REPEAT

newline()

fin:
    IF spacev DO freevec(spacev)

RESULT IS 0
}
CHAPTER 4. THE BCPL CINTCODE SYSTEM

These last few lines end the \textsc{switchon} command and the main command loop. Before returning from the main function \texttt{start}, it returns to free store the memory, if any, pointed to by \texttt{spacev}.

\begin{verbatim}
AND setplugpair(a, b) BE
{ // a and b are capital letters
   LET c = ?
   a := a - 'A'
   b := b - 'A'
   c := plugboard!a
   UNLESS plugboard!a = a DO
      { // Remove previous pairing for a
         plugboard!a := a
         plugboard!c := c
      }
   c := plugboard!b
   UNLESS plugboard!b = b DO
      { // Remove previous pairing for b
         plugboard!b := b
         plugboard!c := c
      }
   UNLESS a=b DO
      { // Set swap pair (a, b).
         plugboard!a := b
         plugboard!b := a
      }
}

This function is used by the plus (+) command to place a plug board cable between letters $a$ and $b$, which are given as character constants in the range 'A' to 'Z'. If $a$ and $b$ are equal, any previous cable to $a$ is removed.

\begin{verbatim}
AND rdlet() = VALOF
{ IF ch=0 DO rch()
   WHILE ch='*s' DO rch()
   IF 'A'<=ch<='Z' DO
      { LET res = ch
         ch := 0
         RESULTIS res
      }
   writeln("*nA letter is required*n")
   ch := 0
} REPEAT

AND rch() BE
\end{verbatim}
4.18. THE ENIGMA MACHINE

{ // Read a keyboard key as soon as it is pressed.
  ch := capitalch(pollrdch())
  wrch(ch)
  deplete(cos)
}

The function rdlet reads a letter from the keyboard, and rch reads any character from the keyboard, replacing lower case letters by their upper case equivalents.

AND rdrotor(v) BE
{ // Returns the rotor wiring string
  // result2 is the rotor name: I, II, III, IV or V
  IF ch=0 DO rch()
  WHILE ch='*s' DO rch()

  IF '0'<=ch<='5' DO
    { IF ch='1' DO v!0, v!1, v!2 := rotorI, "I ", notchI
      IF ch='2' DO v!0, v!1, v!2 := rotorII, "II ", notchII
      IF ch='3' DO v!0, v!1, v!2 := rotorIII, "III", notchIII
      IF ch='4' DO v!0, v!1, v!2 := rotorIV, "IV ", notchIV
      IF ch='5' DO v!0, v!1, v!2 := rotorV, "V ", notchV
      ch := 0
      RETURN
    }
  writef("*nRotor number not in range 1 to 5*n")
  ch := 0
} REPEAT

This function reads a digit in the range 1 to 5 and sets v!0, v!1 and v!2 to the wiring string, the name and the notch letter of the specified rotor.

AND rdringsetting() = VALOF
{ // Return 0 to 25 representing ring setting A to Z
  IF ch=0 DO rch()

  WHILE ch='*s' DO rch()

  IF 'A'<=ch<='Z' DO
    { LET res = ch-'A'
      ch := 0
      RESULTIS res
    }
IF '0'<= ch <= '9' DO
{ LET n = ch-'0'
   rch()
   IF '0'<= ch <= '9' DO n := 10*n + ch - '0'
   // n = 1 to 26 represent ring settings of A to Z
   // encoded as 0 to 25
   ch := 0
   IF 1<=n<=26 RESULTIS n - 1
   writef("*nA letter or a number in range 1 to 26 required*n")
}
} REPEAT

This function reads a ring setting as either a letter or a number in the range 1 to 26. It returns a value in the range 0 to 25.

AND prplugboardpairs() BE FOR a = 0 TO 25 DO
{ // Print plug board pairs in alphabetical order
   LET b = plugboard!a
   IF a < b DO writef("%c%c ", a+'A', b+'A')
}

This function outputs the current wiring of the plug board as letter pairs in alphabetic order.

AND step_rotors() BE IF stepping DO
{ LET advM = posR=notchR | posM=notchM
   LET advL = posM=notchM

   posR := (posR+1) MOD 26       // Step the right hand rotor
   IF advM DO posM := (posM+1) MOD 26  // Step the middle rotor
   IF advL DO posL := (posL+1) MOD 26  // Step the left rotor
}

Whenever a key is pressed one or more rotors advance by one letter position. Each rotor has a notch disk attached to the letter ring on its left side. A notch is shaped like an asymmetric V with one edge on a radius line towards the centre of the rotor and the other at an angle of about 70 degrees forming a gentle slope back to the rim of the disk. On the right hand side of each rotor there is a disk, we will call the ratchet disk, containing 26 equally spaced notches of similar shape. Between the middle and right hand rotors there is a spring loaded pawl that is typically just clear of the rim of the notch disk to its right. When a key is pressed, the pawl is pushed towards the notch disk and advances by one letter position. Normally, the notch disk is not in its notch position so the pawl will rest on the rim and slides without moving the rotor. The rim will also holds the
pawl clear of the notches on the ratchet disk on its left, so the middle rotor will not be moved. If, on the other hand, the right hand rotor is at its notch position, the pawl will fall into the notch and will also engage a notch in the ratchet disk of the middle rotor causing both rotors to advance. As the key is released the pawl will slide up the gentle slope of both notches and eventually be lifted clear of the both disks.

There are pawls positioned just to the right of each of the three rotors. The pawl between the left and middle rotors behaves just like the pawl between the middle and right hand rotors, but the pawl on the right of the right hand rotor will always engage its ratchet disk causing this to advance on every key stroke.

If the right hand rotor is in its notch position, the next key stroke will advance both the right hand and middle rotors. If the middle rotor is now in its notch position, the next key stroke will advance both the middle and left hand rotors. Notice that, in this situation, the middle rotor advances on two consecutive key strokes. You can observe this double stepping behaviour by selecting rotors III, II and I (#321) whose notch positions are V, E and Q, and setting the rotor positions to KDO (=KDO) before typing a few letters with tracing turned on.

In the above function, the variable \( \text{advM} \) is set to \( \text{TRUE} \) if the middle rotor advances on the current key stroke and similarly \( \text{advL} \) is \( \text{TRUE} \) if the left hand rotor advances at the same time. Notice that \( \text{advM} \) is \( \text{TRUE} \) if either \( \text{posR} \text{=} \text{notchR} \) or \( \text{posM} \text{=} \text{notchM} \), and \( \text{advL} \) is only \( \text{TRUE} \) if \( \text{posM} \text{=} \text{notchM} \). Rotors are advanced by adding one to their positions held in \( \text{posL} \), \( \text{posM} \) or \( \text{posR} \). The addition of \( \text{MOD 26} \) deals with the situation of a rotor advancing from its Z to A positions.

When no key is being pressed, the pawls are clear of the notch disks and the rotors can be rotated forward or backwards by hand.

AND encodestr() BE
{ // Set initial state
  posL, posM, posR := initposL, initposM, initposR
  // The rotor numbers and ring settings are already set up.
  IF len = 0 RETURN

  FOR i = 1 TO len DO
  { LET x = inchar!i - 'A' // letter to encode
    IF stepping DO step_rotors()
      outchar!i := enigmafn(x) + 'A'
    }
  TEST tracing
  THEN prsigpath()
  ELSE writef(" %c", plugoutB+'A')
  }

  This function causes the entire message in \text{inchar} to be encrypted, updating \text{outchar} appropriately. It does this by initialising \text{posL}, \text{posM} and \text{posR} to
initposL, initposM and initposR, then successively calling enigmanf giving it each character of the input message. If tracing is TRUE it then outputs a diagram showing the electrical path through the plug board, rotors and reflector used to encode the final character, otherwise it just outputs the final encrypted character.

The next two functions implement the encryption mechanism of the enigma machine, as you will see these functions are quite simple.

AND enigman(x) = VALOF
{
  // Plug board
  pluginF := x
  rotorRinF := plugboard!pluginF
  // Rotors right to left
  rotorMinF := rotorfn(rotorRinF, rotorFR, posR, ringR)
  rotorLinF := rotorfn(rotorMinF, rotorFM, posM, ringM)
  refin := rotorfn(rotorLinF, rotorFL, posL, ringL)
  // Reflector
  rotorLinB := reflector!refin
  // Rotors left to right
  rotorMinB := rotorfn(rotorLinB, rotorBL, posL, ringL)
  rotorRinB := rotorfn(rotorMinB, rotorBM, posM, ringM)
  pluginB := rotorfn(rotorRinB, rotorBR, posR, ringR)
  // Plugboard
  plugoutB := plugboard!pluginB

  RESULTIS plugoutB
}

The argument x is a number in the range 0 to 25 representing a letter position of an active signal within the machine. This signal must first pass through the plug board, emerging at position plugboard!x. So that the path through the machine of the active signal can be drawn, its position between components is saved in global variables such as pluginF and rotorRinF. Generally speaking F indicates a signal travelling in the forward direction (from right to left) and B indicates travel in the backwards direction (from left to right). The signal entering the right hand rotor in the forward direction is held in rotorRinF and it leaves this rotor in position rotorMinF. The computation is done by a call of rotorfn which takes four arguments giving the input position, the appropriate wiring vector, the position of the rotor and its ring setting. The function rotorfn is described below. The signal from the right hand rotor then passes through the middle rotor and the left hand rotor, emerging at position refin. The signal then re-enters the left hand rotor at position rotorLinB that was computed by the expression reflector!refin. The signal then passes back through the rotors via positions computed by three calls of rotorfn before re-entering the plug board.
at position \texttt{pluginB}. Since the plug board is its own inverse its effect can be computed using \texttt{plugboard!pluginB} to give \texttt{plugoutB} which is the position of the light identifying the encrypted letter. This position is returned as the result of \texttt{enigmafn}.

\begin{verbatim}
AND rotorfn(x, map, pos, ring) = VALOF
{ LET a = (x+pos-ring+26) MOD 26
  LET b = map!a
  LET c = (b-pos+ring+26) MOD 26
  RESULTIS c
}
\end{verbatim}

As explained above, each rotor has a wiring core that connects terminals on its right hand side to contacts contacts on the left. Each of the five available rotors have their own wiring specified by strings held in the variables \texttt{rotorI} to \texttt{rotorV}. When the rotors have been selected their wiring maps will have been placed in vectors such as \texttt{rotorFR} and \texttt{rotorBR}. Here, \texttt{rotorFR} gives the map specifying how the signal passes through the right hand rotor from right to left. If the wiring core has its \texttt{A} position aligned with the \texttt{A} position of the machine, then the signal will emerge at position \texttt{rotorFR!x} where \texttt{x} is the machine position of the signal entering the right hand rotor from the right. But the rotational position of the rotor depends on it position (\texttt{posR}) as displayed in the rotor’s little window, and on its ring setting. As the rotor steps forward from, for instance, \texttt{A} to \texttt{B}, its wiring core rotates anti-clockwise by one position when viewed from the right. So we should add \texttt{posR} to \texttt{x} before computing \texttt{rotorFR!x}. If the ring position is \texttt{B} rather than \texttt{A} the wiring core is effectively rotated clockwise when viewed from the right, and so we must subtract \texttt{ringR} from \texttt{x} before the lookup. To deal with the boundary between \texttt{Z} and \texttt{A} we must add \texttt{26} and the take the remainder after division by \texttt{26}. The addition of \texttt{26} ensures that the left hand operand of \texttt{MOD} is positive. The appropriate position within the map is thus \texttt{(x+pos-ring+26) MOD 26} which is placed in variable \texttt{a}. The result of the lookup is then placed in \texttt{b} by the declaration \texttt{LET b = map!a}. This gives a position relative to the \texttt{A} position of the wiring core. The corresponding position within the machine is \texttt{(b-pos+ring+26) MOD 26} which becomes the result of \texttt{rotorfn}. With suitable arguments this function can be used to compute the effect of each of the three rotors in both the forward and backward directions.

What remains are the functions that generate the ASCII graphics representation of the signal path showing how any given input letter generates the corresponding encrypted letter. Even though it now all looks fairly straightforward, it did take longer to design and implement than all of the rest of \texttt{enigma-m3.b}.

As can be seen in the wiring diagram in Figure 4.4 on page 92 it consists of several blocks placed side by side representing the reflector, the three rotors, the plug board and the keyboard/lights block. Each has edges drawn using vertical
bars (|) and minus signs (-) and separated from each other by three spaces. The signal path has a direction and is drawn using the characters <, >, ^, v. An asterisk (*) is used whenever the path turn a right angle.

The diagram contains 26 lines numbered 0 to 25 from bottom to top with the convention that line 13 corresponds to the A position within the machine. To improve readability some spacer lines consisting mainly of minus signs and vertical bars have been added. Each spacer line has the same line number as the letter line just above it. The diagram is drawn using prsigpath whose definition is as follows.

```plaintext
AND prsigpath() BE
{ newline()
    prsigline(26, TRUE)
    prsigline(25, FALSE)
    prsigline(24, FALSE)
    prsigline(23, FALSE)
    prsigline(22, FALSE)
    prsigline(22, TRUE)
    prsigline(21, FALSE)
    prsigline(20, FALSE)
    prsigline(19, FALSE)
    prsigline(18, FALSE)
    prsigline(18, TRUE)
    prsigline(17, FALSE)
    prsigline(16, FALSE)
    prsigline(15, FALSE)
    prsigline(14, FALSE)
    prsigline(14, TRUE)
    prsigline(13, FALSE)
    prsigline(13, TRUE)
    prsigline(12, FALSE)
    prsigline(11, FALSE)
    prsigline(10, FALSE)
    prsigline( 9, FALSE)
    prsigline( 9, TRUE)
    prsigline( 8, FALSE)
    prsigline( 7, FALSE)
    prsigline( 6, FALSE)
    prsigline( 5, FALSE)
    prsigline( 5, TRUE)
    prsigline( 4, FALSE)
    prsigline( 3, FALSE)
    prsigline( 2, FALSE)
    prsigline( 1, FALSE)
    prsigline( 0, FALSE)
}
```
Each line is drawn by calls of \texttt{prsigline} whose first argument is the line number, and whose second argument specifies whether or not it is a spacer line. The top and bottom space lines are drawn by the calls \texttt{prsigline(26,TRUE)} and \texttt{prsigline(0,TRUE)}. Below the bottom line, labels are written giving the names of the reflector, the rotors, the plug board and the keyboard. Below this there are two lines giving the message text and its encryption.

Each line in the wiring diagram contains characters representing a line through the reflector, the three rotors, the plug board and the keyboard/lights. These are drawn by calls of \texttt{prsigline} whose definition is as follows.

\begin{verbatim}
AND prsigline(n, sp) BE
  { prsigreflector(n, sp, reflin, rotlinB)
    prsigrotor(n, sp, posL, ringL, notchL, 
               rotlinF, reflin, rotlinB, rotminB)
    prsigrotor(n, sp, posM, ringM, notchM, 
               rotminF, rotlinF, rotminB, rotminB)
    prsigrotor(n, sp, posR, ringR, notchR, 
               rotminF, rotminF, rotminB, pluginB)
    prsigplug(n, sp, pluginF, rotminF, pluginB, plugoutB)
    prsigkbd(n, sp, pluginF, plugoutB)
    newline()
  }
\end{verbatim}

As can be seen, the parts of the line corresponding to the reflector, the rotors, the plug board and the keyboard are drawn using suitable calls of \texttt{prsigreflector}, \texttt{prsigrotor}, \texttt{prsigplug} and \texttt{prsigkbd}. The functions are defined below.

\begin{verbatim}
AND prsigreflector(n, sp, inF, outB) BE
  { LET iF = (inF +13) MOD 26
    LET oB = (outB +13) MOD 26
  }
\end{verbatim}
LET letter = (n+13) MOD 26 + 'A'
LET c0, c1, c2, c3 = '|', ' ', ' ', ' '
LET c4, c5, c6 = letter, '|', ' '

TEST sp
THEN { c1,c2,c3,c4 := '-', '-', '-', '-', '-',
    IF iF<n<=oB DO c2 := '^',
    IF iF>=n>oB DO c2 := 'v',
    IF n=0 | n=26 DO c0,c5 := ' ',''
}
ELSE { IF iF=n | oB=n DO c2 := '**',
    IF iF<n<oB DO c2 := '^',
    IF iF>n>oB DO c2 := 'v',
    IF iF=n DO c3,c6 := '<','<',
    IF oB=n DO c3,c6 := '>','>',
}
    writef("%c%c%c%c%c%c%c", c0,c1,c2,c3,c4,c5,c6,c6)
}

The arguments n and sp give the line number to be drawn and whether it is a spacer line or not, and inF and outB are in the range 0 to 25 representing A to Z, specifying the machine positions of the input and output signals to the reflector.

The declaration LET iF = (inF+13) MOD 26 converts the input signal position to a line number, and the declaration of oB does the same for the output signal. The declaration LET letter = (n+13) MOD 26 + 'A' converts the line number to the letter representing the machine position of the line. By convention line 13 corresponds to A.

The variables c0 to c6 will hold characters representing the line of the reflector to be drawn. Normally c0 and c5 hold vertical bars for the left and right edges of the reflector, c1 is normally a space and c2 is used to represent a wire joining the input and output signal positions. It is thus normally a space character or one of ^, v or *. Normally c3 and c6 hold spaces but can be set to < or > to represent a signal entering or leaving the reflector. The letter position within the machine is held in c4.

The TEST command then adjusts these settings mainly depending on whether a spacer line is being drawn and the relative positions of the line and the input and output positions. Finally, it outputs the characters using a writef statement, duplicating c6 for readability.

Drawing a line of a rotor is more complicated since it is necessary to draw signal wires for the forward and backward paths as well as showing the rotor and notch positions, and the ring setting. This is done by the function prsigrotor defined as follows.

AND prsigrotor(n, sp, pos, ring, notch,
inF, outF, inB, outB) BE
{ LET iF = (inF+13) MOD 26
LET iB = (inB+13) MOD 26
LET oF = (outF+13) MOD 26
LET oB = (outB+13) MOD 26
LET nch = (notch-pos+13+26) MOD 26
LET rng = (ring-pos+13+26) MOD 26
LET let1 = (n+pos+13+26) MOD 26 + 'A'
LET let2 = (n+pos-ring+13+26) MOD 26 + 'A'
LET c0,c1,c2,c3,c4,c5 = ' ','|',let1,'|',let2,' '
LET c6,c7,c8,c9 = ' ',let2,'|',' ' 
TEST sp
THEN { c2,c3,c4,c5,c6,c7 := '−','|','−','−','−','−','−','−'
        IF n=0 | n=26 DO c1,c3,c8 := ' ','−',' '
    }
ELSE { IF n=iF DO c6,c9 := '<','<'
        IF n=oB DO c6,c9 := '>','>'
        IF n=oF DO c0,c5 := '<','<'
        IF n=iB DO c0,c5 := '>','>
        IF n=nch DO c0 := '='
        IF n=rng DO c3 := '**'
        IF n=13 DO c1,c3 := ' [',']'
    }
writef("%c%c%c%c%c", c0,c1,c2,c3,c4,c5)
prsigwiring(n, sp, iF, oF, iB, oB)
writef("%c%c%c%c%c", c6,c7,c8,c9,c9) 
}

The forward and backward input and output positions are specified by the arguments inF, outF, inB and outB. These are in the range 0 to 25 representing A to Z. The declaration LET iF = (inF+13) MOD 26 converts inF to a line number in the wiring diagram, with the convention line 13 corresponds to A. The variable iB, oF, oB are similarly defined. The variable nch holds the line number corresponding to the position of the rotor’s notch, and rng is the line number corresponding to the A position of the rotor’s wiring core. The letter on the rotor’s ring corresponding to the current line is held in let1, and wiring core letter corresponding to the current line is held in let2.

The variables c0 to c9 will hold characters representing the current line in the rotor. The notch position is represented by an equal sign (=) in c0. If this is line 13 then the rotor is at its notch position and the next key press will advance the rotor on its left. Normally, c0 is not an equal sign it will hold a space unless a signal enters or leaves on this line, in which case it will hold either < or >. The rotor’s ring of letters has a letter in c2 normally surrounded by vertical bars in c1 and c3, but we are on line 13 it will be surrounded by square brackets to indicate
that the letter is in the rotor’s little window. If the letter corresponds to the ring setting, \(c_3\) holds and asterisk (\(*\)). The variables \(c_4\) and \(c_8\) normally hold \(let_2\), the letter on the wiring core corresponding to this line. The routing of the two wires in the wiring core occupies three character positions between \(c_5\) and \(c_6\). These are written by a call of \texttt{prsigwiring} which is defined below. The entry and exit positions are marked using < and > in \(c_5\) and \(c_6\). The right hand edge of the rotor is marked by a vertical bar in \(c_8\), and the signal entering or leaving the rotor on the right is marked by either < or > in \(c_9\), which is duplicated for readability.

The initial settings of these character variables are adjusted by the \texttt{TEST} command. For spacer lines the correction is simple, and for non spacer lines attention is paid to input and output positions of signals, the notch and ring positions and whether the ring letter is displayed in the rotor’s little window.

The plug board is similar to a rotor in that it requires the routing of two wires which may cross each other. This routing is again done using \texttt{prsigwiring}, otherwise dealing with the plugboard is simple. The definition of \texttt{prsigplug} is as follows.

\begin{verbatim}
AND prsigplug(n, sp, inF, outF, inB, outB) BE
{ LET iF = (inF +13) MOD 26
  LET oF = (outF+13) MOD 26
  LET iB = (inB +13) MOD 26
  LET oB = (outB+13) MOD 26

  LET letter = (n+13) MOD 26 +'A'
  LET c0,c1,c2,c3 = ' ','|', letter, ' '
  LET c4,c5,c6,c7 = ' ', letter, '|', ' '

  TEST sp
  THEN { c2,c3,c4,c5 := '-','-','-','-'
          IF n=0 | n=26 DO c1,c6,c7 := ' ',' ',' '
       }
  ELSE { IF n=iF DO c4,c7 := '<','<'
           IF n=oF DO c0,c3 := '<','<'
           IF n=iB DO c0,c3 := '>','>'
           IF n=oB DO c4,c7 := '>','>'
       }
  writeln("%c%c%c%c", c0,c1,c2,c3)
  prsigwiring(n, sp, iF,oF,iB,oB)
  writeln("%c%c%c%c%c%c", c4,c5,c6,c7,c7,c7)
}
\end{verbatim}

As with \texttt{prsigrotor}, the variables \(iF\), \(oF\), \(iB\) and \(oB\) are declared to give the line numbers of these signals. The edges are marked by vertical bars in \(c_1\) and
c6. The letter position is duplicated in c2 and c5. The entry and exit positions to the wiring is marked by < or > in c4 and c5. Much of the coding is similar to that used in prsigrotor.

Finally, the keyboard and lights are deal with prsigkbd whose definition is as follows.

```
AND prsigkbd(n, sp, inF, outB) BE
{ LET iF = (inF +13) MOD 26
  LET oB = (outB+13) MOD 26
  LET letter = (n+13) MOD 26 + 'A'
  LET c0,c1,c2 = '|',letter,'|'

  IF sp DO
  { c1 := '-'
    IF n=0 | n=26 DO c0,c2 := ' ',' '
  }

  writef("%c%c%c", c0,c1,c2)
  IF n=iF UNLESS sp DO { writef("<<%c", letter); RETURN }
  IF n=oB UNLESS sp DO { writef(">>%c", letter); RETURN }
}
```

This is particularly simple because it just outputs the machine letter positions surrounded by vertical bars, and marks which key was pressed and which encrypted letter was generated by writing strings such as <<Q and >>D to the right of the keyboard.

The routing of wires in the rotors and the plug board is done by prsigwiring. It is quite long since there are many separate cases to deal with. It definition starts as follows.

```
AND prsigwiring(n, sp, iF, oF, iB, oB) BE
{ // iF, oF, iB and oB are in the range 0 to 25 representing
  // line numbers within the wiring diagram of the forward and
  // backward input and output signals.

  LET Flo,Fhi,Blo,Bhi = iF,oF,iB,oB
  LET aF, aB = '^','^'
  LET c1,c2,c3=' ',' ',' '

  IF iF>oF DO Flo,Fhi,aF := oF,iF,'v'
  IF iB>oB DO Blo,Bhi,aB := oB,iB,'v'
  // aF and aB = ` or v giving the vertical direction
  // for the forward and backward paths.
```
/* n = the line number in range 0 to 26 */
/* with the convention n=13 corresponds to position A */
/* sp = TRUE for spacer lines */
/* c1, c2 and c3 are for the three wiring characters */
/* for this line. */

The arguments n and sp specify the line number and whether the line is a spacer. The remaining arguments iF, oF, iB and oB give the line numbers of the forward and backward entry and exit positions. The variables Flo, Fhi, Blo and Bhi are declared and initialised to the smaller and larger values of iF, oF, iB and oB, and aF and aB are declared and initialised to hold ^ and v to indicate the vertical direction of the forward and backward wires. These are used in many places in the code that follows.

The variables c1, c2 and c3 will hold the routing of the signals, if any, through the current line. There are many cases to consider and these will be taken in turn.

IF sp DO
{
    // Find every spacer line containing no wires.
    IF n>Fhi & n>Bhi | n=Flo & n=Blo | Bhi<n=Flo | Fhi<n=Blo DO
    { writef("---") // Draw a spacer line with no wires.
        RETURN
    }
    c1,c2,c3 := '-','-','-'
}

This tests to see if the current line is a spacer line containing no wires, and if so just outputs three minus signs (---). A spacer line that does contain wires has the default setting of c1 to c3 changed from spaces to minus signs.

// Find all non spacer lines containing no wires.
IF n>Fhi & n>Bhi | n<Flo & n<Blo | Bhi<n<Flo | Fhi<n<Blo DO
{ // Non spacer line at position n contains no wires.
    writef(" ")
    RETURN
}

This code deals with non spacer lines containing no wires by simply outputting three spaces and returning from prsigwiring.
From now on we know there is at least one signal wire passing through this line.

\[
\text{IF } \text{Flo} > \text{Bhi} \mid \\
\text{Blo} > \text{Phi} \text{ DO} \\
\{ \text{ // There is only one wire at this region so} \\
\text{ // the middle column can be used.} \\
\text{UNLESS } \text{sp} \text{ DO} \\
\{ \text{ IF } \text{iF} = \text{oF} \text{ DO } \{ \text{ writef("<<<"); RETURN } \} \\
\text{ IF } \text{iB} = \text{oB} \text{ DO } \{ \text{ writef(">>>"); RETURN } \} \\
\text{ // Position } n \text{ has an up or down going wire.} \\
\text{ IF } n = \text{iF} \text{ DO } \{ \text{ writef(" **<"); RETURN } \} \\
\text{ IF } n = \text{oF} \text{ DO } \{ \text{ writef("<** "); RETURN } \} \\
\text{ IF } n = \text{iB} \text{ DO } \{ \text{ writef(">** "); RETURN } \} \\
\text{ IF } n = \text{oB} \text{ DO } \{ \text{ writef(" **>"); RETURN } \} \\
\} \\
\text{ IF } \text{Flo} < n \leq \text{Fhi} \text{ DO } c2 := aF \\
\text{ IF } \text{Blo} < n \leq \text{Bhi} \text{ DO } c2 := aB \\
\text{ writef("%c%c%c", c1, c2, c3)} \\
\text{ RETURN}
\}
\]

We now know there is at least one wire passing through this line, so we test for the special case of the forward wire being entirely above or entirely below the backward wire. If this happens both wires can be routed along the middle column, namely \( c_2 \). We must deal with signals that enter or leave on this line, and we must also check whether the signal both enters and leaves on this line, necessitating \( <<< \) or \( >>> \). The general case is to conditionally plant the appropriate vertical arrow in \( c_2 \).

\[
\text{IF } \text{iB} < \text{oF} < \text{iF} \mid \\
\text{oB} < \text{iF} \text{ AND } \text{iF} < \text{oF} < \text{iB} \text{ DO} \\
\{ \text{ TEST } \text{sp} \\
\text{ THEN } \{ \text{ // This is a spacer line} \\
\text{ // so only contains vertical wires} \\
\text{ IF } \text{Flo} < n \leq \text{Fhi} \text{ DO } c1 := aF \\
\text{ IF } \text{Blo} < n \leq \text{Bhi} \text{ DO } c3 := aB \\
\} \\
\text{ ELSE } \{ \text{ // This is a non spacer line} \\
\text{ IF } n = \text{iF} \text{ DO } c1, c2, c3 := '***', '<', '<' \\
\text{ IF } n = \text{oF} \text{ DO } c1 := '***' \\
\text{ IF } n = \text{iB} \text{ DO } c1, c2, c3 := '>', '>', '***' \\
\text{ IF } n = \text{oB} \text{ DO } c3 := '***' \\
\}
\]
IF Flo<n<Fhi DO c1 := aF
IF Blo<n<Bhi DO c3 := aB
}
writef("%c%c%c", c1,c2,c3)
RETURN
}

This tests whether the forward wire can be placed on the left and drawn without the two wires crossing. If so, the vertical portion of the forward wire is placed in c1, and c3 is used by the backward wire. Again, there are special cases if any signal enters or leaves at this line position.

IF oB<iF<oF & iB<oF |
oF<iB & oF<iF<oB DO
{ TEST sp
  THEN { // This is a spacer line
    // so only contains vertical wires
    IF Flo<n<=Fhi DO c3 := aF
    IF Blo<n<=Bhi DO c1 := aB
  }
  ELSE { // This is a non spacer line
    IF n=oF DO c1,c2,c3 := '<','<','**'
    IF n=iF DO c3 := '**'
    IF n=oB DO c1,c2,c3 := '**','>','>'
    IF n=iB DO c1 := '**'
    IF Flo<n<Fhi DO c3 := aF
    IF Blo<n<Bhi DO c1 := aB
  }
writef("%c%c%c", c1,c2,c3)
RETURN
}

This case is the mirror image of the previous one and routes the forward wire on the right hand side in c3.

We now know there are two wires that cannot be drawn without crossing.

IF iF=oF DO
{ c2 := aB
  TEST sp
  THEN { IF n=Blo DO c2 := '-'
  }
  ELSE { IF n=iF DO c1,c3 := '<','<'
    IF n=iB DO c1,c2 := '>','**'
    IF n=oB DO c2,c3 := '**','>'
  }

This tests whether the forward wire can be placed on the left and drawn without the two wires crossing. If so, the vertical portion of the forward wire is placed in c1, and c3 is used by the backward wire. Again, there are special cases if any signal enters or leaves at this line position.

IF oB<iF<oF & iB<oF |
oF<iB & oF<iF<oB DO
{ TEST sp
  THEN { // This is a spacer line
    // so only contains vertical wires
    IF Flo<n<=Fhi DO c3 := aF
    IF Blo<n<=Bhi DO c1 := aB
  }
  ELSE { // This is a non spacer line
    IF n=oF DO c1,c2,c3 := '<','<','**'
    IF n=iF DO c3 := '**'
    IF n=oB DO c1,c2,c3 := '**','>','>'
    IF n=iB DO c1 := '**'
    IF Flo<n<Fhi DO c3 := aF
    IF Blo<n<Bhi DO c1 := aB
  }
writef("%c%c%c", c1,c2,c3)
RETURN
}
writef("%c%c%c", c1,c2,c3)
RETURN
}

This code tests whether the backward wire can use the centre column with
the forward wire passing straight through it.

IF iB=oB DO
{ // The F wire can use the centre column.
c2 := aF
TEST sp THEN { IF n=Flo DO c2 := '-'
}
ELSE { IF n=iB DO c1,c3 := '>','>'
IF n=oF DO c1,c2 := '<','**'
IF n=iF DO c2,c3 := '**','<'
}
writef("%c%c%c", c1,c2,c3)
RETURN
}

This is the mirror image of the previous situation. It places the forward wire
in the centre c2 and lets the backward wire pass straight through it.

// Test whether the F and B signals enter at the
// same level, and leave at the same level.
// Note that iF cannot equal oB,
// and iB cannot equal oF.
IF iF=iB &
oF=oB TEST Fhi-Flo<=2
THEN { // No room for a cross over
TEST sp THEN { IF n>iF | n>oF DO c2 := '|' 
}
ELSE { IF Flo<n<Fhi DO c2 := '|' 
IF n=iF DO c1,c2,c3 := '>', '**','<'
IF n=oF DO c1,c2,c3 := '<', '**','>'
}
writef("%c%c%c", c1,c2,c3)
RETURN
}
ELSE { // The gap between iF and oF is more than 1 line
// so the F wire can use the centre column and
// the B wire can cross it half way down.
LET m = (iF+oF)/2
// Place the F wire down the centre.
c2 := aF
IF n=iF DO c2,c3 := '**','<'
IF n=oF DO c1,c2 := '<','**'
// Now place the B wire, crossing half way down.
TEST iB>oB
THEN { IF n>m DO c1 := aB
      IF n=m DO c3 := aB
    }
ELSE { IF n>m DO c3 := aB
       IF n=m DO c1 := aB
    }
UNLESS sp DO
{ IF n=iB DO c1 := '**'
  IF n=oB DO c3 := '**'
  IF n=m DO c1,c2,c3 := '**','>','**'
}
writef("%c%c%c", c1,c2,c3)
RETURN

This code deal with the special case of both signal entering on the same line and leaving on the same line. Somehow they must be made to cross but there may not be room. If this happens we resort to patterns such as the following.

>**< or >**<

|<*> |

|<**>

But if there is room, we can place one wire along the centre c2 and let the other wire pass cross half way down.

IF Flo<iB<Fhi | Blo<iF<Bhi DO
{ // The F wire can be on the left.
  IF Flo<=n<=Fhi DO c1 := aF
  IF Blo<=n<=Bhi DO c3 := aB

  UNLESS sp DO
  { IF n=iF DO c1,c2,c3 := '**','<','<'
    IF n=iB DO c2,c3 := '>','**'
    IF n=oF DO c1 := '**'
    IF n=oB DO c3 := '**'

If Flo<iB<Fhi | Blo<iF<Bhi DO
4.19. **BREAKING THE ENIGMA CODE**

The Enigma machine was beautifully engineered, reliable and easy to maintain. It had an incredibly large number of possible settings most generating completely different encryptions.

There were two reflectors to choose from and \( 5 \times 4 \times 3 = 60 \) possible selections of three rotors from the five available. There were \( 26 \times 26 \times 26 = 17576 \) possible initial rotor core positions. The \( 26 \times 26 = 676 \) ring settings of the middle and right hand rotors affected the encryption, but since the middle rotor typically
only steps once every 26 characters and the left hand rotor almost never steps, the difficulty of finding a compatible ring setting is considerably reduced.

The main complication is finding the plugboard’s setting. There were ten cables each causing two letters to swap. There were thus six letters that pass straight through the plugboard unchanged. We first calculate how many ways we can select six letters from an alphabet of 26 letters. Mathematicians have no difficulty with this and instantly give the answer \( \frac{26!}{6! \times 20!} \), which is known as a binomial coefficient, often written as \( C_{26}^6 \). This turns out to be the coefficient of \( x^6 \) in the expansion of \((1 + x)^{26}\). If we have no knowledge of binomials, we can derive this formula from first principles as follows. Consider all the permutations of 26 letters. For any particular permutation, the first letter will be any one of the 26 letters, the second will be any one of the remaining 25, the third will be one of 24, and so on. This tells us that the number of permutations of 26 letters is \( 26 \times 25 \times 24 \times \ldots \times 1 \) which is known as 26 factorial and is normally written as 26!. If we now look at the first six letters these permutations, we will find it contains all possible selections of six letters from the alphabet but repeated many times over. We should divide by 6!, the number of permutations of six letters, and by 20! the number of permutations of the remaining 20 letters that were not selected. This gives the answer \( \frac{26!}{6! \times 20!} \), which can be written as \( \frac{26 	imes 25 	imes 24 	imes 23 	imes 22 	imes 21}{6 	imes 5 	imes 4 	imes 3 	imes 2 	imes 1} \). This can be simplified by observing \( 22/2 = 11, 21/3 = 7, 24/(6 \times 4) = 1 \) and \( 25/5 = 5 \). So the result is \( 25 \times 5 \times 23 \times 11 \times 7 = 230230 \), which is the number of ways of choosing the six letters that pass straight through the plugboard. The remaining 20 letters are paired up by the ten cables. First sort the 20 letters in alphabetical order, then select the left most letter and pair it with any one of the remaining 19 letters. Then select the leftmost letter that has not yet been paired and pair it with one of the remaining 17 letters. The next pairings have choices of 15, 13, etc. The total number of ways the pairing that can be done is thus \( 19 \times 17 \times 15 \times 13 \times 11 \times 9 \times 7 \times 5 \times 3 \times 1 = 654729075 \), and so the total number of way the plugboard can be set is thus \( 230230 \times 654729075 = 150739274937250 \) which is slightly more than 150 million million. If we multiply this by the number of ways the rotors can be set up we get a staggeringly large number in the region of \( 10^{23} \). This large number provided convincing evidence the enigma code was unbreakable, and the Germans relied on this belief throughout the war.

However, Alan Turing and others at Bletchley Park discovered a weakness in the code and designed a largely mechanical machine called the bombe to help decode Enigma messages. This section outlines a program (bcplprogs/raspi/bombe.b) that uses some of the principles used in the bombe. There is not space here to describe the program in detail. This section just gives an outline some of the principles used.

The method relies on having a crib consisting of some plain text and its encryption. Such cribs are obtained by guessing some likely plain text and matching it with all encrypted messages transmitted on that day. If the plain text is long enough most alignments of the plain text with encrypted text will be thrown
out by the rule that no letter encrypts to itself. In the program, a crib is used consisting of the first 29 letters of the message given in the previous section and its encryption. This choice has the advantage we know the answer and its long length means a solution can be found reasonably quickly. The decryption breakthrough came as a result of discovering a way of deducing the plugboard setting from the crib.

The program uses the first 29 letters of the message and its encryption shown below.

```
1  6  11  16  21  26  31  36  41
QBTLW LDAHH YEOEF PTWYB LENDP MKOXL DFAMU DWIJD XRJZ
DERFU EHRER ISTTO DXDER KAMPF GEHTW EITER XDOEN ITZX
```

It first converts the crib into what mathematicians like to call a graph consisting of 26 letter nodes joined by edges labelled with integers. The numbers are positions within the crib. For instance there is an edge labelled 1 joining node Q to node D, corresponding to the first position in the crib. As a debugging aid, the program outputs the graph as shown below. Notice that the line starting Q: has an edge 1D and that the line starting D: has and edge 1Q.

```
A: E 27  22E  8R
B: E 27  20R  2E
C: C  0
D: E 27  24P  18W  16P  7H  1Q
E: E 27  27K  22A  19Y  14T  12S  9H  6L  2B
F: E 27  25P  15O  4T
G: M  2  26M
H: E 27  28O  10R  9E  7D
I: E 27  11Y
J: J  0
K: E 27  27E  21L
L: E 27  21K  6E  3R
M: M  2  26G  23N
N: M  2  23M
O: E 27  28H  15F  13T
P: E 27  25F  24D  16D
Q: E 27  1D
R: E 27  20B  10H  8A  3L
S: E 27  12E
T: E 27  29X  17X  14E  130  4F
U: E 27  5W
V: V  0
W: E 27  18D  5U
X: E 27  29T  17T
```
This graph allows us to generate a series of tests to see if a particular initial setting of the Enigma machine is consistent with the crib. The beauty of this mechanism is that we do not have to guess the wiring of the plugboard since it can be deduced as the tests are performed. We do, however, have to guess which reflector is used, which rotors have been selected for the left, middle and right hand positions. We also have to guess the rotational positions of the rotors and the notch positions of the middle and right hand rotors. Once these have been chosen, we can deduce the rotational position of each rotor for each position of the crib.

If the `bombe` program is called with the `-t` option, it generates the following trace output, and stops with an `ABORT 1000`, allowing the user to resume execution of the program using the `c` debugging command. A summary of other debugging commands can be seen by typing a question mark (`?`).

Testing reflector B rotors I II V notches QEZ
Trying posL=A
turnpattern=1 nr=0
 1: ABB lgqpxboymuwartbdcmvnksjifie
### 4.19. **BREAKING THE ENIGMA CODE**

| 2: | ABC | lqxkuyjvngdasitrbpmoehzcfw |
| 3: | ABD | pfvzedbjnklwyastqrcmuog |
| 4: | ABE | zorkxvgwmlshbhcncayqfjeta |
| 5: | ABF | mqxediyofpzvwjtburslncgk |
| 6: | ABG | zhdcrykbnqwjutieqposlmfa |
| 7: | ABH | smhfjdqrewxtvqziyanoklup |
| 8: | ABI | wqhzfctclvmyxuwsrgjaond |
| 9: | ABJ | zgluribjfhvpsntoeqdkmyxa |
| 10: | ABK | luspoimvzqagtedkycnbhxwrj |
| 11: | ABL | hwuipsaeonzckjfxtgdybqv |
| 12: | ABM | lmyqoqghwpbvejfcuksinizx |
| 13: | ABN | volwpzrkjchczqbyutadfsm |
| 14: | ABO | hkwecjoeapbqrunyvmxqntasl |
| 15: | ABR | rleucytmahbixjvwgaqpsnfq |
| 16: | ABQ | gcdwialftzthbpynxqrvueqoj |
| 17: | ABR | dtwamzpyvkjnelrgxoucsibqh |
| 18: | ABS | jifwqcvobamkxuhterspengdlzy |
| 19: | ABD | ncakiuylewdsbrzomxpfjtgq |
| 20: | ABQ | siwznpmboqrejflkaxvuctgd |
| 21: | ABV | ieyobpqjahusxvdfgwlzknrmct |
| 22: | ABO | duragqewmsvjjonyfeciwbkhlpt |
| 23: | ABS | guewcxaroismpknvhybqjdjtf |
| 24: | ABR | gtjwlkaoxcferyhunsqnpzdlimv |
| 25: | ABL | xisstrnfykjboqmpedwzaiv |
| 26: | ABA | uwkloxxrncdyepqghvatbfmj |
| 27: | ACB | rlhzzxocnsebgiumajypwftd |
| 28: | ACC | hmunxjgaygbkxlswqcrteif |
| 29: | ACD | jkmoqligabfcydhevzwxtums |

**Guess D -- trying inner=a**

```
!! ABORT 1000: Unknown fault
*
```

This shows that the program has selected reflector B and rotors I, II and V. The setting of `turnpattern=1` causes the notch position of the middle rotor to be such that the left hand rotor remains in the same rotation position for all 29 letters of the crib. The variable `nr` which is in the range 0 to 25 specified the initial position of the right hand rotor’s notch. Since `nr` is set to 0, as the initial letter of the crib is pressed the right hand and middle rotors both advance to position B. So at message position 1, the rotors have stepped to ABB. Notice that the rotors step from ABA to ABC between message position 26 and 27, as expected, and notice also that the left hand rotor remains at position A throughout the crib.

The sequence of letters `lgqqzbxoyuwrhdcmvnsjifie` associated with rotor positions ABB shows that a signal entering the right hand rotor at position a will
return to position 1 after passing through the rotors to the reflector and back to
the right hand rotor. Similarly b maps to g, and c maps to q. These mappings
are sometimes written as a1l, b1g, c1q, etc.

By convention, lower case letters, called inner letters, are used for positions of
signals between the plugboard and the right hand rotor. Upper case letters, called
outer letters, represent positions on the keyboard or lamp side of the plugboard.
Thus Q1D shows the mapping of key Q to lamp D when the rotors are in position
1.

If we look carefully at the graph, we see that, at position 16, pressing D gen-
erates P, and at position 24 pressing P generates D. The beauty of this observation
is that we can try all the 26 possible inner letters that the plugboard might map
outer letter D into. Most, if not all, of these will instantly lead to inconsistencies.
Suppose we try mapping D to b using the program’s choice of initial settings. At
message position 16 we have b16m, and so, if our assumptions are correct, plug-
board m must map to outer letter P. If we now consider the edge P24D, the inner
letter for P is already known to be m, and at position 24 there is the mapping
m24y implying that the inner letter for D should be y. But it has already been
assigned inner letter b. So either mapping D to b is wrong or the initial settings
are wrong. In either case we must backtrack.

The sequence of tests the program does can be represented by the following
list of statements.

guess D
edge D 16 P
dge P 24 D
dge D 7 H
dge H 9 E
dge E 14 T
dge T 4 F
dge F 25 P
dge T 13 O
dge O 15 F
dge H 28 O
dge T 17 X
dge T 29 X
dge H 10 R
dge E 2 B
dge R 20 B
dge R 3 L
dge E 6 L
dge R 8 A
dge A 22 E
dge L 21 E
dge E 27 K
The `guess` statements tries all possible plugboard mappings for its given outer letter, and the `edge` statements tests edges and the `fin` statement indicates that all edges have been tested.

We can see the effect of these statements by running the `bombe` with the `-t` option and stepping through the execution by typing `c` after each `ABORT 1000`. The effect of the first two choices `guess` makes is shown as follows (by typing `c` twice).

```
Guess D -- trying inner=a

!! ABORT 1000: Unknown fault
* c
  Guess setting plugboard D to a
  Guess setting plugboard A to d
edge D 16 P
  a16g
  Plugboard P and G are both unset, so
  Edge setting plugboard P to g
  Edge setting plugboard G to p
edge P 24 D
  g24a
  Plugboard D is already a, which is OK
edge D 7 H
  a7s
  Plugboard H and S are both unset, so
  Edge setting plugboard H to s
  Edge setting plugboard S to h
edge H 9 E
  s9o
  Plugboard E and O are both unset, so
  Edge setting plugboard E to o
  Edge setting plugboard O to e
edge E 14 T
  o14g
```
Plugboard G is already set to p, so cannot set G to t -- Backtrack
Edge unsetting plugboard E
Edge unsetting plugboard O
Edge unsetting plugboard H
Edge unsetting plugboard S
Edge unsetting plugboard P
Edge unsetting plugboard G
Guess unsetting plugboars D
Guess unsetting plugboard A

Guess D -- trying inner=b

!! ABORT 1000: Unknown fault
* c
  Guess setting pluboard D to b
  Guess setting plugboard B to d
edge D 16 P
  b16m
    Plugboard P and M are both unset, so
    Edge setting plugboard P to m
    Edge setting plugboard M to p
edge P 24 D
  m24y
    Plugboard D is already set to b, so cannot be set D to y -- Backtrack
Edge unsetting plugboard P
Edge unsetting plugboard M
Guess unsetting plugboars D
Guess unsetting plugboard B

Guess D -- trying inner=c

!! ABORT 1000: Unknown fault
*

The sequence of statements is compiled by the function trans which first constructs the graph using structures to represent letter nodes and edges.

A letter node is represented by a small vector whose fields are accessed by the selectors: n_parent, n_letter, n_list, n_len, n_size, n_visited and n_compiled. The parent field is either zero or points to another letter node. It provides a cunning mechanism to determine whether there is a path of edges connecting two nodes. If there is such a path the two nodes are said to be in the same connected component. The mechanism will be described later. The letter field holds a number in the range 0 to 25 specifying the outer letter this node represents. The list field holds the list of edges belonging to this node, and the len field holds the length of this list. If the parent field is zero, the node is
called a root, and the size field holds the total number of edges reachable from this root node. This is a measure of the complexity of the connected component this root node belongs to. The fields visited and compiled are used by the program that translates the graph into interpretive code. The field compiled is set to TRUE for all nodes in a connected component when all its the edges have been compiled.

An edge is represented by a vector whose fields are accessed by the selectors: e.next, e.pos and e.dest. The next field points to the next edge node in the list. The pos field holds the position in the crib corresponding to this edge and the dest field points to the destination node of this edge.

The vector nodetab whose subscripts range from 0 to 25 representing the letter A to Z has elements that point to the 26 letter nodes. Initially all the fields of each node are set to zero, except for its letter field which is set appropriately.

Edges are now added to the graph one at a time, the first being from S to D at position 1. This involves adding appropriate edge nodes to the lists belonging to the nodes for S and D. The len fields are incremented. The parent of any node provides a path to the root node of the connected component that the node belongs to. The root nodes for S and D are currently different so this edge joins the two previously disconnected components. This is implemented by choosing one of them to become the root of the combined component and setting the parent field of the other to point this new root. The sizes of the two components are summed and placed in the new root, and its value incremented because a new edge has just been added. When finding the root, it is often a good strategy to update all the parent links in the path to the root by direct links to the root since this typically makes later searches more efficient. Additionally, when combining two components, a good strategy is to make the root of the larger component the root of the combined component. These optimisations are important in applications involving millions of nodes. But in this program, they are not needed, and have only been done for educational reasons.

Once the graph has been constructed, the program compiles it into a sequence of the interpretive instructions. The interpretive code as shown above has instructions with only three function codes: c_guess, c_edge and c_fin.

The function code guess takes an outer letter argument and invites the interpreter to try all 26 possible plugboard mappings for this letter.

The function code edge takes three arguments representing the source letter, the message position of the edge, and the destination letter. The source letter refers to a node that has already been visited and so already has an inner letter assigned. The destination node may or may not have an inner letter assigned. If it has, it is checked for consistency, usually causing the program to backtrack. If the destination has no inner letter assigned, it is given the required letter and the plugboard is updated appropriately. Note that if, for instance, W is to be mapped to g, then G must also be mapped to w. This second mapping may be found to be inconsistent again causing the program to backtrack, but if not, the
unvisited node for \( G \) will be given inner letter \( w \) increasing the chance of finding an inconsistency later.

The function code \texttt{fin} indicates that all edges of the graph have been checked and no inconsistencies have been found, so the current initial setting may be correct and should be checked. This function code outputs the current initial setting then backtracks so that other possible solutions can be found.

The translation into interpretive code is done with care to attempt to increase the efficiency of the tests. The graph is searched for a good starting node and, once chosen, it generates an appropriate \texttt{guess} instruction. The starting node will belong to a connected component of largest size, and will, if possible, be in a loop of length two. If no such loop exists, a node with the largest number of edges will be chosen. The edges of the connected component are then explored generating an \texttt{edge} instruction each time. As the compilation proceeds, nodes that have been visited and edges that have been used are marked as such.

The strategy used to select the next edge to compile is as follows. First choose an unused edge connecting two visited nodes. If no such edge is found, choose an unused edge from a visited node to a node that has a different edge back to a visited node. If no such edge exists, choose an edge from a visited node to a node having the largest number of edges. When all the edges of the component have been compiled, the \texttt{compiled} field of every node in the connected component is set to \texttt{TRUE}, causing them to play no further part in the compilation. If there are any unused edges left, the whole process is repeated, ignoring all nodes marked as compiled. The \texttt{fin} instruction is compiled when all edges have been compiled. Notice that nodes that have no edges correspond to letters that do not occur in the plain or encrypted text of the crib. After compiling the graph the resulting interpretive code is output.

The final part of the program successively selects the reflector, the three rotors, their initial core positions, the message position (0 to 25) of the first step of the middle rotor and a code (1 to 5) specifying if and when the left hand rotor steps and if and when the middle rotor does a double step. Having given this specification of the machine setting the interpretive code is executes to see if the setting is compatible with the crib. It will almost always find an incompatibility quickly and backtracks to test the next setting.

The \texttt{bombe} program can be compiled into native machine code and run by typing:

```
cd ..../natbcpl
make -f MakefileRaspiSDL clean
make -f MakefileRaspiSDL bombe
./bombe
```

I ran it on my Pentium based laptop (replacing \texttt{MakefileRaspiSDL} by \texttt{MakefileSDL}) and found it took 3 minutes 28 seconds to find the solution, trying
all possible rotor selections but only using reflector B. On a 256Mb Raspberry Pi, it takes about 29 minutes. This slow speed is probably because my program uses much more memory than it really needs.

### 4.20 The Advanced Encryption Standard

Having just studied how the Enigma machine was used to encrypt messages, it is perhaps appropriate to see how encryption is done on modern computers. The Advanced Encryption Standard (AES) supercedes the previous Data Encryption Standard (DES) that was published in 1977. DES used a key length of 56 bits which is now thought insufficiently secure considering the enormous power of modern computers. AES is now a well established replacement. It was announced by the U.S. National Institute of Standards and Technology (NIST) in 2001 after a five year standardisation process in which many rival systems were compared. The clear winner was a scheme developed by two Belgian cryptographers, Joan Daemen and Vincent Rijmen. It is normally called AES128, AES192 or AES256 depending on the key length being used. The scheme is elegant and cunning allowing encryption to be done efficiently on simple hardware such as smart cards as well as normal computers, and it is well worth studying.

This section presents a demonstration implementation (aes128.b) of the version using 128 bit keys. The program starts as follows.

```plaintext
GET "libhdr"

GLOBAL {
    Rkey: ug
    sbox
    rsbox
    mul
    tracing
    MixColumns_ts
    InvMixColumns_st
    Cipher
    InvCipher
    prstate
    prbytes
    prmat

    // The s state matrix
    s00; s01; s02; s03
    s10; s11; s12; s13
    s20; s21; s22; s23
    s30; s31; s32; s33
```
The algorithm performs a sequence of transformations of a 4 by 4 matrix of 8-bit bytes. This matrix is called the state and, for convenience, is held either in the variables \( s_{00} \) to \( s_{33} \) or \( t_{00} \) to \( t_{33} \). The key is 128 bits long represented by a vector of \( \text{Keylen} = 16 \) bytes. This key is expanded by the function \( \text{KeyExpand} \), described below, to form a schedule of 11 keys in \( \text{Rkey} \) used during the encryption process. The data to be encrypted is broken into 128-bit chunks, placed in turn in the 16 bytes of the state matrix where the encryption process takes place. This consists of a sequence of ten repeated rounds of simple matrix transformations. All these transformations are reversible, so performing the inverse versions in reverse order can be used to decrypt the encrypted message.

One such matrix transformation is performed by the function \( \text{ShiftRows} \) defined below.

```plaintext
// The \text{ShiftRows}() function shifts the rows in the state to the left.
// Each row is shifted with different offset.
// Offset = Row number. So the first row is not shifted.
LET ShiftRows_st() BE
{ t00, t01, t02, t03 := s00, s01, s02, s03
  t10, t11, t12, t13 := s11, s12, s13, s10
  t20, t21, t22, t23 := s22, s23, s20, s21
  t30, t31, t32, t33 := s33, s30, s31, s32
}
LET InvShiftRows_ts() BE
{ s00, s01, s02, s03 := t00, t01, t02, t03
  s10, s11, s12, s13 := t13, t10, t11, t12
  s20, s21, s22, s23 := t22, t23, t20, t21
  s30, s31, s32, s33 := t31, t32, t33, t30
}
```
Another matrix transformation is performed by the function \texttt{SubBytes\_ts}, defined as follows.

\begin{verbatim}
LET SubBytes\_ts() BE 
{ // Apply sbox from t state to s state
  FOR i = 0 TO 15 DO stateS!i := sbox%(stateT!i)
}
\end{verbatim}

This uses the byte vector \texttt{sbox}, which specifies a permutation of the numbers 0 to 255, to convert bytes in state \texttt{t} to bytes in state \texttt{s}. Since a permutation is being used, the effect of \texttt{SubBytes\_ts} can be reversed by the function \texttt{InvSubBytes\_st}, defined as follows.

\begin{verbatim}
LET InvSubBytes\_st() BE
{ // Apply rsbox from s state to t state
  FOR i = 0 TO 15 DO stateT!i := rsbox%(stateS!i)
}
\end{verbatim}

This uses the byte vector \texttt{rsbox} representing the inverse of \texttt{sbox}. That is \(\texttt{rsbox}%(\texttt{sbox}%(x)) = x\) for all \(x\) in the range 0 to 255. These permutation vectors are defined by the function \texttt{inittables} as follows.

\begin{verbatim}
LET inittables() BE 
{ sbox := TABLE
  #x7B777C63, #xC56F6BF2, #x2B670130, #x76ABD7FE,
  #x7D982CA, #xF04759FA, #AFA2D4AD, #xC072A49C,
  #x2693DB7, #xC7F3F36, #xF1E5A534, #x1531D871,
  #xC323C704, #x9A059618, #xE2801207, #x75B227EB,
  #x1A2C8309, #xA05A6E1B, #xB3D63B52, #x842FE329,
  #xED00D153, #x5BB1FC20, #x9BE6CB6A, #xCF584C4A,
  #xFBAACED0, #x85334D34, #x7F02F945, #x1089F3C50,
  #x8F40A351, #x5389D92, #x21DAB6BC, #xD2F3FF10,
  #xEC130CCD, #x1744975F, #x3D7EA7C4, #x73195D64,
  #xD4F8160, #x88902A22, #xB8EE46, #xDB0B5EDE,
  #x0A3A32E0, #x5C240649, #x62ACD3C2, #x79E49591,
  #xD37C8E7, #xA94ED58D, #x4A4566C, #x08AE7A65,
  #x2E2578BA, #xC664A61C, #xF74DE8, #x8A8BB4DB,
  #x66B5E70, #x0EF60348, #x9B573561, #x9E1DC186,
  #x1198F8E1, #x948E969, #xE987E9B, #xDF2B55CE,
  #x0D89A18C, #x684E6BF, #x0F2D9941, #x16BB54B0

rsbox := TABLE
  #xD56A0952, #x38A53630, #x9EA340BF, #xFBD7F381,
  #x8239E7C, #x87FF2F9B, #x4443E34, #xCBE9DEC4,
\end{verbatim}
These TABLEs assume that BCPL is running on a, so called, little ended 32 bit version of BCPL such as that used on the Raspberry Pi and Pentium based machines. Notice that, for instance, $sbox\%0 = \#x63$ and $sbox\%1 = \#x7C$.

The next function AddRoundKey_st applies a specified round key from the schedule to the state matrix.

\[
\text{LET} \text{AddRoundKey} \text{st}(i) \text{ BE} \\
\{ // \text{ Add key round } i \text{ from } s \text{ state to } t \text{ state} \\
\quad \text{LET} \ K = @Rkey!(16*i) \quad // \text{ n = number of elements per row} \\
\quad \text{FOR} \ i = 0 \ \text{TO} \ 15 \ \text{DO} \ \text{stateT}!i := \text{stateS}!i \text{ XOR } K!i \\
\}\]

The vector $Rkey$ holds a schedule of round keys numbered from 0 to 10. Each round key consists of 16 bytes occupying four words in $Rkey$. $K$ is declared to point to round key $i$. AddRoundKey(i) XORs the bytes of round key $i$ with the corresponding elements of state $s$, placing the result in state $t$.

It is convenient to have a version of AddRoundKey that transforms state $t$ into state $s$. This is defined as follows.

\[
\text{LET} \text{AddRoundKey} \text{ts}(i) \text{ BE} \\
\{ // \text{ Add key round } i \text{ from } s \text{ state to } t \text{ state} \\
\quad \text{LET} \ K = @Rkey!(16*i) \quad // \text{ n = number of elements per row} \\
\quad \text{FOR} \ i = 0 \ \text{TO} \ 15 \ \text{DO} \ \text{stateS}!i := \text{stateT}!i \text{ XOR } K!i \\
\}\]

This function is also the inverse of AddRoundKey_st.

The AddRoundKey functions use round keys numbered 0 to 10, each being 16 words in length, holding one byte per word. This schedule of keys is derived from the given cipher key and is constructed by the function KeyExpansion defined as follows.
4.20. THE ADVANCED ENCRYPTION STANDARD

LET KeyExpansion(key) BE
{ LET rcon = 1

// The first round key is the cipher key itself,
// stored column by column.
Rkey!00, Rkey!01, Rkey!02, Rkey!03 := key%00, key%04, key%08, key%12
Rkey!04, Rkey!05, Rkey!06, Rkey!07 := key%01, key%05, key%09, key%13
Rkey!08, Rkey!09, Rkey!10, Rkey!11 := key%02, key%06, key%10, key%14
Rkey!12, Rkey!13, Rkey!14, Rkey!15 := key%03, key%07, key%11, key%15

// Add 10 more keys to the round schedule
FOR i = 1 TO 10 DO
{ LET p = @Rkey!(16*i) // Pointer to space for key in round i
  LET q = p-16 // Pointer to round key i-1

  p!00 := q!00 XOR sbox%(q!07) XOR rcon
  p!04 := q!04 XOR sbox%(q!11)
  p!08 := q!08 XOR sbox%(q!15)
  p!12 := q!12 XOR sbox%(q!03)

  FOR j = 1 TO 3 DO
  { p!(00+j) := q!(00+j) XOR p!(j-01)
    p!(04+j) := q!(04+j) XOR p!(j+03)
    p!(08+j) := q!(08+j) XOR p!(j+07)
    p!(12+j) := q!(12+j) XOR p!(j+11)
  }

  rcon := mul(2, rcon)
  }
}

Round key 0 is just the given 16 byte cipher key, packed one byte per word. Each subsequent round key is a simple modification of the previous round key. Each of the first 4 bytes of the new round key are the corresponding bytes of the previous key modified by one of the last four bytes of the previous round key changed by an application of the sbox. In addition the first byte of the new round key is modified by rcon which holds the value $2^i$ where $i$ is the new round key number. This value is calculated using the 8-bit arithmetic of GF($2^8$). That is why the next value of rcon is computed by the call mul(2, rcon) using mul, defined below. Words 4 to 15 of the new key is just the exclusive or earlier pairs of words in Rkey.

The next matrix function MixColumns_ts replaces each column of the state matrix t by a values that are linear combinations of the column elements, leaving the result in state s. For instance, it sets s00 to $2 \times t00 + 3 \times t10 + t20 + t30$. All
16 elements of the state are modified, and the total transformation corresponds to the following matrix product.

\[
\begin{pmatrix}
2 & 3 & 1 & 1 \\
1 & 2 & 3 & 1 \\
1 & 1 & 2 & 3 \\
3 & 1 & 1 & 2
\end{pmatrix}
\begin{pmatrix}
t_{00} & t_{01} & t_{02} & t_{03} \\
t_{10} & t_{11} & t_{12} & t_{13} \\
t_{20} & t_{21} & t_{22} & t_{23} \\
t_{30} & t_{31} & t_{32} & t_{33}
\end{pmatrix}
\Rightarrow
\begin{pmatrix}
s_{00} & s_{01} & s_{02} & s_{03} \\
s_{10} & s_{11} & s_{12} & s_{13} \\
s_{20} & s_{21} & s_{22} & s_{23} \\
s_{30} & s_{31} & s_{32} & s_{33}
\end{pmatrix}
\]

When 4 by 4 matrices are multiplied together the rule is as follows.

\[
\begin{pmatrix}
. & . & . & . \\
. & . & . & . \\
a & b & c & d \\
. & . & . & .
\end{pmatrix}
\begin{pmatrix}
. & x & . & . \\
. & y & . & . \\
. & z & . & . \\
. & w & . & .
\end{pmatrix}
\Rightarrow
\begin{pmatrix}
. & . & . & . \\
. & . & . & . \\
. & . & . & . \\
. & . & . & .
\end{pmatrix}
\]

where \( r = ax + by + cz + dw \), thus the value in the \( i^{th} \) row and \( j^{th} \) column of the result is the sum of the products of the elements of the \( i^{th} \) row of the left hand matrix with the corresponding elements of the \( j^{th} \) column of the right hand one.

Since the elements of the state matrix are all 8-bit bytes (held in words), ordinary addition and multiplication cannot be used since they will cause overflow. Instead, arithmetic belonging to the Galois Field \( GF(2^8) \) is used. This replaces \(+\) by XOR and \( x \times y \) by \( \text{mul}(x,y) \), where \( \text{mul} \) is defined as follows.

```c
LET mul(x, y) = VALOF
{ // Return the product of x and y using GF(2**8) arithmetic
  LET res = 0
  WHILE x DO
    { IF (x & 1)>0 DO res := res XOR y
      x := x>>1
      y := y<<1
      IF y > 255 DO y := y XOR #x11B
    }
  RESULTIS res
}
```

This performs the multiplication by conditionally adding \( y \) to the result \( \text{res} \) whenever the least significant bit of \( x \) is a one. Then dividing \( x \) by 2 with a right shift (\( x := x>>1 \)) and doubling \( y \) with a left shift (\( y := y<<1 \)), but whenever

\[1\] Named after the French mathematician Evariste Galois who died aged only 20 in Paris in May 1832 from wounds suffered in a duel. He laid the foundations for Galois theory and Group Theory.
y becomes larger than 255, it is brought back into range by the assignment
\( y := y \text{ XOR } #x11B \). The constant \(#x11B\) was carefully chosen so that, for any \( x \)
in the range 1 to 255, we can find a unique \( y \) such that \( \text{mul}(x,y) = 1 \). Addition and
subtraction are replaced by applications of the XOR operator. We thus have, in
\( \text{GF}(2^8) \), versions of addition, subtraction, multiplication and division that obey
the algebraic rules of ordinary arithmetic, but on values that are always in the
range 0 to 255. You still have to be careful since, for instance \( 2 \times x \neq x + x \) and
\( 3 \times x \) is \( \text{mul}(3,x) = \text{mul}(2,x) \text{ XOR } x \), not \( x + x + x \) which just equal \( x \).

To implement the matrix multiplication, we frequently need to compute expres-
sions of the form \( ax + by + cz + dw \). This is often called the inner product of
\( (a,b,c,d) \) and \( (x,y,z,w) \), and so we have a function called \text{inprod} to do the job.
It definition is as follows.

\[
\text{AND inprod}(a,b,c,d, x,y,z,w) =
\]
\[// \text{ Calculate } ax+by+cz+dw \text{ using } \text{GF}(2^8) \text{ arithmetic}
\text{mul}(a,x) \text{ XOR } \text{mul}(b,y) \text{ XOR } \text{mul}(c,z) \text{ XOR } \text{mul}(d,w)
\]

The implementation of \text{MixColumns} is now straightforward and is as fol-
low.

\[
\text{LET MixColumns}(x) \text{ BE}
\]
\[// \text{ MixColumns function mixes the columns of the state matrix}
\{ // Compute the matrix product
\[// (2 3 1 1) \begin{pmatrix} t00 & t01 & t02 & t03 \\ t10 & t11 & t12 & t13 \end{pmatrix} \Rightarrow \begin{pmatrix} s00 & s01 & s02 & s03 \\ s10 & s11 & s12 & s13 \end{pmatrix}
\]
\[// (1 2 3 1) \begin{pmatrix} t20 & t21 & t22 & t23 \end{pmatrix} \Rightarrow \begin{pmatrix} s20 & s21 & s22 & s23 \end{pmatrix}
\]
\[// (3 1 1 2) \begin{pmatrix} t30 & t31 & t32 & t33 \end{pmatrix} \Rightarrow \begin{pmatrix} s30 & s31 & s32 & s33 \end{pmatrix}
\]
\[
\begin{align*}
s00 & := \text{inprod}(2, 3, 1, 1, t00, t10, t20, t30) \\
s01 & := \text{inprod}(2, 3, 1, 1, t01, t11, t21, t31) \\
s02 & := \text{inprod}(2, 3, 1, 1, t02, t12, t22, t32) \\
s03 & := \text{inprod}(2, 3, 1, 1, t03, t13, t23, t33) \\
s10 & := \text{inprod}(1, 2, 3, 1, t00, t10, t20, t30) \\
s11 & := \text{inprod}(1, 2, 3, 1, t01, t11, t21, t31) \\
s12 & := \text{inprod}(1, 2, 3, 1, t02, t12, t22, t32) \\
s13 & := \text{inprod}(1, 2, 3, 1, t03, t13, t23, t33) \\
s20 & := \text{inprod}(1, 1, 2, 3, t00, t10, t20, t30) \\
s21 & := \text{inprod}(1, 1, 2, 3, t01, t11, t21, t31) \\
s22 & := \text{inprod}(1, 1, 2, 3, t02, t12, t22, t32) \\
s23 & := \text{inprod}(1, 1, 2, 3, t03, t13, t23, t33) \\
s30 & := \text{inprod}(3, 1, 1, 2, t00, t10, t20, t30)
\end{align*}
\]
s31 := inprod(3, 1, 1, 2, t01, t11, t21, t31)
s32 := inprod(3, 1, 1, 2, t02, t12, t22, t32)
s33 := inprod(3, 1, 1, 2, t03, t13, t23, t33)
}

The choice of this transformation matrix is well chosen because multiplication by 1, 2 and 3 in GF(2^8) can be done efficiently both in hardware and software, and it also has the vital property that it has an inverse in GF(2^8) namely:

\[
\begin{pmatrix}
14 & 11 & 13 & 9 \\
9 & 14 & 11 & 13 \\
13 & 9 & 14 & 11 \\
11 & 13 & 9 & 14
\end{pmatrix}
\]

We can easily see that this is indeed the inverse by checking the following equation.

\[
\begin{pmatrix}
14 & 11 & 13 & 9 \\
9 & 14 & 11 & 13 \\
13 & 9 & 14 & 11 \\
11 & 13 & 9 & 14
\end{pmatrix}
\begin{pmatrix}
2 & 3 & 1 & 1 \\
1 & 2 & 3 & 1 \\
1 & 1 & 2 & 3 \\
3 & 1 & 1 & 2
\end{pmatrix}
= \begin{pmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{pmatrix}
\]

The value that should be in element (0,0) of the result is $14 \times 2 + 11 \times 1 + 13 \times 1 + 9 \times 3$ using GF(2^8) arithmetic. Note that $9 \times 3$ is 10010 XOR 1001 = 11011 in binary. So the sum in binary is:

\[
\begin{array}{cccc}
14x2 & 11100 \\
11x1 & 1011 \\
13x1 & 1101 \\
9x3 & 11011 & ( = 10010 \text{ XOR } 1001 ) \\
\hline
00001
\end{array}
\]

Similarly, the value that should be in element (0,1) of the result is:

\[
\begin{array}{cccc}
14x3 & 10010 & ( = 11100 \text{ XOR } 1110 ) \\
11x2 & 10110 \\
13x1 & 1101 \\
9x1 & 1001 \\
\hline
00000
\end{array}
\]
The other 14 elements of the product can easily be checked.

To undo the effect of MixColumns_ts we simply multiply the state matrix by the inverse transform. This is done by \texttt{InvMixColumns_st} define as follows.

\begin{verbatim}
LET InvMixColumns_st() BE
{ // Compute the matrix product
    // (14 11 13 9) (s00 s01 s02 s03) (t00 t01 t02 t03)
    // (9 14 11 13) x (s10 s11 s12 s13) => (t10 t11 t12 t13)
    // (13 9 14 11) (s20 s21 s22 s23) (t20 t21 t22 t23)
    // (11 13 9 14) (s30 s31 s32 s33) (t30 t31 t32 t33)

t00 := inprod(14, 11, 13, 9, s00, s10, s20, s30)
t01 := inprod(14, 11, 13, 9, s01, s11, s21, s31)
t02 := inprod(14, 11, 13, 9, s02, s12, s22, s32)
t03 := inprod(14, 11, 13, 9, s03, s13, s23, s33)
t10 := inprod(9, 14, 11, 13, s00, s10, s20, s30)
t11 := inprod(9, 14, 11, 13, s01, s11, s21, s31)
t12 := inprod(9, 14, 11, 13, s02, s12, s22, s32)
t13 := inprod(9, 14, 11, 13, s03, s13, s23, s33)
t20 := inprod(13, 9, 14, 11, s00, s10, s20, s30)
t21 := inprod(13, 9, 14, 11, s01, s11, s21, s31)
t22 := inprod(13, 9, 14, 11, s02, s12, s22, s32)
t23 := inprod(13, 9, 14, 11, s03, s13, s23, s33)
t30 := inprod(11, 13, 9, 14, s00, s10, s20, s30)
t31 := inprod(11, 13, 9, 14, s01, s11, s21, s31)
t32 := inprod(11, 13, 9, 14, s02, s12, s22, s32)
t33 := inprod(11, 13, 9, 14, s03, s13, s23, s33)
}
\end{verbatim}

The function \texttt{Cipher} defined below performs a long sequence of these matrix transformations. This is a demonstration version since it can output helpful tracing information and has not been optimised to run efficiently.

\begin{verbatim}
LET Cipher(in, out) BE
{ // Copy the input PlainText into the state array.
    s00, s01, s02, s03 := in%00, in%04, in%08, in%12
    s10, s11, s12, s13 := in%01, in%05, in%09, in%13
    s20, s21, s22, s23 := in%02, in%06, in%10, in%14
    s30, s31, s32, s33 := in%03, in%07, in%11, in%15

    IF tracing DO
    { writef("%i2.input ", 0); prstate(stateS)
\end{verbatim}
writef("%i2.k_sch ", 0); prstate(Rkey)
}

// Add the First round key to the state before starting the rounds.
AddRoundKey_st(0)

FOR round = 1 TO Nr-1 DO
{ IF tracing DO
  { writef("%i2.start ", round); prstate(stateT) }

  SubBytes_ts()
  IF tracing DO
  { writef("%i2.s_box ", round); prstate(stateS) }

  ShiftRows_st()
  IF tracing DO
  { writef("%i2.s_row ", round); prstate(stateT) }

  MixColumns_ts()
  IF tracing DO
  { writef("%i2.s_col ", round); prstate(stateS) }

  AddRoundKey_st(round)
  IF tracing DO
  { writef("%i2.k_sch ", round); prstate(@Rkey!(16*round)) }
}

// The last round is given below.
IF tracing DO
{ writef("%i2.start ", Nr); prstate(stateT) }

SubBytes_ts()
IF tracing DO
{ writef("%i2.s_box ", Nr); prstate(stateS) }

ShiftRows_st()
IF tracing DO
{ writef("%i2.s_row ", Nr); prstate(stateT) }

// Do not mix the columns in the final round
AddRoundKey_ts(Nr)
IF tracing DO
{ writef("%i2.k_sch ", Nr); prstate(@Rkey!(16*Nr))
  writef("%i2.output ", Nr); prstate(stateS) }
4.20. THE ADVANCED ENCRYPTION STANDARD

} // The encryption process is over.
// Copy the state array to output array.
out%00, out%04, out%08, out%12 := s00, s01, s02, s03
out%01, out%05, out%09, out%13 := s10, s11, s12, s13
out%02, out%06, out%10, out%14 := s20, s21, s22, s23
out%03, out%07, out%11, out%15 := s30, s31, s32, s33

16 bytes of input data given in in are copied into the state matrix and then modified by the call AddRoundkey(0) before performing 10 rounds of matrix modification. Each round successively calls SubBytes_ts, ShiftRows_st(), MixColumns_ts(), and AddRoundKey_st, except in last round when MixColumns_ts is not called. As a debugging aid the state matrix is conditionally output after each call. After the tenth round is complete the data in the state matrix are copied the byte vector out.

To decipher a message the function InvCipher, defined below, is used. It structure is similar to Cipher but performs the inverse matrix transformations in reverse order, using the same key schedule.

LET InvCipher(in, out) BE
{ // Copy the input CipherText to state array.
  s00, s01, s02, s03 := in%00, in%04, in%08, in%12
  s10, s11, s12, s13 := in%01, in%05, in%09, in%13
  s20, s21, s22, s23 := in%02, in%06, in%10, in%14
  s30, s31, s32, s33 := in%03, in%07, in%11, in%15

  IF tracing DO
    { writef("%i2.iinput ", 0); prstate(stateS)
      writef("%i2.ik_sch ", 0); prstate(@Rkey!(16*Nr))
    } // Add the Last round key to the state before starting the rounds.
    AddRoundKey_st(Nr)
  FOR round = Nr-1 TO 1 BY -1 DO
    { IF tracing DO
      { writef("%i2.istart ", Nr-round); prstate(stateT) }

      InvShiftRows_ts()
      IF tracing DO
        { writef("%i2.is_row ", Nr-round); prstate(stateS) }
    }
InvSubBytes_st()
IF tracing DO
{ writef("%i2.is_box ", Nr-round); prstate(stateT) }

AddRoundKey_ts(round)
IF tracing DO
{ writef("%i2.ik_sch ", Nr-round); prstate(@Rkey!(16*round))
  writef("%i2.is_add ", Nr-round); prstate(stateS)
}

InvMixColumns_st()

IF tracing DO
{ writef("%i2.istart ", Nr); prstate(stateT) }

// The final round is given below.
InvShiftRows_ts()
IF tracing DO { writef("%i2.is_row ", Nr); prstate(stateS) }

InvSubBytes_st()
IF tracing DO { writef("%i2.is_box ", Nr); prstate(stateT) }

// Do not mix the columns in the final round
AddRoundKey_ts(0)
IF tracing DO { writef("%i2.ik_sch ", Nr); prstate(@Rkey!(16*0))
  writef("%i2.ioutput", Nr); prstate(stateS) }

// The decryption process is over.
// Copy the state array to output array.
out%00, out%04, out%08, out%12 := s00, s01, s02, s03
out%01, out%05, out%09, out%13 := s10, s11, s12, s13
out%02, out%06, out%10, out%14 := s20, s21, s22, s23
out%03, out%07, out%11, out%15 := s30, s31, s32, s33

The main program start exercises these two functions with 16 bytes of plain
text and 16 bytes of cipher key. In this version KeyExpansion, Cipher and
InvCipher are called using the library function instrcount which returns the
number of Cintcode instructions executed during each call.

LET start() = VALOF
{ LET argv = VEC 50
4.20. THE ADVANCED ENCRYPTION STANDARD

LET plain = TABLE #X33221100, #X77665544, #XBBAA9988, #XFFEEDDCC
LET key = TABLE #x03020100, #x07060504, #x0B0A0908, #x0F0E0D0C
// The plain text and key are the same as given in the detailed
// example in Appendix C.1 in
csrc.nist.gov/publications/fips/fips197/fips-197.pdf
// It provides a useful check that this implementation is correct.
// Just execute: aes128 -t
LET in = VEC 63
LET out = VEC 63
LET v = VEC 10*16+15 // For the key schedule of 11 keys
LET countExpand, countCipher, countInvCipher = 0, 0, 0

Rkey := v
stateS, stateT := @$00, @$00

UNLESS rdargs("-t/s", argv, 50) DO
{ writef("Bad arguments for aes128*n")
  RESULTIS 0
}

tracing := argv!0

inittables()

//KeyExpansion(key)
countExpand := instrcount(KeyExpansion, key)

IF tracing DO
{ writef("*nKey schedule*n")
  FOR i = 0 TO Nr DO
   { LET p = 16*i
     writef("%i2: ", i)
     prstate(@Rkey!p)
   }
  }
}
newline()

writef("plain: "); prbytes(plain); newline()
writef("key: "); prbytes(key)
newline()

//Cipher(plain, out)
countCipher := instrcount(Cipher, plain, out)

newline()
CHAPTER 4. THE BCPL CINTCODE SYSTEM

writef("Cipher text: "); prbytes(out); newline()

//InvCipher(out, in)
countInvCipher := instrcount(InvCipher, out, in)
IF tracing DO newline()

writef("InvCipher text: "); prbytes(in); newline()

newline()
writef("Cintcode instruction counts*n*n")
writef("KeyExpansion: %i7*n", countExpand)
writef("Cipher: %i7*n", countCipher)
writef("InvCipher: %i7*n", countInvCipher)

RESULTIS 0
}

The remaining functions, defined below, are used to provide the debugging output.

AND prstate(m) BE
{ // For outputting state matrix or keys, column by column.
   FOR i = 0 TO 3 DO
   { wrch(' ')
      FOR j = 0 TO 3 DO
      { writef("%x2", m!(4*j+i))
      }
   }
   newline()
}

AND prbytes(v) BE
{ // For outputting plain and ciphered text.
   FOR i = 0 TO 15 DO
   { IF i MOD 4 = 0 DO wrch(' ')}
   writef("%x2", v%i)
   }
   newline()
}

When aes128 is run without arguments the output is as follows.
4.20. **THE ADVANCED ENCRYPTION STANDARD**

0.050> aes128

plain: 00112233 44556677 8899AABB CCDDEEFF

key: 00010203 04050607 08090A0B 0C0D0E0F

Cipher text: 69C4E0D8 6A7B0430 D8CDB780 70B4C55A

InvCipher text: 00112233 44556677 8899AABB CCDDEEFF

Cintcode instruction counts

KeyExpansion: 2834
Cipher: 33688
InvCipher: 63581
0.010>

This shows that the given plain text is converted by **Cipher** to suitably random looking text using the given key and that **InvCipher** restores the original plain text correctly.

You will also notice that **InvCipher** executes nearly twice as many Cintcode instructions as **Cipher**. This somewhat surprising result is because much of the time is spent in **mul** while performing the matrix multiplications in **MixColumns** and **InvMixColumns**. In **MixColumns** **mul** is multiplying by 1, 2 or 3 which takes far fewer instructions than the calls of **mul** in **InvMixColumns** where the multiplications are by 9, 11, 13 or 14.

For completeness, I have included a demonstration version of AES using a 256 bit cipher key. This program is called `bcplprog/raspi/aes256.b`. It has much in common with `aes128.b` using, for instance, the same 4 by 4 state matrix and the same matrix transformations, but it performs 14 rounds rather than 10. The main difference is how the schedule of 16 byte keys are generated from the given 32 byte cipher key. The increased running time of **aes256** is small being mainly due to the increased number of rounds.

### 4.20.1 Final Observation

The security of encryption is based entirely on keeping keys secret and not on hiding the details of the encryption algorithm. After all AES is available on thousands of million machines around the world and anyone with a superuser of administrator password would be able to see the algorithm.

I hope you agree that AES128 is incredibly simple and elegant, and remarkably efficient. It is natural to wonder whether it could be extended to allow even stronger encryption. One obvious possibility is to use a larger state matrix, perhaps of size $8 \times 8$ or even $16 \times 16$. All the transformations are easily extended
except possibly for the difficulty of finding suitable column mixing matrices and their inverses using \( \text{GP}(2^8) \) arithmetic. But this turns out to be simple with the aid of the program \texttt{invert.b} which shows, for instance, that the following two 8 × 8 matrices are mutual inverses:

\[
\begin{pmatrix}
73 & 129 & 196 & 102 & 231 & 219 & 65 & 198 \\
198 & 73 & 129 & 196 & 102 & 231 & 219 & 65 \\
65 & 198 & 73 & 129 & 196 & 102 & 231 & 219 \\
219 & 65 & 198 & 73 & 129 & 196 & 102 & 231 \\
231 & 219 & 65 & 198 & 73 & 129 & 196 & 102 \\
102 & 231 & 219 & 65 & 198 & 73 & 129 & 196 \\
129 & 196 & 102 & 231 & 219 & 65 & 198 & 73
\end{pmatrix}
\times
\begin{pmatrix}
2 & 3 & 4 & 5 & 6 & 1 & 1 & 1 \\
1 & 2 & 3 & 4 & 5 & 6 & 1 & 1 \\
1 & 1 & 2 & 3 & 4 & 5 & 6 & 1 \\
1 & 1 & 1 & 2 & 3 & 4 & 5 & 6 \\
6 & 1 & 1 & 2 & 3 & 4 & 5 & 6 \\
5 & 6 & 1 & 1 & 2 & 3 & 4 & 5 \\
4 & 5 & 6 & 1 & 1 & 2 & 3 & 4 \\
3 & 4 & 5 & 6 & 1 & 1 & 1 & 2
\end{pmatrix}
= \begin{pmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{pmatrix}
\]

using \( \text{GF}(2^8) \) arithmetic. The program will also find inverses of matrices of other sizes, such as 16 × 16.

If we choose to use 8 × 8 matrices it would be natural to use a key length of 64 bytes (or 512 bits), and encrypt the data in blocks of 64 bytes. For 16 × 16 matrices, we would use keys of 256 bytes (or 2048 bits). As a demonstration, the program \texttt{aesnxn.b} implements these two possibilities. Using the \texttt{stats} option \texttt{aesnxn} will output some statistics on the encoding process. For instance, some of the output generated by the command: \texttt{aesnxn 8 stats} is as follows.

**Histogram of the number of bits changed**

<table>
<thead>
<tr>
<th>Number</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>16</td>
<td>0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>224</td>
<td>1 0 1 0 0 1 1 2 4 2 2 3 7 2 6 6</td>
</tr>
<tr>
<td>240</td>
<td>8 4 7 11 12 12 13 19 16 13 14 12 13 21 21 21</td>
</tr>
<tr>
<td>256</td>
<td>17 12 16 14 22 13 11 17 16 18 10 13 13 6 10 7</td>
</tr>
<tr>
<td>272</td>
<td>7 9 3 6 3 2 3 2 1 2 1 1 1 0 0 1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
4.20. THE ADVANCED ENCRYPTION STANDARD

Histogram of the number of times each bit changes

| 0:   | 251 275 239 261 257 268 224 247 267 258 266 267 258 266 252 259 255 |
| 16:  | 251 257 260 268 273 267 264 245 270 252 231 255 244 262 274 262 |
| 32:  | 256 255 261 245 252 251 258 252 265 254 257 259 264 256 246 266 |
| ...  | 264 251 260 235 264 252 241 255 271 259 256 255 249 251 271 256 |
| 480: | 265 247 238 267 256 251 250 256 257 265 259 236 266 247 259 254 |
| 496: | 255 265 263 259 257 267 275 243 272 236 251 247 265 257 250 252 |

This shows two histograms based on 512 runs of the ciphering process complementing a different bit of the plain text each time. The first histogram shows that roughly half the bits of the encyphered data change each time, and the second shows that every bit of the encyphered data is equally likely to change.

As a final remark, you may like to look at the following function.

\[
\text{AND increment}(p, w) \text{ BE WHILE } w \text{ DO}
\]
\[
\{ \text{ LET } c = !p \& w \text{ // The carry bits} \\
\quad !p := !p \text{ XOR } w \\
\quad w, p := c, p+1 \text{ // The next bit position is one word later.} 
\}
\]

This function is used to increment the counts needed by the second histogram. The argument \( p \) points to a word containing the least significant bit of 32 counters. The more significant bits are held in \( p!1, p!2 \) and so on. The argument \( w \) is a bit pattern specifying which counters are to be incremented. The function is thus capable of incrementing any subset of 32 counters simultaneously.

The function \text{countervalue}, defined below, converts a selected counter to a normal integer. Note that the counters in this implementation are limited to 16 bits.

\[
\text{AND countvalue}(p, \text{bit}) = \text{VALOF} \\
\{ \text{ LET } res = 0 \\
\quad \text{FOR } j = 15 \text{ TO } 0 \text{ BY } -1 \text{ DO} \\
\quad \{ res := 2*res \\
\quad \quad \text{UNLESS } (p!j \& \text{bit}) = 0 \text{ DO } res := res+1 \\
\quad \} 
\}
\]

RESULTIS res
4.21 GF($2^8$) Arithmetic

We have seen that GF($2^8$) arithmetic was used in the implementation of the advanced encryption standard, but it turns out this form of arithmetic is used in many other algorithms, so it is worth a little more explanation. GF($2^8$) is an example of a mathematical field and such a field consists of a set of elements and two operators normally written as $+$ for addition and $\times$ for multiplication, satisfying the following algebraic rules.

(1) If $x$ and $y$ are elements of the set then so are $x + y$ and $x \times y$.
(2) If $x$, $y$ and $z$ are elements of the set then $x + (y + z) = (x + y) + z$ and $x \times (y \times z) = (x \times y) \times z$.
(3) If $x$ and $y$ are elements of the set then $x + y = y + x$ and $x \times y = y \times x$.
(4) There exists an element 0 such that $x + 0 = x$ for all $x$ in the set.
(5) There exists an element 1 different from 0 such that $x \times 1 = x$ for all $x$ in the set.
(6) For every element $x$ in the set, there exist an element $y$ such that $x + y = 0$.
(7) For every element $x$ in the set other than 0, there exist an element $y$ such that $x \times y = 1$.
(8) If $x$, $y$ and $z$ are elements of the set then $x \times (y + z) = (x \times y) + (x \times z)$.

You will notice that signed real numbers satisfy these properties but unsigned reals do not, for instance, there is no unsigned $y$ satisfying $1.5 + y = 0$. Similarly neither signed nor unsigned integers form a field since, for instance, there is no $y$ satisfying $7 \times y = 1$. However GF arithmetic does satisfy all these rules and has the valuable property that the set of elements is of finite size. For GF($2^8$) the number of elements is 256. Although algebra in GF($2^8$) feels similar to that on real numbers, you still have to be careful. For instance, $x + x$ is equal to zero and not $2 \times x$.

One notable example of the use of GF arithmetic is in the Reed-Solomon Error Correcting Codes. A simple demonstration is given in the following sections. The program starts with the following declarations which decalres variables that will be described later when they are used.

GET "libhdr"

GLOBAL {
  testno:ug // =0 for small demo, =1 for a larger demo.
  gf_log2 // Vector of discrete logarithms in GF($2^8$)
  gf_exp2 // Vector of powers of 2 in GF($2^8$)
  n // The codeword length in bytes
  k // The message length
  e // n-k The number of parity bytes
We have already seen that addition and subtraction in \( \text{GF}(2^8) \) are replaced by XOR but, for completeness, we define the following two functions.

\[
\text{LET } \text{gf}_\text{add}(x, y) = x \oplus y
\]

\[
\text{AND } \text{gf}_\text{sub}(x, y) = x \oplus y
\]

In \( \text{GF}(2^8) \), 2 has the interesting property that all 255 elements other than 0 can be represented by \( 2^n \) for suitably chosen values of \( n \). It is useful to precompute these powers of 2 placing them in a vector \( \text{gf}_\text{exp2} \) and while doing so we can construct a vector \( \text{gf}_\text{log2} \) holding this inverse values. These two vectors are allocated and initialised by the function \( \text{initlogs} \), defined as follows.

\[
\text{AND } \text{initlogs}() \text{ BE}
\]

\[
\{ \text{LET } x = 1
\]

\[
\text{gf}_\text{log2} := \text{getvec}(255)
\]

\[
\text{gf}_\text{exp2} := \text{getvec}(510) \text{ // 510 = 255+255}
\]

\[
\text{// Using a double sized vector for exp2 improves the efficiency}
\]

\[
\text{// of functions such as } \text{gf}_\text{mul} \text{ and } \text{gf}_\text{div}, \text{ defined below.}
\]

\[
\text{UNLESS } \text{gf}_\text{log2} \& \text{gf}_\text{exp2} \text{ DO}
\]

\[
\{ \text{writef(}"\text{initlogs: More space needed*n}"
\]

\[
\text{abort}(999)
\]

\[
\}
\]

\[
\text{gf}_\text{log2}!0 := -1 \text{ // log2 of zero is undefined.}
\]

\[
\text{FOR } i = 0 \text{ TO } 255 \text{ DO } \text{ // All possible element values}
\]

\[
\{ \text{ // } 2^{-i} = x \text{ so } i = \text{log2}(x)
\]

\[
\text{gf}_\text{exp2!}i := x
\]

\[
\text{gf}_\text{exp2!}(i+255) := x \text{ // Note } 2^{-255=1} \text{ in } \text{GF}(2^8)
gf_log2!x := i
x := x<<1  // Multiply x by 2
UNLESS ( x & #b_1_0000_0000 ) = 0 DO
  x := x XOR #b_1_0001_1101
}
}

The vectors $\text{gf}_\text{exp2}$ and $\text{gf}_\text{log2}$ are used in the definitions of $\text{gf}_\text{mul}$ defined below based on the following observation.

$$x \times y = 2^{\log_2(x)} \times 2^{\log_2(y)} = 2^{\log_2(x)+\log_2(y)}$$

Since $\log_2(0)$ is undefined, cases where $x$ or $y$ are zero are treated specially.

AND $\text{gf}_\text{mul}(x, y) = \text{VALOF}$
{ IF $x=0$ | $y=0$ RESULTIS 0 
  RESULTIS $\text{gf}_\text{exp2}!(\text{gf}_\text{log2}!x + \text{gf}_\text{log2}!y)$
}

The functions $\text{gf}_\text{div}$, $\text{gf}_\text{pow}$ and $\text{gf}_\text{inverse}$ are also implemented efficiently using these vectors.

AND $\text{gf}_\text{div}(x, y) = \text{VALOF}$
{ IF $y=0$ DO 
  { writef("gf_div: Division by zero")
    abort(999)
  }
  IF $x=0$ RESULTIS 0
  RESULTIS $\text{gf}_\text{exp2}!(255 + \text{gf}_\text{log2}!x - \text{gf}_\text{log2}!y)$
}

AND $\text{gf}_\text{pow}(x,y) = \text{gf}_\text{exp2}!(\text{gf}_\text{log2}!x \times y \mod 255)$

AND $\text{gf}_\text{inverse}(x) = \text{gf}_\text{exp2}!(255 - \text{gf}_\text{log2}!x)$

### 4.22 Polynomials with GF($2^8$) Coefficients

The Reed-Solomon Error Correction mechanism make extensive use of polynomials with GF coefficients, so this section presents some functions relating to such polynomials. In this implementation polynomials are represented by vectors containing the degree of the polynomial and its coefficients. If $p$ points to such a polynomial then $p!0$ holds its degree, $n$ say, and $p!1$ holds the coefficient of $x^n$. Successive elements of $p$ hold the coefficients of lower powers of $x$, with the final coefficient in $p!(n+1)$ representing the constant term. So the polynomial $5x^2 + 6x + 7$ would be represented by a vector whose elements are 2,5,6 and 7.
4.22. POLYNOMIALS WITH \( GF(2^8) \) COEFFICIENTS

The first few polynomial functions are straightforward and need no additional explanation.

```
AND gf_poly_copy(p, q) BE
{ // Copy polynomial from p to q.
    FOR i = 0 TO p!0+1 DO q!i := p!i
}

AND gf_poly_scale(p, x, q) BE
{ // Multiply, using gf_mul, every coefficient of polynomial p by
    // scalar x leaving the result in q.
    LET deg = p!0 // The degree of polynomial p
    q!0 := deg // The degree of the result
    FOR i = 1 TO deg+1 DO q!i := gf_mul(p!i, x)
}

AND gf_poly_add(p, q, r) BE
{ // Add polynomials p and q leaving the result in r
    LET degp = p!0 // The number of coefficients is one larger
    LET degq = q!0 // than the degree of the polynomial.
    LET degr = degp
    IF degq>degr DO degr := degq
    // degr is the larger of the degrees of p and q.
    r!0 := degr // The degree of the result
    FOR i = 1 TO degr+1 DO r!(i+degr-degp) := p!i
    FOR i = 1 TO degr-degp DO r!i := 0 // Pad higher coeffs with 0s
    FOR i = 1 TO degq+1 DO r!(i+degr-degq) := r!(i+degr-degq) XOR q!i
}

// GF addition and subtraction are the same.
AND gf_poly_sub(p, q, r) BE gf_poly_add(p, q, r)

AND gf_poly_mul(p, q, r) BE
{ // Multiply polynomials p and q leaving the result in r
    LET degp = p!0
    LET degq = q!0
    LET degr = degp+degq

    r!0 := degr // Degree of the result
    FOR i = 1 TO degr+1 DO r!i := 0
    FOR j = 1 TO degq+1 DO
        FOR i = 1 TO degr+1 DO
            r!(i+j-1) := r!(i+j-1) XOR gf_mul(p!i, q!j)
    }

AND gf_poly_mulbyxn(p, n, r) BE
```
\[
\begin{align*}
\{ & // Multiply polynomials \ p \ by \ x^n \ leaving \ the \ result \ in \ \ r \\
& \quad \text{LET} \ degp = p!0 \\
& \quad \text{LET} \ degr = degp + n \\
& \quad \text{r!0} := degr \\
& \quad \text{FOR} \ i = 1 \ \text{TO} \ degp+1 \ \text{DO} \ r!i := p!i \\
& \quad \text{FOR} \ i = degp+2 \ \text{TO} \ degr+1 \ \text{DO} \ r!i := 0 \\
\}
\]

\[
\text{AND} \ \text{gf\_poly\_eval}(p, \ x) = \text{VALOF} \\
\{ & // Evaluate polynomial \ p \ for \ a \ given \ x \ using \ Horner's \ method. \\
& \quad \text{// Eg use: \ ax^3 + bx^2 + cx^1 + d = ((ax + b)x + c)x + d} \\
& \quad \text{LET} \ res = p!1 \\
& \quad \text{FOR} \ i = 2 \ \text{TO} \ p!0+1 \ \text{DO} \\
& \quad \quad \text{res := gf\_mul(res,x) XOR p!i // mul by \ x \ and \ add \ next \ coeff} \\
& \quad \quad \text{RESULTIS res} \\
\}
\]

\[
\text{AND} \ \text{pr\_poly}(p) = \text{BE} \\
\{ & // Output the polynomial \ in \ hex \\
& \quad \text{FOR} \ i = 1 \ \text{TO} \ p!0+1 \ \text{DO} \ \text{writef(" \ %x2", \ p!i)} \\
& \quad \text{newline()} \\
\}
\]

\[
\text{AND} \ \text{pr\_poly\_dec}(p) = \text{BE} \\
\{ & // Output the polynomial \ in \ decimal \\
& \quad \text{FOR} \ i = 1 \ \text{TO} \ p!0+1 \ \text{DO} \ \text{writef(" \ %i3", \ p!i)} \\
& \quad \text{newline()} \\
\}
\]

The function \text{gf\_poly\_divmod} divides polynomial \ p \ by polynomial \ q \ using long division leaving both the quotient and remainder in \ r.

\[
\text{AND} \ \text{gf\_poly\_divmod}(p, \ q, \ r) = \text{BE} \\
\{ & \quad \text{LET} \ degp = p!0 \ // \ The \ degree \ of \ polynomial \ p. \\
& \quad \text{LET} \ degq = q!0 \ // \ The \ degree \ of \ polynomial \ q. \\
& \quad \text{LET} \ degr = degp \\
& \quad \text{LET} \ t = \text{VEC} \ 255 \ // \ Vector \ to \ hold \ the \ next \ product \ of \ the \ generator \\
& \quad \text{UNLESS} \ q!1 > 0 \ \text{DO} \\
& \quad \quad \text{writef("The divisor must have a non zero leading coefficient\n")} \\
& \quad \quad \text{abort(999)} \\
& \quad \quad \text{RETURN} \\
\}
\]
4.22. **POLYNOMIALS WITH GF($2^8$) COEFFICIENTS**

// Copy polynomial p into r.
\[ r!0 := \text{deg}_r \]
FOR i = 1 TO degr+1 DO r!i := p!i

//writef("p: "); pr_poly(p)
//writef("q: "); pr_poly(q)
//writef("r: "); pr_poly(r)

FOR i = 1 TO degp-degq+1 DO
{ LET dig = gf_div(r!i, q!1)
  IF dig DO
  { gf_poly_scale(q, dig, t)
    //writef("scaled q: ")
    //FOR j = 2 TO i DO writef(" ")
    //pr_poly(t)
    r!i := dig
    FOR j = 2 TO t!0+1 DO r!(i+j-1) := r!(i+j-1) XOR t!j
  }
  //writef("new r: "); pr_poly(r)
}

If the write statements in \texttt{gf\_poly\_divmod} are un-commented, it is possible to generate the following output.

\[
\begin{array}{l}
\text{p:} \quad 12 \ 34 \ 56 \ 78 \ 00 \ 00 \ 00 \ 00 \ 00 \ 00 \\
\text{q:} \quad 71 \ 11 \ 22 \ 33 \ 44 \ 55 \ 66 \\
\text{initial r:} \quad 12 \ 34 \ 56 \ 78 \ 00 \ 00 \ 00 \ 00 \ 00 \ 00 \\
\text{scaled q:} \quad 12 \ F4 \ F5 \ 01 \ F7 \ 03 \ 02 \\
\text{new r:} \quad 2E \ C0 \ A3 \ 79 \ F7 \ 03 \ 02 \ 00 \ 00 \ 00 \\
\text{scaled q:} \quad C0 \ 4A \ 94 \ DE \ 35 \ 7F \ A1 \\
\text{new r:} \quad 2E \ 82 \ E9 \ ED \ 29 \ 36 \ 7D \ A1 \ 00 \ 00 \\
\text{scaled q:} \quad E9 \ D8 \ AD \ 75 \ 47 \ 9F \ EA \\
\text{new r:} \quad 2E \ 82 \ AA \ 35 \ 84 \ 43 \ 3A \ 3E \ EA \ 00 \\
\text{scaled q:} \quad 35 \ C1 \ 9F \ 5E \ 23 \ E2 \ BC \\
\text{new r:} \quad 2E \ 82 \ AA \ 1C \ 45 \ DC \ 64 \ 1D \ 08 \ BC
\end{array}
\]

This shows the long division steps being used to divide \texttt{p} by \texttt{q}. The quotient \texttt{2E 82 AA 1C} and remainder \texttt{45 DC 64 1D 08 BC} are left in \texttt{r}. The functions \texttt{gf\_poly\_div} and \texttt{gf\_poly\_mod} use \texttt{gf\_poly\_divmod} to obtain the quotient and remainder separately.

\[
\text{AND gf\_poly\_div(p, q, r) BE}
\{ \text{gf\_poly\_divmod(p, q, r)}
\quad r!0 := p!0 - q!0 \quad \text{// Select just the quotient}
\]
AND gf_poly_mod(p, q, r) BE
{ LET degp = p!0
  LET degq = q!0
  LET degr = degq - 1

  gf_poly_divmod(p, q, r)

  r!0 := degr // Overwrite the quotient with the remainder.
  FOR i = 1 TO degr+1 DO r!i := r!(i+degp-degr)
}

4.23 Reed-Solomon Error Correction

Reed-Solomon Error Correction take a sequence of message elements combined with an arbitrary number of parity elements to form a codeword that can be corrected provided not too many of its elements have been corrupted. It is used in 2D QR barcodes where errors might occur as a result of the scanner misreading a damaged image, and it is also used in radio communication such as digital television where errors might occur as the result of weak signals or electrical interference. The mechanism is both efficient and almost optimal. The codewords represent polynomials whose coefficients use use GF(2^4) for digital television or GF(2^8) for QR barcodes. This demonstration program used GF(2^8) and we will assume that the elements are 8-bit bytes.

If there are e parity bytes then all errors can be found and corrected provided there are no more than e/2 of them. In the unusual situation where the locations of the errors are known, up to e errors can be corrected.

Assuming we have a message of k bytes, this can be represented as a polynomial of degree k - 1 using the message bytes as the coefficients. To add e parity bytes, we multiply the message polynomial by x^e and add the remainder after dividing it by a special generator polynomial of degree e. The generator polynomial is the expansion of:

\[(x - 2^0)(x - 2^1)(x - 2^2) \ldots (x - 2^{(e-1)})\]

The following function creates the generating polynomial of degree e placing the result in g.

AND gf_generator_poly(e, g) BE
{ // Set in g the polynomial resulting from the expansion of
  // (x-2^0)(x-2^1)(x-2^2) \ldots (x-2^{(e-1)}). Note that it is
  // of degree e and that the coefficient of x^e is 1.
  LET t = VEC 255
  g!0, g!1 := 0, 1 // The polynomial: 1.
FOR i = 0 TO e-1 DO
    { LET d, a, b = 1, 1, gf_pow(2,i) // (x + 2^i)
      // @d points to polynomial: (x - 2^i)
      // which in GF arithmetic is also: (x + 2^i)
      FOR i = 0 TO g!0+1 DO t!i := g!i // Copy g into t
        gf_poly_mul(t, @d, g) // Multiply t by (x-2^i) into g
    }
}

The function \texttt{rs\_encode\_msg} returns in \texttt{r} the polynomial \texttt{Msg} concatenated with the \texttt{e} Reed-Solomon check bytes which represent remainder after the \texttt{Msg} polynomial multiplied by \texttt{x^e} and divided by the generator polynomial created by \texttt{rs\_generator\_poly}. As an example with message polynomial \texttt{12 34 56 78} and \texttt{e=6}, the generator polynomial is \texttt{01 3F 01 DA 20 E3 26} and the division proceeds as follows.

\[
\begin{array}{c}
12 \ 9D \ 43 \ 57 \\
\hline
01 \ 3F \ 01 \ DA \ 20 \ E3 \ 26 \ ) \ 12 \ 34 \ 56 \ 78 \ 00 \ 00 \ 00 \ 00 \ 00 \ 00 \ 00 \ 00 \\
12 \ A9 \ 12 \ 88 \ 7A \ 4D \ 16 \\
\hline
9D \ 44 \ F0 \ 7A \ 4D \ 16 \ 00 \\
9D \ 07 \ 9D \ 3F \ 4A \ 51 \ 23 \\
\hline
43 \ 6D \ 45 \ 07 \ 47 \ 23 \ 00 \\
43 \ 3A \ 43 \ F7 \ 88 \ 5A \ 1F \\
\hline
57 \ 06 \ F0 \ CF \ 79 \ 1F \ 00 \\
57 \ 11 \ 57 \ 99 \ 32 \ 67 \ DD \\
\hline
17 \ A7 \ 56 \ 4B \ 78 \ DD \end{array}
\]

It thus computes \texttt{12 9D 43 57} as the quotient and \texttt{17 A7 56 4B 7B DD} as the remainder. As can be seen, the process is basically long division using \texttt{gf\_mul} for multiplication and \texttt{XOR} for subtraction. If at each stage the senior byte is not subtracted, the senior 4 bytes of the accumulator become the quotient and the junior 6 bytes hold the remainder. This assumes that the senior coefficient of the generator polynomial is always a one. If, at the end, we replace the quotient bytes of the accumulator by the original message bytes, we create the Reed-Solomon codeword.

The definition of \texttt{rs\_encode\_msg} is as follows.

\texttt{AND rs\_encode\_msg() BE}

\{ // This appends e Reed-Solomon parity bytes onto the end of the}
We have seen the a Reed-Solomon codeword consists of \( k \) bytes of message followed by \( e \) parity bytes which represent the remainder after dividing the message polynomial multiplied by \( x^e \) by the generator polynomial. Since addition and subtraction are both the same in GF arithmetic, the codeword will be exactly divisible by the generating polynomial and, since the generator polynomial is the product of many factors of the form \((1 - x \cdot 2^i)\), each of these also divides into the codeword exactly. However, if some bytes of the codeword are corrupted, most of these factors will not divide the corrupted codeword exactly. We can easily create a polynomial of degree \( e - 1 \) whose coefficients are the \( e \) remainders obtained when attempting to divide the corrupted codeword by each factor of the generator polynomial.

To demonstrate how the error correction is performed, we will use an example of a 4 byte message \( 12 \ 34 \ 56 \ 78 \) and 6 parity bytes. We thus have \( k = 4 \), \( e = 6 \) and so \( n = 10 \). The generator polynomial \( G(x) \) is therefore:

\[
G(x) = (x - 2^0)(x - 2^1)(x - 2^2)(x - 2^3)(x - 2^4)(x - 2^5) \\
= (x - 01)(x - 02)(x - 04)(x - 08)(x - 10)(x - 20) \\
= 01x^6 + 3Fx^5 + 01x^4 + DAx^3 + 20x^2 + E3x + 26
\]

It turns out that using this generator of this form maximises the Hamming distance between codewords.

From now on we will write \( G \) for the generator, \( M \) the codeword and \( R \) the corrupted codeword as follows:

\[
G = 01 \ 3F \ 01 \ DA \ 20 \ E3 \ 26 \\
M = 12 \ 34 \ 56 \ 78 \ 17 \ A7 \ 56 \ 4B \ 78 \ DD \\
R = 12 \ 34 \ 00 \ 00 \ 17 \ 00 \ 56 \ 4B \ 78 \ DD
\]

You will notice that bytes 3, 4 and 6 of the codeword have been zeroed, and that these correspond to the coefficients of \( x^7 \), \( x^6 \) and \( x^4 \), respectively.

In general, when we attempt to read a codeword some of its bytes may be corrupted resulting in a different polynomial \( R(x) \) which can be written as the
sum of $M(x)$, the original codeword, and $E(x)$ an errors polynomial giving a correction value for each coefficient of $R$. This is stated in the following equation:

$$R(x) = M(x) + E(x)$$

Assuming the corrupted codeword is:

$$R = 12\ 34\ 00\ 00\ 17\ 00\ 56\ 4B\ 78\ DD$$

then the errors polynomial $E$ will be:

$$E = 00\ 00\ 56\ 78\ 00\ A7\ 00\ 00\ 00\ 00$$

which when added to $R$ gives the corrected codeword. Our problem is how to deduce the errors polynomial knowing only $R$ and the generator polynomial. It turns out that we can, provided not too many bytes have been corrupted. With 6 check bytes we can find and correct the $6/2=3$ corrupted bytes in $R$.

To do this we first construct a polynomial $S$ (called the syndromes polynomial) whose coefficients are the remainders after dividing $R$ by each of the factors of the generator polynomial. In our example $e = 6$ so the generator has 6 factors $(x - 2^0)$, $(x - 2^1)$, $(x - 2^2)$, $(x - 2^3)$, $(x - 2^4)$ and $(x - 2^5)$. $S$ can be written as

$$S(x) = S_5x^5 + S_4x^4 + S_3x^3 + S_2x^2 + S_1x + S_0$$

When we divide $R$ by $(x - 2^i)$ we obtain a quotient polynomial $Q_i$ and a remainder $S_i$. These, of course, satisfy the following equation:

$$R(x) = (x - 2^i) \ast Q_i(x) + S_i$$

and if we set $x = 2^i$ this reduces to

$$R(2^i) = S_i$$

So $S_i$ can be calculated just by evaluating the polynomial $R(x)$ at $x = 2^i$. For our example the syndromes polynomial is:

$$S = 2E\ B8\ 0E\ CB\ 50\ 35$$

If we happen to know in advance the positions in the codeword that have been corrupted, in this case 3, 4 and 6, then we could write the errors polynomial as

$$E(x) = Y1\ast x^-7 + Y2\ast x^-6 + Y3\ast x^-4$$

Hopefully there is sufficient information to deduce these positions and $Y1=56$, $Y2=78$ and $Y3=A7$.

Since we have just shown $E(2^-i) = S_i$, and assuming we know the error positions, we can say
Si = E(2^i)
    = Y1*2^i + Y2*2^i + Y3*2^i
where X1 = 2^7, X2 = 2^6 and X3 = 2^4

These 6 equations can be written as a matrix product as follows:

\[
\begin{align*}
\text{S0} &= \begin{pmatrix} X1^0 & X2^0 & X3^0 \end{pmatrix} \times \begin{pmatrix} Y1 \end{pmatrix} \\
\text{S1} &= \begin{pmatrix} X1^1 & X2^1 & X3^1 \end{pmatrix} \times \begin{pmatrix} Y2 \end{pmatrix} \\
\text{S2} &= \begin{pmatrix} X1^2 & X2^2 & X3^2 \end{pmatrix} \times \begin{pmatrix} Y3 \end{pmatrix} \\
\text{S3} &= \begin{pmatrix} X1^3 & X2^3 & X3^3 \end{pmatrix} \\
\text{S4} &= \begin{pmatrix} X1^4 & X2^4 & X3^4 \end{pmatrix} \\
\text{S5} &= \begin{pmatrix} X1^5 & X2^5 & X3^5 \end{pmatrix}
\end{align*}
\]

We know that
\[
S = 2E\ B8\ 0E\ CB\ 50\ 35
\]
and assuming we know that \(X1=2^7\), \(X2=2^6\) and \(X3=2^4\), this product simplifies to

\[
\begin{align*}
\text{S0} &= \begin{pmatrix} 2^0 & 2^0 & 2^0 \end{pmatrix} \times \begin{pmatrix} Y1 \end{pmatrix} \\
\text{S1} &= \begin{pmatrix} 2^7 & 2^6 & 2^4 \end{pmatrix} \times \begin{pmatrix} Y2 \end{pmatrix} \\
\text{S2} &= \begin{pmatrix} 2^14 & 2^12 & 2^8 \end{pmatrix} \times \begin{pmatrix} Y2 \end{pmatrix} \\
\text{S3} &= \begin{pmatrix} 2^21 & 2^18 & 2^12 \end{pmatrix} \\
\text{S4} &= \begin{pmatrix} 2^28 & 2^24 & 2^16 \end{pmatrix} \\
\text{S5} &= \begin{pmatrix} 2^35 & 2^30 & 2^20 \end{pmatrix}
\end{align*}
\]

or

\[
\begin{align*}
\text{S0} &= \begin{pmatrix} 01 & 01 & 01 \end{pmatrix} \times \begin{pmatrix} Y1 \end{pmatrix} \\
\text{S1} &= \begin{pmatrix} 80 & 40 & 10 \end{pmatrix} \times \begin{pmatrix} Y2 \end{pmatrix} \\
\text{S2} &= \begin{pmatrix} 13 & CD & 1D \end{pmatrix} \times \begin{pmatrix} Y2 \end{pmatrix} \\
\text{S3} &= \begin{pmatrix} 75 & 2D & CD \end{pmatrix} \\
\text{S4} &= \begin{pmatrix} 18 & 8F & 4C \end{pmatrix} \\
\text{S5} &= \begin{pmatrix} 9C & 60 & B4 \end{pmatrix}
\end{align*}
\]

If these equations are consistent and non-singular they can be solved. The solution in this case turns out to be \(Y1=56\), \(Y2=78\) and \(Y3=A7\), as expected.

These values for \(Y1\), \(Y2\) and \(Y3\) tell us that \(E(x)=56x^7+78x^6+A7x^4\) giving us the required result

\[E = 00\ 00\ 56\ 78\ 00\ A7\ 00\ 00\ 00\ 00\]

which when added to

\[R = 12\ 34\ 00\ 00\ 17\ 00\ 56\ 4B\ 78\ DD\]
gives the corrected codeword

\[T = 12\ 34\ 56\ 78\ 17\ A7\ 56\ 4B\ 78\ DD\]
4.23. REED-SOLOMON ERROR CORRECTION

It turns out that if we know the locations of 6 error, we could correct all 6. But, as is usually the case, we do not know the location of any of them we have more work to do.

The following functions calculate the syndromes polynomial and use it to confirm the accuracy the description just given.

\[
\text{AND } \text{rs_calc Syndromes}(\text{codeword, e, s}) \text{ BE}
\]

\{
// e = the number of error correction bytes
// writef("*rs_calc Syndromes:*n")
// writef("codeword: "); pr_poly(codeword)
LETS degs = e-1
s!0 := degs // The degree of the syndromes polynomial.
FOR i = 0 TO e-1 DO
{ LET p2i = gf pow(2, i)
  LET res = gf Poly_eval(codeword, p2i)
  // writef("%i2 2^-i = %x2 => %x2 %i3*n", i, p2i, res, res)
  // s!(i+1) := res // s!(i+1) = codeword(2^-i)
  s!(degs+1-i) := res // s! = codeword(2^-i)
}
\}

The typesetting of the following needs more work.

/*

Our problem is now to try and find the locations of errors in the corrupted codeword using only its syndromes polynomial and the generator polynomial.

It is common in mathematics and computing to pick out a seemingly unrelated construct, as if by magic, and after a little elementary manipulation suddenly realise it is just what we want.

Let us assume there are three locations e1, e2 and e3 containing corrupted bytes in the codeword. Let us now consider the following polynomial.

\[
\Lambda(x) = (1+x*2^-e1)(1+x*2^-e2)(1+x*2^-e3)
= 1 + L1*x + L2*x^-2 + L3*x^-3
\]

This polynomial is zero when x=2^-e1, or x=2^-e2 or x=2^-e3. If we write Xi=2^-e1, we can say the root of this \( \Lambda(x)=0 \) are \( X1^-1 \), \( X2^-1 \) and \( X3^-1 \). Knowing the roots allows us the write the following:
1 + L1*2^-ej + L2*2^-2ej + L3*2^-3ej = 0

If we multiply this equation by Yj*2^-(i+3)ej, we get

Yj*2^-3+3)ej + L1*Yj*2^-2(i+2)ej + L2*Yj*2^-2(i+1)ej + L3*Yj*2^-iej = 0

If we write these for each value of j, we get

Y1*2^-3+3)e1 + L1*Y1*2^-2(i+2)e1 + L2*Y1*2^-2(i+1)e1 + L3*Y1*2^-ie1 = 0
Y2*2^-3+3)e2 + L1*Y2*2^-2(i+2)e2 + L2*Y2*2^-2(i+1)e2 + L3*Y2*2^-ie2 = 0
Y3*2^-3+3)e3 + L1*Y3*2^-2(i+2)e3 + L2*Y3*2^-2(i+1)e3 + L3*Y3*2^-ie3 = 0

or

Y1*(2^-3+3))^e1 + L1*Y1*(2^-2(i+2))^e1 + L2*Y1*(2^-2(i+1))^e1 + L3*Y1*(2^-i)^e1 = 0
Y2*(2^-3+3))^e2 + L1*Y2*(2^-2(i+2))^e2 + L2*Y2*(2^-2(i+1))^e2 + L3*Y2*(2^-i)^e2 = 0
Y3*(2^-3+3))^e3 + L1*Y3*(2^-2(i+2))^e3 + L2*Y3*(2^-2(i+1))^e3 + L3*Y3*(2^-i)^e3 = 0

Remembering that

E(x) = Y1*x^e1 + Y2*x^e2 + Y3*x^e3

We can add these equations together giving:

E(2^-3+3) + L1*E(2^-2(i+2)) + L2*E(2^-2(i+1)) + L3*E(2^-i) = 0

We thus have the 3 following equations by setting i to 0, 1 and 2.

E(2^-3) + L1*E(2^-2) + L2*E(2^-1) + L3*E(2^-0) = 0
E(2^-4) + L1*E(2^-3) + L2*E(2^-2) + L3*E(2^-1) = 0
E(2^-5) + L1*E(2^-4) + L2*E(2^-3) + L3*E(2^-2) = 0

Since we know E(2^-i) = R(2^-i), these become:

R(2^-3) + L1*R(2^-2) + L2*R(2^-1) + L3*R(2^-0) = 0
R(2^-4) + L1*R(2^-3) + L2*R(2^-2) + L3*R(2^-1) = 0
R(2^-5) + L1*R(2^-4) + L2*R(2^-3) + L3*R(2^-2) = 0

which is the same as:

S3 + L1*S2 + L2*S1 + L3*S0 = 0
S4 + L1*S3 + L2*S2 + L3*S1 = 0
S5 + L1*S4 + L2*S3 + L3*S2 = 0

These equations can be written in matrix form as follows:
4.23. REED-SOLOMON ERROR CORRECTION

\[
\begin{align*}
(S_3) &= (S_2 S_1 S_0) \times (L_1) \\
(S_4) &= (S_3 S_2 S_1) \quad (L_2) \\
(S_5) &= (S_4 S_3 S_2) \quad (L_3)
\end{align*}
\]

Provided the 3x3 matrix is not singular, the equations can be solved giving us the values of \(L_1, L_2, \text{ and } L_3\). We now have the equation

\[
\Lambda(x) = 1 + L_1x + L_2x^2 + L_3x^3
\]

completely defined and we can therefore find its roots \(2^{-e_1}, 2^{-e_2}\) and \(2^{-e_3}\) and hence deduce the error positions \(e_1, e_2, \text{ and } e_3\). We can easily find the root by trial and error since there are only \(n\) possible values for each \(e_i\), where \(n\) is the length of the codeword.

For our example, the equations matrix equation is

\[
\begin{align*}
(6E) &= (DE 81 89) \times (L_1) \\
(82) &= (6E DE 81) \quad (L_2) \\
(7A) &= (82 6E DE) \quad (L_3)
\end{align*}
\]

giving \(L_1=\text{D}0, L_2=\text{1B} \text{ and } L_3=\text{98}\).

In general, we do not know how many errors there are. If there are fewer than 3 the 3x3 matrix will have a zero determinant and we will have to try for 2 errors, but if the top left 2x2 determinant is zero, we will have to try the top left 1x1 matrix.

The solution, if any, of this matrix equation is normally solved using Berlekamp-Massey algorithm, described later.

*/

AND rs_find_error_locator() BE
{ // This sets Lambda to the error locator polynomial
  // using the syndromes polynomial in S. It is only used
  // when we do not know the locations of any of the
  // error bytes, so the maximum number of error that
  // can be found is the (S!0+1)/2. It uses the
  // Berlekamp-Massey algorithm.
  LET old_loc = VEC 50
  LET degs = S!0
  LET k, l = 1, 0
  LET newL = VEC 50 // To hold the error locator polynomial

LET C = VEC 50  // To hold a correction polynomial
LET P1 = VEC 50

//writef("*nComputing the error locator polynomial Lambda*n")
//writef("using the Berlekamp-Massey algorithm.*n")

Lambda!0, Lambda!1 := 0, 1  // Polynomial: Lambda(x) = 1
C!0, C!1, C!2 := 1, 1, 0  // Polynomial: C(x) = x+0

UNTIL k > degs+1 DO // degs+1 = number of correction bytes
{ LET delta = 0//S!(degs+1) // S0 = R(2^0)
  LET degL = Lambda!0
 (()=>newline())
  //writef("Lambda: "); pr_poly(Lambda)
  //writef("R: "); pr_poly(R)
  //writef("S: "); pr_poly(S)
  //writef("k=%n l=%n*n", k, l)

  // First calculate delta
  FOR i = 0 TO l DO
    { LET Li = Lambda!(degL+1-i) // Li -- Coeff of x^i in current Lambda
      LET f = S!(degs+1 - (k-1-i)) // R(2^(k-1-i))
      LET Lif = gf_mul(Li, f)
      //writef("i=%n delta: %x2*n", i, delta)
      delta := delta XOR Lif
      //writef("i=%n Li=%x2 f=%x2 Lif=%x2 => delta=%x2*n",
        // i, Li, f, Lif, delta)
    }
    //writef("delta: %x2*n", delta)

    IF delta DO
      { gf_poly_scale(C, delta, P1)
        //writef("Multiply R by delta=%x2 giving: ", delta); pr_poly(P1)
        gf_poly_add(P1, Lambda, newL)
        //writef("Add L giving newL "); pr_poly(newL)
        IF 2*l < k DO
          { l := k-l
            gf_poly_scale(Lambda, gf_inverse(delta), C)
            //writef("Since 2x1 < k set C = Lambda/delta: "); pr_poly(C)
          }
        }
    }

  // Multiply C by x
  C!0 := C!0 + 1
  C!(C!0+1) := 0
4.23. REED-SOLOMON ERROR CORRECTION

// writef("Multiply C by x giving: "); pr_poly(C)

FOR i = 0 TO newL!0+1 DO Lambda!i := newL!i
// writef("Set new version of Lambda: "); pr_poly(Lambda)
k := k+1
}
}

AND rs_find_error_evaluator() BE
{ // Compute the error evaluator polynomial Omega
   // using S and Lambda.
   // Omega(x) = (S(x) * Lambda(x)) MOD x^\(e+1\)
   LET degs = S!0
   // This could be optimised since we are going to
   // through away many of the terms in the product.
   gf_poly_mul(S, Lambda, Omega)
   writef("S: "); pr_poly(S)
   writef("Lambda: "); pr_poly(Lambda)
   writef("S x Lambda: "); pr_poly(Omega)
   // Remove terms of degree higher than e
   FOR i = 0 TO degs DO Omega!(i+1) := Omega!(i+1+Omega!0-degs)
   Omega!0 := degs
   writef("Omega: "); pr_poly(Omega)
}

AND rs_demo() BE
{ // This will test Reed-Solomon decoding typically using
   // either (n,k) = (9,6) or (26,10) depending on testno.
   LET v = getvec(1000)

   writef("reedsolomon entered*n")

   S := v // For the syndromes polynomial
   M := v + 100 // For the codeword for msg
   R := v + 200 // For the corrupted codeword
   G := v + 300 // For the generator polynomial
   Lambda := v + 400 // For the erasures polynomial
   Ldash := v + 500 // For d/dx of Lambda
   Omega := v + 600 // For the evaluator polynomial
   e_pos := v + 700 // For the error positions
T := v + 800 // temp polynomial

// A simple test
Msg := TABLE 3, #x12, #x34, #x56, #x78
e := 6

IF testno>0 DO
{ // A larger test from the QR barcode given above.
  Msg := TABLE 15, #x40, #xD2, #X75, #x47, #x76, #x17, #x32, #x06, #x27, #x26, #x96, #xC6, #xC6, #x96, #x70, #xEC
  e := 10
}

k := Msg!0 + 1 // Message bytes
n := k+e // codeword bytes

gf_generator_poly(e, G) // Compute the generator polynomial
newline()
//writef("generator: "); pr_poly(G) // 01 3F 01 DA 20 E3 26
//newline()
writef("message: "); pr_poly(Msg) // 12 34 56 78

rs_encode_msg() // Compute in R the RS codeword for Msg.

writef("codeword: "); pr_poly(M) // 12 34 56 78 17 A7 56 4B 78 DD
FOR i = 0 TO M!0+1 DO R!i := M!i
R!3 := 0
R!4 := #xAA
R!6 := 0
IF testno>0 DO
{ // Try 5 errors in all
  R!12 := 0
  R!26 := 0
}
newline()
writef("corrupted: "); pr_poly(R) // 12 34 00 00 17 00 56 4B 78 DD
rs_calc_syndromes(R, e, S) // syndromes of polynomial R

writef("syndromes: "); pr_poly(S) // 7A 82 6E DE 81 89

// Typically: Lambda(x) = L3**x^3 + L2**x^2 + L1**x^1 + 1

writef("*nLambda(x) = ")
FOR i = e/2 TO 1 BY -1 DO
  writef("L%d**x^n + ", i, i)
writef("1*n")

writef("*nIt can be shown that:*n+n")

FOR row = 0 TO e/2-1 DO
{ writef("( S%n ) ", row+e/2)
  wrch(row=0 -> '=' , '')
  writef(" (")
  FOR col = e/2-1 TO 0 BY -1 DO writef(" S%n", col+row)
  writef(" )")
  wrch(row=0 -> 'x', '')
  writef(" ( L%n )*n", row+1)
}

ewline()
writef("*nwhere ")

FOR i = e-1 TO 0 BY -1 DO writef("S%n ", i)
writef("= ")

FOR i = e-1 TO 0 BY -1 DO writef("%x2 ", gf_poly_eval(R, gf_exp2!i))
writef("*n+n")

FOR row = 0 TO e/2-1 DO
{ writef("( %x2 ) ", gf_poly_eval(R, gf_exp2!(row+e/2)))
  wrch(row=0 -> '=' , '')
  writef(" (")
  FOR col = e/2-1 TO 0 BY -1 DO writef(" %x2", gf_poly_eval(R, gf_exp2!(col+row)))
  writef(" )")
  wrch(row=0 -> 'x', '')
  writef(" ( L%n )*n", row+1)
}

newline()
writef("*nThis can be solved using the Berlekamp-Massey algorithm.*n")

rs_find_error_locator()
writef("*nLambda: "); pr_poly(Lambda) // 98 1B D0 01

writef("So ")
FOR i = 1 TO e/2 DO writef(" L%n=%x2", i, Lambda!(e/2+1-i))
writef("*nand")
FOR i = 0 TO e-1 DO writef(" S%n=%x2", i, S!(S!0+1-i))
writef("*n+n")
FOR row = 0 TO e/2-1 DO
{ LET a = 0
  FOR i = 0 TO e/2-1 DO
  { LET b = gf_poly_eval(R, gf_exp2!(e/2-i+row))
    LET c = Lambda!(Lambda!0-i)
    a := a XOR gf_mul(b,c)
    writef("%x2**%x2", b,c)
    TEST i=e/2-1
    THEN writef(" = %x2 -- S%n = %x2*n",
      a, e/2+row, gf_poly_eval(R, gf_exp2!(e/2+row)))
    ELSE writef(" + ")
  }
}
writef("If the coeff of x^i in R(x) is corrupt then*
     * Lambda(2^-i) should be zero.*n")
writef("The solutions of Lambda(x)=0 can be solved by trial and error*n")
e_pos!0 := -1 // No error positions yet found.
FOR i = 0 TO R!0 DO
{ LET Xi = gf_exp2!i
  LET a = gf_poly_eval(Lambda, gf_inverse(Xi))
  IF a=0 DO
  { writef("Lambda(2^-%i2) = 0*n", i)
    e_pos!0 := e_pos!0+1
    e_pos!(e_pos!0+1) := i
  }
}
writef("So the error locations numbered from the left are: ")
pr_poly_dec(e_pos)
newline()

rs_find_error_evaluator(S, Lambda, Omega)
newline()
writef("Checking Omega*n")
FOR row = 0 TO e/2-1 DO
{ LET sum = 0
  writef("%x2", row, Omega!(Omega!0+1-row))
  FOR i = 0 TO row DO
  { LET Li = Lambda!(Lambda!0+1-i)
    LET Sj = S!(S!0+1+i-row)
    sum := sum XOR gf_mul(Li,Sj)
  }
  writef(" = %x2", sum, row)
}
newline()
4.23. REED-SOLOMON ERROR CORRECTION

IF i>0 DO writef(" + ")
writef("%x2**%x2", Sj, Li)
sum := sum XOR gf_mul(Sj,Li)
}
writef(" = %x2*n", sum)
}
newline()

writef("Lambda: "); pr_poly(Lambda)

writef("The formal differential of Lambda(x) is*n*n")
writef("Ldash(x) = L1 + 2**L2*x^1 + 3**L3*x^2 + *
*4**L4*x^3 + 5**L5*x^4 + ...*n")
writef("but here 2=1+1=0, 3=1+1+1=1, 4=1+1+1+1=0, etc, so:*n")
writef("Ldash(x) = L1 + L3*x^2 + L5*x^4 + ...*n*n")

gf_poly_copy(Lambda, Ldash)
// Clear the coefficients of the even powers
FOR i = Ldash!0+1 TO 1 BY -2 DO Ldash!i := 0
// Divide through by x
Ldash!0 := Ldash!0 - 1
writef("Ldash: "); pr_poly(Ldash)

writef("*nLet Xi = 2^i and invXi = 2^-i*n")
writef("*nIf Lambda(invXi) = 0, i will correspond to*
* the position of an error in R*n*n")

writef("To correct the coefficient at this position*
* we subtract Yi defined as follows:*n")
writef("Yi = Xi ** Omega(invXi) / Ldash(invXi)*n")

newline()
FOR i = 0 TO R!0 DO
{ LET j = R!0 + 1 - i // Position in R counting from the left.
  LET Xi = gf_exp2!i
  LET invXi = gf_inverse(Xi)
  LET LambdaInvXi = gf_poly_eval(Lambda, invXi)
  IF LambdaInvXi = 0 DO
    { LET OmegaInvXi = gf_poly_eval(Omega, invXi)
      LET LdashInvXi = gf_poly_eval(Ldash, invXi)
      LET q = gf_div(OmegaInvXi, LdashInvXi)
      LET Yi = gf_mul(Xi, q)
      writef("j=%i2 i=%i2 Xi=%x2 invXi=%x2 OmegaInvXi=%x2*
* LdashInvXi=%x2 q=%x2 Yi=%x2*n",

```
j, i, Xi, invXi, OmegaInvXi, LdashInvXi, q, Yi)
writef("So add %x2 to %x2 at position %i2 in R to give %x2*n*n",
     Yi, R!j, j, R!j XOR Yi)
R!j := R!j XOR Yi // Subtract Yi
}
}
newline()
writef("Corrected R: "); pr_poly(R)
writef("Original M: "); pr_poly(M)
freevec(v)
}

AND start() = VALOF
{ LET argv = VEC 50

    UNLESS rdargs("testno/n", argv, 50) DO
    { writef("*nBad arguments for qr*n")
    RESULTIS 0
    }

testno := 0
IF argv!0 DO testno := !argv!0 // testno/n

newline()
initlogs()
rs_demo()

IF gf_log2 DO freevec(gf_log2)
IF gf_exp2 DO freevec(gf_exp2)

RESULTIS 0
}

/
The following shows the compilation and execution of this program.
For the larger example use: reedsolomon 1

solestreet:$ cintsys

BCPL 32-bit Cintcode System (21 Oct 2015)
4.23. REED-SOLOMON ERROR CORRECTION

BCPL (10 Oct 2014) with simple floating point
Code size = 5096 bytes of 32-bit little ender Cintcode

0.070> reedsolomon

message: 12 34 56 78
generator: 01 3F 01 DA 20 E3 26

initial M: 12 34 56 78 00 00 00 00 00 00
scaled G: 12 A9 12 88 7A 4D 16
new M: 12 9D 44 F0 7A 4D 16 00 00 00
scaled G: 9D 07 9D 3F 4A 51 23
new M: 12 9D 43 6D 45 07 47 23 00 00
scaled G: 43 3A 43 F7 88 5A 1F
new M: 12 9D 43 57 06 F0 CF 79 1F 00
scaled G: 57 11 57 99 32 67 DD
new M: 12 9D 43 57 17 A7 56 4B 78 DD
codeword: 12 34 56 78 17 A7 56 4B 78 DD
corrupted: 12 34 00 AA 17 00 56 4B 78 DD
syndromes: 4B 7D 8B BD 54 23

\[ \Lambda(x) = L^3 x^n + L^2 x^n + L^1 x^n + 1 \]

It can be shown that:

\[
\begin{align*}
(S3) &= (S2 S1 S0) \times (L1) \\
(S4) &= (S3 S2 S1) \times (L2) \\
(S5) &= (S4 S3 S2) \times (L3)
\end{align*}
\]

where \( S5 S4 S3 S2 S1 S0 = 4B 7D 8B BD 54 23 \)

\[
\begin{align*}
(8B) &= (BD 54 23) \times (L1) \\
(7D) &= (8B BD 54) \times (L2) \\
(4B) &= (7D 8B BD) \times (L3)
\end{align*}
\]

This can be solved using the Berlekamp-Massey algorithm.

\[
\Lambda: 98 1B D0 01
\]
So \( L_1 = D_0 \) \( L_2 = 1B \) \( L_3 = 98 \)
and \( S_0 = 23 \) \( S_1 = 54 \) \( S_2 = BD \) \( S_3 = 8B \) \( S_4 = 7D \) \( S_5 = 4B \)

\[
\begin{align*}
BD \cdot D_0 + 54 \cdot 1B + 23 \cdot 98 & = 8B \quad -- \quad S_3 = 8B \\
8B \cdot D_0 + BD \cdot 1B + 54 \cdot 98 & = 7D \quad -- \quad S_4 = 7D \\
7D \cdot D_0 + 8B \cdot 1B + BD \cdot 98 & = 4B \quad -- \quad S_5 = 4B
\end{align*}
\]

If the coeff of \( x^i \) in \( R(x) \) is corrupt then \( \Lambda(2^{-i}) \) should be zero.

The solutions of \( \Lambda(x) = 0 \) can be solved by trial and error:

\[
\begin{align*}
\Lambda(2^{-4}) & = 0 \\
\Lambda(2^{-6}) & = 0 \\
\Lambda(2^{-7}) & = 0
\end{align*}
\]

So the error locations numbered from the left are: \( 4 \ 6 \ 7 \)

\[
\begin{array}{c}
S: & 4B & 7D & 8B & BD & 54 & 23 \\
\Lambda: & 98 & 1B & D0 & 01 \\
S \cdot \Lambda: & C8 & 5B & D8 & 00 & 00 & 0F & 26 & 23 \\
\Omega: & 00 & 00 & 00 & 0F & 26 & 23
\end{array}
\]

Checking \( \Omega \):

\[
\begin{align*}
00 & = 23 \quad 23 \cdot 01 & = 23 \\
01 & = 26 \quad 54 \cdot 01 + 23 \cdot D0 & = 26 \\
02 & = 0F \quad BD \cdot 01 + 54 \cdot D0 + 23 \cdot 1B & = 0F
\end{align*}
\]

\[
\begin{array}{c}
\Lambda: & 98 & 1B & D0 & 01 \\
The formal differential of \( \Lambda(x) \) is:
\end{array}
\]

\[
\begin{align*}
\Lambda'(x) & = L_1 + 2 \cdot L_2 \cdot x^1 + 3 \cdot L_3 \cdot x^2 + 4 \cdot L_4 \cdot x^3 + 5 \cdot L_5 \cdot x^4 + \ldots \\
\text{but here } 2 & = 1 + 1 = 0, \quad 3 & = 1 + 1 + 1 = 1, \quad 4 & = 1 + 1 + 1 + 1 = 0, \text{ etc, so:}
\end{align*}
\]

\[
\begin{align*}
\Lambda'(x) & = L_1 + L_3 \cdot x^2 + L_5 \cdot x^4 + \ldots
\end{align*}
\]

\[
\begin{array}{c}
\Lambda': & 98 & 00 & D0 \\
Let \( X_i = 2^{-i} \) and \( invX_i = 2^{-i} \)
\end{array}
\]

If \( \Lambda(invX_i) = 0 \), \( i \) will correspond to the position of an error in \( R \)

To correct the coefficient at this position we subtract \( Y_i \) defined as follows:

\[
Y_i = X_i \cdot \Omega(invX_i) / \Lambda'(invX_i)
\]

\[
\begin{align*}
j & = 6 \quad i = 4 \quad X_i = 10 \quad invX_i = D8 \quad \Omega(invX_i) = 09 \quad \Lambda'(invX_i) = EA \quad q = 38 \quad Y_i = A7
\end{align*}
\]
So add A7 to 00 at position 6 in R to give A7

\[ j = 4 \quad i = 6 \quad X_i = 40 \quad \Omega X_i = 36 \quad L \dash X_i = B8 \quad q = 28 \quad Y_i = D2 \]
So add D2 to AA at position 4 in R to give 78

\[ j = 3 \quad i = 7 \quad X_i = 80 \quad \Omega X_i = 1B \quad L \dash X_i = D8 \quad q = 79 \quad Y_i = 56 \]
So add 56 to 00 at position 3 in R to give 56

Corrected R: 12 34 56 78 17 A7 56 4B 78 DD
Original M: 12 34 56 78 17 A7 56 4B 78 DD
0.020>
*/

### 4.24 The Queens Problem

A well known problem is to count the number of different ways in which eight queens can be placed on an $$8 \times 8$$ chess board without any two of them sharing the same row, column or diagonal. It was, for instance, used as a case study in Niklaus Wirth’s classic paper “Program development by stepwise refinement” published in the Communications of the ACM in 1971. None of his solutions used either recursion or bit pattern techniques.

The following program solves a slight generalisation of the problem for board sizes from $$1 \times 1$$ to $$12 \times 12$$. 
GET "libhdr"

GLOBAL {
    count:ug
    all
}

LET try(ld, col, rd) BE
    TEST row=all
    THEN count := count + 1
    ELSE { LET poss = all & ~(ld | col | rd)
        WHILE poss DO
            { LET p = poss & ~poss
                poss := poss - p
                try(ld+p << 1, col+p, rd+p >> 1)
            }
        }
    }

LET start() = VALOF
    { all := 1
    FOR i = 1 TO 12 DO
        { count := 0
            try(0, 0, 0)
            writef("Number of solutions to %i2-queens is %i9*n", i, count)
            all := 2*all + 1
        }
    }

RESULTIS 0
}

The program performs a walk over a complete tree of valid (partial) board positions, incrementing count whenever a complete solution is found. The root of the tree is said to be at level 0 representing the empty board. The root has successors (or children) corresponding to the board states with one queen placed in the bottom row. These are all said to be at level 1. Each level 1 state has successors corresponding to valid board states with queens placed in the bottom two rows. In general, any valid board state at level \(i\) \((i > 0)\) contain \(i\) queens in the bottom \(i\) rows and is a successor of a board state at level \(i - 1\). The solutions to the \(n\)-queens problem are the valid board states at level \(n\) when all \(n\) queens have been validly placed. Ignoring symmetries, all these solutions are be distinct.

The walk over the tree of valid board states can be done without actually building the tree. It is done using the function try whose arguments \(ld\), \(col\) and \(rd\) contain sufficient information about the current board state for its successors to be explored. Figure 4.5 illustrated how \(ld\), \(col\) and \(rd\) are used to find where a queen can be validly placed in the current row without being attacked by any queen placed in earlier rows. \(col\) is a bit pattern containing a one in for each column that is already occupied. \(ld\) contains a one for each position attacked along a left going diagonal, while \(rd\) contains diagonal attacks from the other diagonal. The expression \((ld \mid col \mid rd)\) is a bit pattern containing ones in
4.24. THE QUEENS PROBLEM

all positions that are under attack from anywhere. When this is complemented and masked with all, a bit pattern is formed that gives the positions in the current row where a queen can be placed without being attacked. The variable poss is given this as its initial value by the declaration:

\[
\text{LET } \text{poss} = \neg (\text{ld} \mid \text{col} \mid \text{rd}) \& \text{all}
\]

The WHILE loop cunningly iterates over these possible placements, only executing the body of the loop as many times as needed. Notice that the expression \(\text{poss} \& -\text{poss}\) yields the least significant one in poss, as is shown in the following example.

\[
\begin{array}{c}
\text{poss} & 00100010 \\
-\text{poss} & 11011110 \\
\hline
\text{poss} \& -\text{poss} & 00000010
\end{array}
\]

The position of a valid queen placement is held in \text{bit} and removed from \text{poss} by:

\[
\text{LET } \text{bit} = \text{poss} \& -\text{poss} \\
\text{poss} := \text{poss} - \text{bit}
\]

and then a recursive call of \text{try} is made to explore the selected successor state.

\[\text{try( (ld|bit)<<1, col|bit, (rd|bit)>>1 )}\]
Notice that a left shift is needed for the left going diagonal attacks and a right shift for the other diagonal attacks.

When `col=all` a complete solution has been found and so the count of solutions is incremented.

The main function `start` calls `try` to solve the $n$-queens problem for $1 \leq n \leq 12$. The output is as follows:

- Number of solutions to 1-queens is 1
- Number of solutions to 2-queens is 0
- Number of solutions to 3-queens is 0
- Number of solutions to 4-queens is 2
- Number of solutions to 5-queens is 10
- Number of solutions to 6-queens is 4
- Number of solutions to 7-queens is 40
- Number of solutions to 8-queens is 92
- Number of solutions to 9-queens is 352
- Number of solutions to 10-queens is 724
- Number of solutions to 11-queens is 2680
- Number of solutions to 12-queens is 14200

### 4.25 Sudoku

This section presents a program to solve the sudoku puzzles which appear in most newspapers. The logic of the program is rather similar to that of the n-queens program given in the previous section. It just attempts to fill in the cells with valid digits from left to right and top to bottom, backtracking when necessary. As with the queens program, it gains some efficiency by using bit pattern techniques. This rather naive approach usually finds solutions quickly and so a faster algorithm is hardly worth implementing (but might be fun to attempt). The program is called `sudoku.b` and hopefully has sufficient comments to make it understandable without additional description.

```plaintext
// This is a really naive program to solve Su Doku problems
// as set in many newspapers.

// Implemented in BCPL by Martin Richards (c) January 2005

// Modified 4 August 2014

// It consists of a 9x9 grid of cells. Each cell should contain
// a digit in the range 1..9. Every row, column and major 3x3
// square should contain all the digits 1..9. Some cells have
// given values. The problem is to find digits to place in
// the unspecified cells satisfying the constraints.

// A typical problem is:
```
4.25. SUDOKU

// - - - 6 3 8 - - -
// 7 - 6 - - - 3 - 5
// - 1 - - - - - 4 -

// - - 8 7 1 2 4 - -
// - 9 - - - - 5 -
// - - 2 5 6 9 1 - -

// - 3 - - - - 1 -
// 1 - 5 - - - 6 - 8
// - - - 1 8 4 - - -

// The above problem is solved by the command:

// sudoku 000638000 706000305 010000040 -- all on one line
// 008712400 090000050 002569100
// 030000010 105000608 000184000

SECTION "sudoku"

GET "libhdr"

GLOBAL { count:ug

// The 9x9 board consisting of 81 cells

a1; a2; a3; a4; a5; a6; a7; a8; a9
b1; b2; b3; b4; b5; b6; b7; b8; b9
c1; c2; c3; c4; c5; c6; c7; c8; c9
d1; d2; d3; d4; d5; d6; d7; d8; d9
e1; e2; e3; e4; e5; e6; e7; e8; e9
f1; f2; f3; f4; f5; f6; f7; f8; f9
g1; g2; g3; g4; g5; g6; g7; g8; g9
h1; h2; h3; h4; h5; h6; h7; h8; h9
i1; i2; i3; i4; i5; i6; i7; i8; i9

rowabits; col1bits; squ1bits
rowbbits; col2bits; squ2bits
rowcbits; col3bits; squ3bits
rowdbits; col4bits; squ4bits
rowebits; col5bits; squ5bits
rowfbits; col6bits; squ6bits
rowgbits; col7bits; squ7bits
rowhbits; col8bits; squ8bits
MANIFEST {
N1 = #b_000000001 // Bit patterns representing the 9 digits
N2 = #b_000000010
N3 = #b_000000100
N4 = #b_000001000
N5 = #b_000010000
N6 = #b_000100000
N7 = #b_001000000
N8 = #b_010000000
N9 = #b_100000000
All = N1+N2+N3+N4+N5+N6+N7+N8+N9
}

LET start() = VALOF
{ LET argv = VEC 50
   LET r1 = 000_638_000 // The default board setting
   LET r2 = 706_000_305
   LET r3 = 010_000_040
   LET r4 = 008_712_400
   LET r5 = 090_000_050
   LET r6 = 002_569_100
   LET r7 = 030_000_010
   LET r8 = 105_000_608
   LET r9 = 000_184_000

   //LET r1 = 000_000_000 // This version of row 1 gives 14 solutions
   //LET r9 = 000_000_000 // This version of row 9 gives 46 solutions
   // If both row 1 and row 9 are all zeroes
   // there are 2096 solutions.

   UNLESS rdargs("r1/n,r2/n,r3/n,r4/n,r5/n,r6/n,r7/n,r8/n,r9/n",
      argv, 50) DO
{ writef("Bad arguments for SUDOKU*n")
   RESULTIS 0
   }
   IF argv!0 DO
{ // Set the board from the arguments
   r1,r2,r3,r4,r5,r6,r7,r8,r9 := 0,0,0,0,0,0,0,0,0
   IF argv!0 DO r1 := !(argv!0)
4.25. SUDOKU

IF argv!1 DO r2 := !(argv!1)
IF argv!2 DO r3 := !(argv!2)
IF argv!3 DO r4 := !(argv!3)
IF argv!4 DO r5 := !(argv!4)
IF argv!5 DO r6 := !(argv!5)
IF argv!6 DO r7 := !(argv!6)
IF argv!7 DO r8 := !(argv!7)
IF argv!8 DO r9 := !(argv!8)
}

initboard(r1,r2,r3,r4,r5,r6,r7,r8,r9)
writef("*nInitial board*n")
prboard()
count := 0
ta1()
writef("*n*nTotal number of solutions: %n*n", count)
RESULTIS 0
}

AND setrow(row, r) BE
{ LET tab = TABLE 0, N1, N2, N3, N4, N5, N6, N7, N8, N9
  FOR i = 8 TO 0 BY -1 DO
  { LET n = r MOD 10
    r := r/10
    row!i := tab!n
  }
}

AND initboard(r1,r2,r3,r4,r5,r6,r7,r8,r9) BE
{ // Give all 81 cells their initial settings
  setrow(@a1, r1)
  setrow(@b1, r2)
  setrow(@c1, r3)
  setrow(@d1, r4)
  setrow(@e1, r5)
  setrow(@f1, r6)
  setrow(@g1, r7)
  setrow(@h1, r8)
  setrow(@i1, r9)
  // Initialise row bit patterns
  rowabits := a1+a2+a3+a4+a5+a6+a7+a8+a9
  rowbbits := b1+b2+b3+b4+b5+b6+b7+b8+b9
  rowcbits := c1+c2+c3+c4+c5+c6+c7+c8+c9
  rowdbits := d1+d2+d3+d4+d5+d6+d7+d8+d9
}
rowebits := e1+e2+e3+e4+e5+e6+e7+e8+e9
rowfbits := f1+f2+f3+f4+f5+f6+f7+f8+f9
rowgbits := g1+g2+g3+g4+g5+g6+g7+g8+g9
rowhbits := h1+h2+h3+h4+h5+h6+h7+h8+h9
rowibits := i1+i2+i3+i4+i5+i6+i7+i8+i9

// Initialise column bit patterns
col1bits := a1+b1+c1+d1+e1+f1+g1+h1+i1
col2bits := a2+b2+c2+d2+e2+f2+g2+h2+i2
col3bits := a3+b3+c3+d3+e3+f3+g3+h3+i3
col4bits := a4+b4+c4+d4+e4+f4+g4+h4+i4
col5bits := a5+b5+c5+d5+e5+f5+g5+h5+i5
col6bits := a6+b6+c6+d6+e6+f6+g6+h6+i6
col7bits := a7+b7+c7+d7+e7+f7+g7+h7+i7
col8bits := a8+b8+c8+d8+e8+f8+g8+h8+i8
col9bits := a9+b9+c9+d9+e9+f9+g9+h9+i9

// Initialise the 3x3 square bit patterns
squ1bits := a1+a2+a3 + b1+b2+b3 + c1+c2+c3
squ2bits := a4+a5+a6 + b4+b5+b6 + c4+c5+c6
squ3bits := a7+a8+a9 + b7+b8+b9 + c7+c8+c9
squ4bits := d1+d2+d3 + e1+e2+e3 + f1+f2+f3
squ5bits := d4+d5+d6 + e4+e5+e6 + f4+f5+f6
squ6bits := d7+d8+d9 + e7+e8+e9 + f7+f8+f9
squ7bits := g1+g2+g3 + h1+h2+h3 + i1+i2+i3
squ8bits := g4+g5+g6 + h4+h5+h6 + i4+i5+i6
squ9bits := g7+g8+g9 + h7+h8+h9 + i7+i8+i9
}

AND try(p, f, rptr, cptr, sptr) BE TEST !p
THEN f() // The cell pointed to by p is already set
    // so move on to the next cell, if any.
ELSE { LET r, c, s = !rptr, !cptr, !sptr
    // r, c and s are bit patterns indicating which digits
    // already occupy the current row, column or square.
    LET poss = All - (r | c | s)
    // poss is a bit pattern indicating which digits can
    // be placed in the current cell.
    WHILE poss DO
        { // Try each allowable digit in turn.
            LET bit = poss & ~poss
            poss := poss-bit
            // Update the cell, row, column and square bit patterns.
            !p, !rptr, !cptr, !sptr := bit, r+bit, c+bit, s+bit
            // Move on to the next cell, if any.
f()
}

// Restore the cell, row, column and square bit patterns.
p, !rptr, !cptr, !sptr := 0, r, c, s
}

// The following 81 functions try all possible settings for
// each cell on the board.
AND ta1() BE try(@a1, ta2, @rowbits, @col1bits, @squ1bits)
AND ta2() BE try(@a2, ta3, @rowbits, @col2bits, @squ1bits)
AND ta3() BE try(@a3, ta4, @rowbits, @col3bits, @squ1bits)
AND ta4() BE try(@a4, ta5, @rowbits, @col4bits, @squ2bits)
AND ta5() BE try(@a5, ta6, @rowbits, @col5bits, @squ2bits)
AND ta6() BE try(@a6, ta7, @rowbits, @col6bits, @squ2bits)
AND ta7() BE try(@a7, ta8, @rowbits, @col7bits, @squ3bits)
AND ta8() BE try(@a8, ta9, @rowbits, @col8bits, @squ3bits)
AND ta9() BE try(@a9, tb1, @rowbits, @col9bits, @squ3bits)

AND tb1() BE try(@b1, tb2, @rowbits, @col1bits, @squ1bits)
AND tb2() BE try(@b2, tb3, @rowbits, @col2bits, @squ1bits)
AND tb3() BE try(@b3, tb4, @rowbits, @col3bits, @squ1bits)
AND tb4() BE try(@b4, tb5, @rowbits, @col4bits, @squ2bits)
AND tb5() BE try(@b5, tb6, @rowbits, @col5bits, @squ2bits)
AND tb6() BE try(@b6, tb7, @rowbits, @col6bits, @squ2bits)
AND tb7() BE try(@b7, tb8, @rowbits, @col7bits, @squ3bits)
AND tb8() BE try(@b8, tb9, @rowbits, @col8bits, @squ3bits)
AND tb9() BE try(@b9, tc1, @rowbits, @col9bits, @squ3bits)

AND tc1() BE try(@c1, tc2, @rowbits, @col1bits, @squ1bits)
AND tc2() BE try(@c2, tc3, @rowbits, @col2bits, @squ1bits)
AND tc3() BE try(@c3, tc4, @rowbits, @col3bits, @squ1bits)
AND tc4() BE try(@c4, tc5, @rowbits, @col4bits, @squ2bits)
AND tc5() BE try(@c5, tc6, @rowbits, @col5bits, @squ2bits)
AND tc6() BE try(@c6, tc7, @rowbits, @col6bits, @squ2bits)
AND tc7() BE try(@c7, tc8, @rowbits, @col7bits, @squ3bits)
AND tc8() BE try(@c8, tc9, @rowbits, @col8bits, @squ3bits)
AND tc9() BE try(@c9, td1, @rowbits, @col9bits, @squ3bits)

AND td1() BE try(@d1, td2, @rowbits, @col1bits, @squ4bits)
AND td2() BE try(@d2, td3, @rowbits, @col2bits, @squ4bits)
AND td3() BE try(@d3, td4, @rowbits, @col3bits, @squ4bits)
AND td4() BE try(@d4, td5, @rowbits, @col4bits, @squ5bits)
AND td5() BE try(@d5, td6, @rowbits, @col5bits, @squ5bits)
AND td6() BE try(@d6, td7, @rowbits, @col6bits, @squ5bits)
AND td7() BE try(@d7, td8, @rowbits, @col7bits, @squ6bits)
AND td8() BE try(@d8, td9, @rowdbits, @col8bits, @squ6bits)
AND td9() BE try(@d9, te1, @rowdbits, @col9bits, @squ6bits)

AND te1() BE try(@e1, te2, @rowebits, @col1bits, @squ4bits)
AND te2() BE try(@e2, te3, @rowebits, @col2bits, @squ4bits)
AND te3() BE try(@e3, te4, @rowebits, @col3bits, @squ4bits)
AND te4() BE try(@e4, te5, @rowebits, @col4bits, @squ5bits)
AND te5() BE try(@e5, te6, @rowebits, @col5bits, @squ5bits)
AND te6() BE try(@e6, te7, @rowebits, @col6bits, @squ5bits)
AND te7() BE try(@e7, te8, @rowebits, @col7bits, @squ6bits)
AND te8() BE try(@e8, te9, @rowebits, @col8bits, @squ6bits)
AND te9() BE try(@e9, tf1, @rowebits, @col9bits, @squ6bits)

AND tf1() BE try(@f1, tf2, @rowfbits, @col1bits, @squ4bits)
AND tf2() BE try(@f2, tf3, @rowfbits, @col2bits, @squ4bits)
AND tf3() BE try(@f3, tf4, @rowfbits, @col3bits, @squ4bits)
AND tf4() BE try(@f4, tf5, @rowfbits, @col4bits, @squ5bits)
AND tf5() BE try(@f5, tf6, @rowfbits, @col5bits, @squ5bits)
AND tf6() BE try(@f6, tf7, @rowfbits, @col6bits, @squ5bits)
AND tf7() BE try(@f7, tf8, @rowfbits, @col7bits, @squ6bits)
AND tf8() BE try(@f8, tf9, @rowfbits, @col8bits, @squ6bits)
AND tf9() BE try(@f9, tg1, @rowfbits, @col9bits, @squ6bits)

AND tg1() BE try(@g1, tg2, @rowgbits, @col1bits, @squ7bits)
AND tg2() BE try(@g2, tg3, @rowgbits, @col2bits, @squ7bits)
AND tg3() BE try(@g3, tg4, @rowgbits, @col3bits, @squ7bits)
AND tg4() BE try(@g4, tg5, @rowgbits, @col4bits, @squ8bits)
AND tg5() BE try(@g5, tg6, @rowgbits, @col5bits, @squ8bits)
AND tg6() BE try(@g6, tg7, @rowgbits, @col6bits, @squ9bits)
AND tg7() BE try(@g7, tg8, @rowgbits, @col7bits, @squ9bits)
AND tg8() BE try(@g8, tg9, @rowgbits, @col8bits, @squ9bits)
AND tg9() BE try(@g9, th1, @rowgbits, @col9bits, @squ9bits)

AND th1() BE try(@h1, th2, @rowhbits, @col1bits, @squ7bits)
AND th2() BE try(@h2, th3, @rowhbits, @col2bits, @squ7bits)
AND th3() BE try(@h3, th4, @rowhbits, @col3bits, @squ7bits)
AND th4() BE try(@h4, th5, @rowhbits, @col4bits, @squ8bits)
AND th5() BE try(@h5, th6, @rowhbits, @col5bits, @squ8bits)
AND th6() BE try(@h6, th7, @rowhbits, @col6bits, @squ8bits)
AND th7() BE try(@h7, th8, @rowhbits, @col7bits, @squ9bits)
AND th8() BE try(@h8, th9, @rowhbits, @col8bits, @squ9bits)
AND th9() BE try(@h9, ti1, @rowhbits, @col9bits, @squ9bits)

AND ti1() BE try(@i1, ti2, @rowibits, @col1bits, @squ7bits)
AND ti2() BE try(@i2, ti3, @rowibits, @col2bits, @squ7bits)
AND ti3() BE try(@i3, ti4, @rowibits, @col3bits, @squ7bits)
AND ti4() BE try(@i4, ti5, @rowibits, @col4bits, @squ8bits)
AND ti5() BE try(@i5, ti6, @rowibits, @col5bits, @squ8bits)
AND ti6() BE try(@i6, ti7, @rowibits, @col6bits, @squ8bits)
AND ti7() BE try(@i7, ti8, @rowibits, @col7bits, @squ9bits)
AND ti8() BE try(@i8, ti9, @rowibits, @col8bits, @squ9bits)
AND ti9() BE try(@i9, suc, @rowibits, @col9bits, @squ9bits)

// suc is only called when a solution has been found.
AND suc() BE
{ count := count + 1
  writef("*nSolution number %n*n", count)
  prboard()
}

AND c(n) = VALOF SWITCHON n INTO
{ DEFAULT: RESULTIS '?'
  CASE 0: RESULTIS '-'
  CASE N1: RESULTIS '1'
  CASE N2: RESULTIS '2'
  CASE N3: RESULTIS '3'
  CASE N4: RESULTIS '4'
  CASE N5: RESULTIS '5'
  CASE N6: RESULTIS '6'
  CASE N7: RESULTIS '7'
  CASE N8: RESULTIS '8'
  CASE N9: RESULTIS '9'
}

AND prboard() BE
{ LET form = "%c %c %c %c %c %c %c %c %c%n"
  newline()
  writef(form, c(a1),c(a2),c(a3),c(a4),c(a5),c(a6),c(a7),c(a8),c(a9))
  newline()
  writef(form, c(b1),c(b2),c(b3),c(b4),c(b5),c(b6),c(b7),c(b8),c(b9))
  newline()
  writef(form, c(c1),c(c2),c(c3),c(c4),c(c5),c(c6),c(c7),c(c8),c(c9))
  newline()
  writef(form, c(d1),c(d2),c(d3),c(d4),c(d5),c(d6),c(d7),c(d8),c(d9))
  newline()
  writef(form, c(e1),c(e2),c(e3),c(e4),c(e5),c(e6),c(e7),c(e8),c(e9))
  newline()
  writef(form, c(f1),c(f2),c(f3),c(f4),c(f5),c(f6),c(f7),c(f8),c(f9))
  newline()
  writef(form, c(g1),c(g2),c(g3),c(g4),c(g5),c(g6),c(g7),c(g8),c(g9))
  newline()
  writef(form, c(h1),c(h2),c(h3),c(h4),c(h5),c(h6),c(h7),c(h8),c(h9))
  newline()
  writef(form, c(i1),c(i2),c(i3),c(i4),c(i5),c(i6),c(i7),c(i8),c(i9))
   newline()
4.26 The Sliding Blocks Puzzle

This section describes a program that explores the structure of the sliding blocks puzzle pictured below.

As can be seen, the puzzle is played on a 4x5 board on which 10 blocks can slide. There are four unit 1x1 blocks (U), four 1x2 blocks (V) oriented vertically, one 2x1 block (H) oriented horizontally and one 2x2 block (S). The initial position of the blocks is as in the picture and the aim is to slide the pieces until the 2x2 block is centred at the bottom. This takes a minimum of 84 moves, where a move is defined to be moving one block by one position up, down, left or right by one place. When the program is run it tells us there are 65880 different placements of the ten pieces of which only 25955 are reachable from the initial position.

The collection of nodes reachable from a given node is called, by mathematicians, a simply connected component, and it turns out that the sliding block puzzle has 898 of them, the largest and smallest having 25955 and 2 nodes, respectively. As we have seen, one of the components of size 25955 nodes includes the starting position.

The structure of the puzzle can be thought of as a graph with each board position represented by a node having edges to other nodes reachable by single moves. The graph is said to be undirected since every move is reversible.
Since there are only 65880 nodes in the graph the program can build the entire graph in memory and then explore it to discover its properties. As a bye product it outputs a minimum length sequence of moves to solve the puzzle.

The board is represented by a 20 bit pattern with each bit indicating the occupancy of each square on the board. The vector $\text{bitsS}$ holds bit patterns representing the 12 possible placements of the 2x2 block in $\text{bitsS}!1$ to $\text{bitsS}!12$. The upper bound, 12, is held in $\text{bitsS}!0$.

A particular placement of the 2x2 block is represented by a placement number $p$ in the range 1 to 12. The corresponding bit pattern is thus $\text{bitsS}!p$. Its immediately adjacent placement positions are held in the vector $\text{succsS}!p$. If we call this vector $v$, then $v!0=n$ is the number adjacent placements and $v!1$ to $v!n$ are their placement numbers.

The vectors $\text{bitsV}$, $\text{bitsH}$ and $\text{bitsU}$ hold, respectively, the bit patterns representing the 16 possible placements of a vertically oriented 1x2 block, the 15 possible placements of the horizontally oriented 2x1 block, and the 20 possible placements of a 1x1 block. The vectors $\text{succsV}$, $\text{succsH}$ and $\text{succsU}$ contain adjacency information for these blocks in a form similar to $\text{succsS}$.

The program starts as follows.

GET "libhdr"

MANIFEST {
  // Selectors for a placement node
  s_link=0  // link=0 or link -> another node at the dist value.
  s_dist  // dist=-1 or the distance from the starting position.
    // If dist=-1, this node has not yet been visited.
  s_prev  // prev=0 or prev -> predecessor node in the path
    // from the starting position to this node.
  s_chain  // chain=0 or chain -> another node with the same hash value.
  s_succs  // List of adjacent placement nodes.

  // succs=0 or succs -> [next, node]
  // Piece placement numbers
  s_S  // The 2x2 block
  s_Va; s_Vb; s_Vc; s_Vd  // The four 1x2 blocks
  s_H  // The 2x1 block
  s_Ua; s_Ub; s_Uc; s_Ud  // The four 1x1 blocks

  // Board placement bit patterns
  s_S1  // Positions occupied by the 2x2 piece
  s_V4  // Positions occupied by the 1x2 virtual pieces
  s_H1  // Positions occupied by the 2x1 horizontal piece
  s_U4  // Positions occupied by the 1x1 pieces
s_upb=s_U4 // The upb of a placement node
}

These **MANIFEST** constants define the fields of a placement node. The **link** field is used to link all nodes at the same distance from the starting node. This distance is held in the **dist** field with the convention that the starting node is at distance zero. The vector **listv** holds these lists with **listv!d** being the list of all nodes at distance d. The **dist** field is set to -1 in all nodes that have not yet been visited.

The program creates nodes all 65880 valid board placements and puts pointers to them in elements **nodev!1** to **nodev!65880**. The upper bound, 65880, is placed in **nodev!0**. The fields **S1**, **V4**, **H1** and **U4** hold bit patterns representing the placements of the 2x2 block, the 2x1 blocks, the 1x2 block and the 1x1 blocks. These four bit patterns uniquely represent each possible placement of the ten blocks. The placement numbers of the ten blocks are held in the **S**, **Va**, **Vb**, **Vc**, **Vd**, **H**, **Ua**, **Ub**, **Ua** and **Ua** fields.

A hash table, **hashtab**, allows efficient looking up of a placement node given its **S1**, **V4**, **H1** and **U4** settings. The call **hashtab(S1,V4,H1,U4)** computes the hash value. The pointer to the next node in a hash chain is held in the **chain** field.

All the placement nodes are created by the call **createnodes()**. The program then creates, for each placement node, the list of immediately adjacent placements. This list is held in the **succs** field. These lists are created by the call **createsuccs()** which makes calls of the form **mksuccs(node)** for every node in **nodev**.

The program next creates lists of nodes at different distances from the starting position. As we have seen, these lists are placed in the vector **listv**. They are are created by the call **createlists()**. The call **find(#x66000,#x09999,#x00006,#00660)** finds the starting node, which is given a **dist** value of zero and becomes the only node in **listv!0**. All other nodes initially have **dist** values of -1, indicating that their distances are not yet known. The list of nodes at distance d from the starting position is constructed by the call **createlists(d)** which inspects every node in **listv!(d-1)**. Each successor to these nodes, that have not be visited previously, is inserted into **listv!d**, with its **dist** field set to d and its **prev** field set to the immediate predecessor. The variable **solution** points to the first node visited that has the 2x2 block placed centrally at the bottom. This combined with the **prev** field values allows the solution to be output. If **listv!d** turns out to be empty, all reachable nodes have been visited and **createlists** returns.

The program shows that a solution can be found in 84 moves and that of the 25955 reachable board positions there are four that are most distant from the initial position taking 133 moves to reach. These positions are:
and their mirror images. No reachable position has the horizontal block in the top row.

While there are still unvisited nodes, the program goes on to find another component using any unvisited node as the starting node and calling createlists again.

The program continues as follows declaring the global variables and some more constants used in the program.

GLOBAL {
    bitsS:ug; succsS
    bitsH; succsH
    bitsV; succsV
    bitsU; succsU
    spacev; spacep; spacet
    mkvec
    mk2
    tracing
    nodev
    nodecount
    edgecount
    listv
    hashtab
    root
    componentcount
    componentsize
    componentsizemax
}
componentsizemin
componentp
solution

hashfn
find
initpieces
createnodes
createsuccs
mksuccs
explore
prboard
prsol
}

MANIFEST {
Spaceupb = 2_000_000
nodevupb = 65880
listvupb = 200
hashtabsize = 5000
}

The definition of start is as follows.

LET start() = VALOF
{ LET argv = VEC 50
  LET stdout = output()
  LET out = stdout
  UNLESS rdargs("-o/k,-t/s", argv, 50) DO
  { writef("Bad arguments for blocks*n")
    RESULTIS 20
  }

  IF argv!0 DO // -o/k
  { out := findoutput(argv!0)
    UNLESS out DO
    { writef("Unable to open output file %s*n", argv!0)
      RESULTIS 20
    }
    selectoutput(out)
  }

  tracing := argv!1 // -t/s
  solution := 0
4.26. THE SLIDING BLOCKS PUZZLE

nodecount := 0
degreecount := 0
componentcount := 0
componentsize := 0
componentsizeiemax := 0
componentsizeiemin := maxint
componentp := 0

spacev := getvec(Spaceupb)
spacep, spacet := spacev, spacev+Spaceupb

UNLESS spacev DO
{ writef("Insufficient space available\n")
  RESULTIS 20
}

hashtab := mkvec(hashtabsize-1)
FOR i = 0 TO hashtabsize-1 DO hashtab!i := 0
nodev := mkvec(nodevupb)
listv := mkvec(listvupb)
odecount := 0
solution := 0
root := 0

initpieces()
createnodes() // Create all 65880 placement nodes
createsuccs() // Create the successor list for each node

IF FALSE DO
FOR i = 1 TO nodev!0 DO
{ LET node = nodev!i
  LET succs = s_succs!node
  writef("node %i7: ", i)
  prboard(s_S1!node, s_V4!node, s_H1!node, s_U4!node)
  //writef("*nsuccs: ")
  //WHILE succs DO
  //  { writef(" %i5", succs!1)
  //   succs := succs!0
  //  }
  newline()
  succs := s_succs!node
  WHILE succs DO
  { LET succ = succs!1
       writef("succ %i7: ", succ)
       prboard(s_S1!succ, s_V4!succ, s_H1!succ, s_U4!succ)
  }
newline()
succs := succs!0
}
//abort(1000)
}

explore()

// Lists of nodes at all distances have now been created
// so output the solution

IF solution DO prsol(solution)

writef("nodecount= %n*n", nodecount)
writef("edgecount= %n*n", edgecount)
writef("componentcount= %n*n", componentcount)
writef("componentsizemax= %n*n", componentsizemax)
writef("componentsizemin= %n*n", componentsizemin)
writef("space used = %n words*n", spacep-spacev)

fin:
UNLESS out=stdout DO endwrite()
freevec(spacev)
RESULTIS 0
}

The program continues as follows.

AND mkvec(upb) = VALOF
{ LET p = spacep
  spacep := spacep+upb+1
  IF spacep>spacet DO
  { writef("Insufficient space*n")
    abort(999)
    RESULTIS 0
  }
  //writef("mkvec(%n) => %n*n", upb, p)
  RESULTIS p
  }

AND mk2(a, b) = VALOF
{ LET p = mkvec(1)
  p!0, p!1 := a, b
  RESULTIS p
}
The program continues as follows.

AND mkinitvec(n, a, b, c, d) = VALOF
{ LET p = spacep
  spacep := spacep+n+1
  IF spacep>spacet DO
    { writef("Insufficient space\(n\)")
      abort(999)
      RESULTIS 0
    }
  FOR i = 0 TO n DO p!i := (@n)!i
  RESULTIS p
}

AND initpieces() BE
{ // 2x2 square block
  bitsS := TABLE 12, // placement bits
  #xC000, #x60000, #x30000, // 1 2 3
  #x0C00, #x06000, #x03000, // 4 5 6
  #x00C0, #x00600, #x00300, // 7 8 9
  #x000C, #x0006, #x0003 // 10 11 12
  succsS := mkvec(12)
  succsS! 0 := 12
  succsS! 1 := mkinitvec(2, 2, 4)
  succsS! 2 := mkinitvec(3, 1, 3, 5)
  succsS! 3 := mkinitvec(2, 2, 6)
  succsS! 4 := mkinitvec(3, 1, 5, 7)
  succsS! 5 := mkinitvec(4, 2, 4, 8)
  succsS! 6 := mkinitvec(3, 3, 5, 9)
  succsS! 7 := mkinitvec(3, 4, 8, 10)
  succsS! 8 := mkinitvec(4, 5, 7, 11)
  succsS! 9 := mkinitvec(3, 6, 8, 12)
  succsS!10 := mkinitvec(2, 7, 11)
  succsS!11 := mkinitvec(3, 8, 10, 12)
  succsS!12 := mkinitvec(2, 9, 11)

  // 1x2 vertical block
  bitsV := TABLE 16, // placement bits
  #x80000, #x40000, #x20000, #x10000, // 1 2 3 4
  #x08000, #x04000, #x02000, #x01000, // 5 6 7 8
  #x00800, #x00400, #x00200, #x00100, // 9 10 11 12
  #x00080, #x00040, #x00020, #x00010, // 13 14 15 16
  succsV := mkvec(16)
  succsV! 0 := 16
  succsV! 1 := mkinitvec(2, 2, 5)
sucsv! 2 := mkinitvec(3, 1, 3, 6)
sucsv! 3 := mkinitvec(3, 2, 4, 7)
sucsv! 4 := mkinitvec(2, 3, 8)
sucsv! 5 := mkinitvec(3, 1, 6, 9)
sucsv! 6 := mkinitvec(4, 2, 5, 7, 10)
sucsv! 7 := mkinitvec(4, 3, 6, 8, 11)
sucsv! 8 := mkinitvec(3, 4, 7, 12)
sucsv! 9 := mkinitvec(3, 5, 10, 13)
sucsv!10 := mkinitvec(4, 6, 9, 11, 14)
sucsv!11 := mkinitvec(4, 7, 10, 12, 15)
sucsv!12 := mkinitvec(3, 8, 11, 16)
sucsv!13 := mkinitvec(2, 9, 14)
sucsv!14 := mkinitvec(3, 10, 13, 15)
sucsv!15 := mkinitvec(3, 11, 14, 16)
sucsv!16 := mkinitvec(2, 12, 15)

// 2x1 horizontal block
bitsH := TABLE 15, // placement bits
    #xc0000, #x60000, #x30000, // 1 2 3
    #xc0000, #x60000, #x30000, // 4 5 6
    #xc0000, #x60000, #x30000, // 7 8 9
    #xc0000, #x60000, #x30000, // 10 11 12
    #xc0000, #x60000, #x30000 // 13 14 15

sucsh := mkvec(15)
sucsh! 0 := 15
sucsh! 1 := mkinitvec(2, 2, 4)
sucsh! 2 := mkinitvec(3, 1, 3, 5)
sucsh! 3 := mkinitvec(2, 2, 6)
sucsh! 4 := mkinitvec(3, 1, 5, 7)
sucsh! 5 := mkinitvec(4, 2, 4, 6, 8)
sucsh! 6 := mkinitvec(3, 3, 5, 9)
sucsh! 7 := mkinitvec(3, 4, 8, 10)
sucsh! 8 := mkinitvec(4, 5, 7, 9, 11)
sucsh! 9 := mkinitvec(3, 6, 8, 12)
sucsh!10 := mkinitvec(3, 7, 11, 13)
sucsh!11 := mkinitvec(4, 8, 10, 12, 14)
sucsh!12 := mkinitvec(3, 9, 11, 15)
sucsh!13 := mkinitvec(2, 10, 14)
sucsh!14 := mkinitvec(3, 11, 13, 15)
sucsh!15 := mkinitvec(2, 12, 14)

// 1x1 unit squares
bitsu := TABLE 20, // placement bits
    #x80000, #x40000, #x20000, #x10000, // 1 2 3 4
The program continues as follows.

AND addnode(s, va,vb,vc,vd, h, ua,ub,uc,ud) BE 
{ // Insert a new placement node in nodev
LET node = mkvec(s_upb)
LET S1  = bitsS!s
LET V4  = bitsV!va + bitsV!vb + bitsV!vc + bitsV!vd
LET H1  = bitsH!h
LET U4  = bitsU!ua + bitsU!ub + bitsU!uc + bitsU!ud
LET hashval = hashfn(S1, V4, H1, U4)
s_link!node := 0
s_dist!node := -1
s_prev!node := 0
s_chain!node := hashtab!hashval
hashtab!hashval := node
s_succs!node := 0}
The program continues as follows.

\[
\text{AND } hashfn(S1, V4, H, U4) = (S1 \text{ XOR } V4*5 \text{ XOR } H*7 \text{ XOR } U4*11) \text{ MOD hashtabsize}
\]

\[
\text{AND } find(S1, V4, H1, U4) = \text{VALOF}
\]

\[
\text{LET hashval = hashfn(S1, V4, H1, U4)}
\]
\[
\text{LET node = hashtab!hashval}
\]
\[
//\text{writef("find: entered, hashval=\%n*n", hashval)}
\]
\[
\text{WHILE node DO}
\]
\[
\text{IF S1=s_S1!node \&
V4=s_V4!node \&
H1=s_H1!node \&
U4=s_U4!node RESULTIS node
node := s_chain!node}
\]
\[
\text{writef("find: Failed to find "); prboard(S1,V4,H1,U4)
newline()}
\]
abort(999)
RESULT IS 0
}

The program continues as follows.

AND createnodes() BE
{ FOR s = 1 TO bitsS!0 DO
{ LET bits = bitsS!s
 FOR va = 1 TO bitsV!0 - 3 IF (bits & bitsV!va)=0 DO
{ bits := bits + bitsV!va
 FOR vb = va+1 TO bitsV!0 - 2 IF (bits & bitsV!vb)=0 DO
{ bits := bits + bitsV!vb
 FOR vc = vb+1 TO bitsV!0 - 1 IF (bits & bitsV!vc)=0 DO
{ bits := bits + bitsV!vc
 FOR vd = vc+1 TO bitsV!0 IF (bits & bitsV!vd)=0 DO
{ bits := bits + bitsV!vd
 FOR h = 1 TO bitsH!0 IF (bits & bitsH!h)=0 DO
{ bits := bits + bitsH!h
 FOR ua = 1 TO bitsU!0 - 3 IF (bits & bitsU!ua)=0 DO
{ bits := bits + bitsU!ua
 FOR ub = ua+1 TO bitsU!0 - 2 IF (bits & bitsU!ub)=0 DO
{ bits := bits + bitsU!ub
 FOR uc = ub+1 TO bitsU!0 - 1 IF (bits & bitsU!uc)=0 DO
{ bits := bits + bitsU!uc
 FOR ud = uc+1 TO bitsU!0 IF (bits & bitsU!ud)=0 DO
{ bits := bits + bitsU!ud
 addnode(s,va,vb,vc,vd,h,ua,ub,uc,ud)
 bits := bits - bitsU!ud
 }
 bits := bits - bitsU!uc
 }
 bits := bits - bitsU!ub
 }
 bits := bits - bitsU!ua
 }
 bits := bits - bitsH!h
 }
 bits := bits - bitsV!vd
 }
 bits := bits - bitsV!vc
 }
 bits := bits - bitsV!vb
 }
 bits := bits - bitsV!va
}
The program continues as follows.

AND createsuccs() BE
{ // Create the successor list for every node
  FOR i = 1 TO nodev!0 DO mksuccs(nodev!i)
}

AND mksuccs(node) BE
{ LET all = s_S1!node + s_V4!node + s_H1!node + s_U4!node
  //writef("mksuccs: node is ")
  //prboard(s_S1!node, s_V4!node, s_H1!node, s_U4!node)
  //newline()
  //abort(2000)
  mksuccsS(node, all, s_S !node)
  mksuccsV(node, all, s_Va!node)
  mksuccsV(node, all, s_Vb!node)
  mksuccsV(node, all, s_Vc!node)
  mksuccsV(node, all, s_Vd!node)
  mksuccsH(node, all, s_H !node)
  mksuccsU(node, all, s_Ua!node)
  mksuccsU(node, all, s_Ub!node)
  mksuccsU(node, all, s_Uc!node)
  mksuccsU(node, all, s_Ud!node)
  //abort(2003)
}

AND mksuccsS(p, all, q) BE
{ // all is a bit pattern giving all occupied squares
  // q is the current placement number of the 2x2 S piece
  LET succsv = succsv!q  // Vector of successors of placement q
  LET bitsq = bitsS!q    // The bit pattern for placement q
  LET bits = all - bitsq // all with placement q removed
  FOR i = 1 TO succsv!0 DO
    { LET j = succsv!i    // An adjacent placement of the 2x2 S piece
      LET bitsj = bitsS!j  // The bit pattern for placement j
      //writef("mksuccsS: q=%n i=%n j=%n bits=%x5 bitsq=%x5 bitsj=%x5*%n", q, i, j, bits, bitsq, bitsj)
      //abort(2001)
      IF (bits & bitsj) = 0 DO
        { // Found a successor
          LET S1, V4, H1, U4 = bitsj, s_V4!p, s_H1!p, s_U4!p
AND mksuccsV(p, all, q) BE
{ // all is a bit pattern giving all occupied squares
  // q is the current placement number of a 1x2 V piece
  LET succsv = succsV!q // Vector of successors of placement q
  LET bitsq = bitsV!q // The bit pattern for placement q
  LET bits = all - bitsq // all with placement q removed
  FOR i = 1 TO succsv!0 DO
    { LET j = succsv!i // An adjacent placement of the 1x2 V piece
      LET bitsj = bitsV!j // The bit pattern for placement j
      //writef("mksuccsV: q=%n i=%n j=%n bits=%x5 bitsq=%x5 bitsj=%x5*n",
        q, i, j, bits, bitsq, bitsj)
      //abort(2001)
      IF (bits & bitsj) = 0 DO
        { // Found a successor
          LET S1, V4, H1, U4 = s_S1!p, s_V4!p-bitsq+bitsj, s_H1!p, s_U4!p
          LET succ = find(S1,V4,H1,U4)
          s_succs!p := mk2(s_succs!p, succ)
          edgecount := edgecount+1
          //writef("V successor ")
          //prboard(S1,V4,H1,U4)
          //newline()
          //abort(1000)
        }
      }
    }
  }
}

AND mksuccsH(p, all, q) BE
{ // all is a bit pattern giving all occupied squares
  // q is the current placement number of the 2x1 H piece
  LET succsh = succsH!q // Vector of successors of placement q
  LET bitsq = bitsH!q // The bit pattern for placement q
  LET bits = all - bitsq // all with placement q removed
  FOR i = 1 TO succsh!0 DO
    { LET j = succsh!i // An adjacent placement of the 2x1 H piece
      LET S1, V4, H1, U4 = s_S1!p, s_V4!p-bitsq+bitsj, s_H1!p, s_U4!p
      LET succ = find(S1,V4,H1,U4)
      s_succs!p := mk2(s_succs!p, succ)
      edgecount := edgecount+1
      //writef("H successor ")
      //prboard(S1,V4,H1,U4)
      //newline()
      //abort(1000)
    }
  }
}
LET bitsj = bitsH!j  // The bit pattern for placement j
//writef("mksuccsH: q=%n i=%n j=%n bits=%x5 bitsq=%x5 bitsj=%x5*n",",
// q, i, j, bits, bitsq, bitsj)
//abort(2001)
IF (bits & bitsj) = 0 DO
{  // Found a successor
    LET S1, V4, H1, U4 = s_S1!p, s_V4!p, bitsj, s_U4!p
    LET succ = find(S1,V4,H1,U4)
    s_succs!p := mk2(s_succs!p, succ)
    edgecount := edgecount+1
    //writef("H successor ")
    //prboard(S1,V4,H1,U4)
    //newline()
    //abort(1000)
}
}

AND mksuccsU(p, all, q) BE
{  // all is a bit pattern giving all occupied squares
    // q is the current placement number of a 1x1 U piece
    LET succsv = succsU!q  // Vector of successors of placement q
    LET bitsq = bitsU!q  // The bit pattern for placement q
    LET bits = all - bitsq  // all with placement q removed
    FOR i = 1 TO succsv!0 DO
    {  // An adjacent placement of a 1x1 U piece
        LET bitsj = bitsU!j  // The bit pattern for placement j
        //writef("mksuccsU: q=%n i=%n j=%n bits=%x5 bitsq=%x5 bitsj=%x5*n",",
        // q, i, j, bits, bitsq, bitsj)
        //abort(2001)
        IF (bits & bitsj) = 0 DO
        {  // Found a successor
            LET S1, V4, H1, U4 = s_S1!p, s_V4!p, s_H1!p, s_U4!p-bitsq+bitsj
            LET succ = find(S1,V4,H1,U4)
            s_succs!p := mk2(s_succs!p, succ)
            edgecount := edgecount+1
            //writef("U successor ")
            //prboard(S1,V4,H1,U4)
            //newline()
            //abort(1000)
        }
    }
}

The program continues as follows.
AND explore() BE
{ componentp := 1
  componentcount := 0
  componentsizemax := 0
  componentsizemin := maxint

  // Find the starting position
  root := find(#x66000, #x09999, #x00006, #x00660)

  WHILE root DO
    { dist := ?

      // Insert the root of the next simply connected component
      s_link!root, s_dist!root := 0, 0
      listv!0 := root
      dist := 0
      componentcount := componentcount + 1
      componentsize := 1

      WHILE listv!dist DO
        { dist := dist+1
          createlist(dist)
        }

      // The component is now complete
      IF componentsize > componentsizemax DO componentsizemax := componentsize
      IF componentsize < componentsizemin DO componentsizemin := componentsize

      IF tracing DO
        { writef("Component %i3 size %i5 root ", componentcount, componentsize)
          prboard(s_S1!root, s_V4!root, s_H1!root, s_U4!root)
          newline()
          //abort(1007)
        }

      // Find the root of the next component
      root := 0
      WHILE componentp <= nodevupb DO
        { LET node = nodev!componentp
          //writef("componentp = %i5*n", componentp)
          IF s_dist!node < 0 DO
            { root := node
              //writef("new component root = %i5*n", root)
            }
          //abort(1008)
          BREAK
        }
    }
  // Find the root of the next simply connected component
  root := 0
  WHILE componentp <= nodevupb DO
    { LET node = nodev!componentp
      //writef("componentp = %i5*n", componentp)
      IF s_dist!node < 0 DO
        { root := node
          //writef("new component root = %i5*n", root)
        }
      //abort(1008)
      BREAK
    }
The program continues as follows.

\[
\text{componentp := componentp + 1}
\]

\[
\]

AND createlist(dist) BE

{ LET prevnode = listv!(dist-1) // List of nodes at distance dist
  //writef("Making list of nodes at distance \text{n*\text{n}n", dist)
  //writef("prevnode=%n*\text{n}n", prevnode)
  //abort(1006)

  // Create list of nodes at the new distance.
  // The list is initially empty.
  listv!dist := 0

  // Inspect every node at distance dist-1
  WHILE prevnode DO
    { // prevnode is a node at the previous distance.
      // Any successors of prevnode that have not yet been
      // visited are to be inserted into listv!dist.
      LET succs = s_succs!prevnode // List of nodes adjacent to prevnode

      //writef("exploring successors of \text{ }")
      //prboard(s_S1!prevnode, s_V4!prevnode, s_H!prevnode, s_U4!prevnode)
      //newline()

      WHILE succs DO
        { LET succ = succs!1 // succ is a successor to prevnode
          IF s_dist!succ < 0 DO
            { // succ has not yet been visited
              s_dist!succ := dist
              s_prev!succ := prevnode
              s_link!succ := listv!dist
              listv!dist := succ
              componentsize := componentsize + 1
              //writef("dist=%i4 \text{, }", dist)
              //prboard(s_S1!succ, s_V4!succ, s_H1!succ, s_U4!succ)
              //newline()
            UNLESS solution IF s_S1!succ=#x00066 DO
              { solution := succ
                //writef("Solution=n")
                //abort(1111)
4.26. THE SLIDING BLOCKS PUZZLE

The program continues as follows.

\[
\begin{align*}
&\text{AND prboard(S1, V4, H1, U4) BE} \\
&\{ \ \text{LET bit} = \#x80000 \\
&\text{WHILE bit DO} \\
&\{ \ \text{LET ch} = '**', \\
&\text{UNLESS (S1 & bit) = 0 DO ch := 'S'} \\
&\text{UNLESS (H1 & bit) = 0 DO ch := 'H'} \\
&\text{UNLESS (V4 & bit) = 0 DO ch := 'V'} \\
&\text{UNLESS (U4 & bit) = 0 DO ch := 'U'} \\
&\text{writef(" %c", ch)} \\
&\text{IF (bit & \#x11110) > 0 DO writef(" ")} \\
&\text{bit := bit>>1} \\
&\} \\
&\} \\
&\text{AND prsol(node) BE} \\
&\{ \ \text{LET S1} = s_{S1}!node \\
&\text{LET V4} = s_{V4}!node \\
&\text{LET H1} = s_{H1}!node \\
&\text{LET U4} = s_{U4}!node \\
&\text{IF s_prev!node DO prsol(s_prev!node)} \\
&\text{writef("%3i: ", s_dist!node)} \\
&\text{prboard(S1, V4, H1, U4)} \\
&\text{IF S1=#x00066 DO writes(" solution")} \\
&\text{newline()} \\
&\} \\
\end{align*}
\]

When this program runs it outputs the following.

0:  * S S *  V S S V  V U U V  V U U V  V H H V
1:  V S S *  V S S V  * U U V  V U U V  V H H V
4.26. THE SLIDING BLOCKS PUZZLE

47: V U V V V U V * U S S * V S S V U H H
48: V U V V V U V V V V U U S S V * U H H
49: V U V V V V U V V V S S V S U S H H
50: V U V V V U V V V V U U S S S U H H
51: V U V V V U V V * V S S V S S S S S H H
52: V U V V V U V V * V U V V V * S S V U H H
53: V U V V V U V V V S S S V * S S * U H H
54: V U V V V U V V V * S S V U S S * U H H
55: V U V V V U V * V V S S V * S S V U H H
56: V * U V V U * V V V S S V U S S V H H
57: V * U V V * U V V V S S V S S V U H H
58: * V U V V * U V V V S S V S S V U H H
59: * V U V V V U V V V V S S V * S S V U H H
60: V V V V V U V V * S S V * S S V U H H
61: V V V V V U V V S S S * V U H H
62: V V V V V V * S S V S S V U H H
63: V V V * V V U * V V V S S V S S * V U H H
64: V V V * V V V * V S S V S S V U H H
65: V V V * V V V V S S V S S V U H H
66: V V V * V V V * V S S V S S V U H H
67: V V V * V V V * V S S V S S V U H H
68: V V V V V V V V S S U * S S U * U H H
69: V V V V V V V V S S U S S U U U H H
70: V V V V V V V V S S U S S * * S S H H H U U H H
71: V V V V V V V V S S U S S U S S H H H U U * * * U
72: V V V V V V V V S S U S S U S S H H H U U * * U
73: V V V V V V V V S S U S S H H H U U * U * U
74: V V V V V V V V S S U S S U S S H H H U U * U * U
75: V V V V V V V V S S U S S H H H * * U U
76: V V V V V V V V S S U S S H H H * * U U
77: V V V V V V V V S S U S S H H H * * U U
78: V V V V V V V V S S U S S H H H * * U U
79: V V V V V V V V U U * * U S S H H S S U U
80: V V V V V V V V U U * * U S S H H S S U U
81: V V V V V V V V U U H H S S * * S S U U
82: V V V V V V V V U U H H S S U S S * U
83: V V V V V V V V U U H H S S U S S * U
84: V V V V V V V V U U H H S S U S S * U

nodecount= 65880
edgecount= 206780
componentcount= 898
componentsizemax= 25955
componentsizemin= 2
space used = 1736680 words
4.27 The Rubik Cube

The popular Rubik Cube puzzle, pictured below, has much in common with the sliding blocks puzzle described above. From any position, you can make a small number of moves to reach adjacent positions. Unfortunately there are $43,252,033,639,748,818,256,000$ possible positions (see rubik cube on the web) making it impossible to represent the entire graph in memory.

Much more to follow

My aim was to construct a program to solve the rubik cube starting at any random position without resorting to one of the recipies available on the web. I have so far failed and am unlikely to attempt to improve the program, so here is the current draft (called rubik.b). Even if you choose not to study this program in detail, you might like to look at the function findnode since it shows how hash tables can be implemented. Ffloyd’s algorithm might also be of interest (see the function ffloyd). Information about this algorithm is easily available on the web. Do a web search on ffloyds algorithm.

/*

############## UNDER DEVELOPMENT ###################################

This program is unlikely to ever be finished, but may be of interest all the same.

This is a second attempt to write a program to solve the rubik cube. The first attempt (in rubik1.b) used a strategy that was too
slow to be useful unless the solution has a rather small number of moves.

This program attempts to solve Rubik Cube problems, given a textual specification of an initial position, it will hopefully output a sequence of rotations to solve the cube.

Implemented by Martin Richards (c) January 2015

This program uses a lot of work space so it is a good idea to run cintsys with a large memory size. You can, for instance, run the system with 100 million words of Cintcode memory by executing the following shell command.

cintsys -m 100000000

This program is still too slow to find solutions in general, but seems to get quite close. For instance, output generated by the command

rubik -s 4

ends as follows:

new bestscore=434 nodecount=4491837
        W W W
        W W W
        W W W
        G G G R R R B B B O O O
        G G G R R R B B O B O O
        G Y G R O R B B B O G O
        Y Y Y
        R Y Y
        Y Y Y
Insufficient space

nodecount = 8259446
space used: 75000002 out of 75000000
360.630>

So it found that partial solution after visiting fewer than 5 million nodes. Note that only a few pieces are not in their correct positions.

*/

GET "libhdr"
MANIFEST {
    // This program assumes the cube is always in the same orientation
    // with upper face being white and the front face red.
    // The other faces are
    // right blue
    // back orange
    // left green
    // down yellow

    // Corner piece definitions
    // orientation 0 means W/Y piece face is parallel to up face
    // 1 means the piece was rotated anticlockwise once
    // when looking towards its corner.
    // 2 means the piece was rotated anticlockwise twice
    WRB0=0*3+0; WRB1=0*3+1; WRB2=0*3+2 // Corner 0
    WBO0=1*3+0; WBO1=1*3+1; WBO2=1*3+2 // Corner 1
    WGO0=2*3+0; WGO1=2*3+1; WGO2=2*3+2 // Corner 2
    WGR0=3*3+0; WGR1=3*3+1; WGR2=3*3+2 // Corner 3
    YBR0=4*3+0; YBR1=4*3+1; YBR2=4*3+2 // Corner 4
    YOB0=5*3+0; YOB1=5*3+1; YOB2=5*3+2 // Corner 5
    YGO0=6*3+0; YGO1=6*3+1; YGO2=6*3+2 // Corner 6
    YRG0=7*3+0; YRG1=7*3+1; YRG2=7*3+2 // Corner 7

corncostvupb = YRG2
    corncostvsize = corncostvupb+1 // Number of elements in a row or column
    corncostmupb = corncostvsize*corncostvsize-1 // Upb of the matrix

    // There are 12 Edge pieces
    // The edge directions are
    // 0->1 1->2 2->3 3->0
    // 0->4 1->5 2->6 3->7
    // 4->7 5->4 6->5 7->6
    // orientation 0 means the first colour is on the left when
    // looking forward along the edge
    // orientation 1 means the first colour is on the right when
    // looking forward along the edge

    // Upper level edges
    WR0= 0*2+0; WR1= 0*2+1 // in edge 0->1
    WBO= 1*2+0; WBO1= 1*2+1 // in edge 1->2
    WGO= 2*2+0; WGO1= 2*2+1 // in edge 2->3
    WGR= 3*2+0; WGR1= 3*2+1 // in edge 3->0

    // Middle layer edges
4.27. THE RUBIK CUBE

BR0= 4*2+0; BR1= 4*2+1 // in edge 0->4
OB0= 5*2+0; OB1= 5*2+1 // in edge 1->5
GO0= 6*2+0; GO1= 6*2+1 // in edge 2->6
RG0= 7*2+0; RG1= 7*2+1 // in edge 3->7

// Down layer edges
YR0= 8*2+0; YR1= 8*2+1 // in edge 4->7
YB0= 9*2+0; YB1= 9*2+1 // in edge 5->4
YO0=10*2+0; YO1=10*2+1 // in edge 6->5
YG0=11*2+0; YG1=11*2+1 // in edge 7->6

egecostvupb = YG1
dedgecostvsize = edgecostvupb+1 // Number of elements in a row or column
dedgecostmupb = edgecostvsize*edgecostvsize-1 // Upb of the matrix

// 8 Corner positions used in the cost function
cWBR=0; cWBO; cWGO; cWGR // White corners
cYBR; cYOB; cYGO; cYRG // Yellow corners

// 12 Edge positions used in the cost function
eWR=0; eWB; eWD; eWG
eBR; eOB; eGO; eRG
eYR; eYB; eYO; eYG

// 8 Corner byte position indexes on the cube
iWBR=0; iWBO; iWGO; iWGR // White corners
iYBR; iYOB; iYGO; iYRG // Yellow corners

// 12 Edge byte position indexes on the cube
iWR; iWB; iWO; iWG
iBR; iOB; iGO; iRG
iYR; iYB; iYO; iYG

s_chain= iYG / bytesperword + 1 // Hash chain field
s_prev // Immediate predecessor
s_move // The move from predecessor to this node
s_maxdepth // This node has been or is being searched
          // with this setting of maxdepth
nodeupb = s_maxdepth

// Moves for Upper, Front, Right, Back, Left and Down
//   c = clockwise
//   a = anti clockwise
// These are used to record the sequence of moves
mUc='U'; mUa='u'
mFc='F'; mFa='f'
mRc='R'; mRa='r'
mBc='B'; mBa='b'
mlc='L'; mLa='l'
mDc='D'; mDa='d'
}

GLOBAL {
// 8 Corner positions on the p cube as global variables
pWRB;ug; pWBO; pWOG; pWGR // White corners
pYBR; pYOB; pYGO; pYRG // Yellow corners
pWR; pWB; pWO; pWG // 12 Edge positions on the p cube
pBR; pOB; pGO; pRG
pYR; pYB; pYO; pYG

// 8 Corner positions on the q cube as global variables
qWRB; qWBO; qWOG; qWGR // White corners
qYBR; qYOB; qYGO; qYRG // Yellow corners
qWR; qWB; qWO; qWG // 12 Edge positions on the q cube
qBR; qOB; qGO; qRG
qYR; qYB; qYO; qYG

corncostm
corncostv
// corncostm is a 24x24 matrix giving the cost of moving a
// piece from one corner of the cube to another changing its
// orientation at the same time. If i and j are row and
// column subscripts of corncostm then they have the form
// corner*3+oritation where corner is the corner number
// in the range 0 to 7 and oritation is the orientation
// number in the range 0 to 2.
// corncostv!i is a vector corresponding to the ith row
// of matrix corncostm. So the (i,j)th element of the matrix
// can be accessed by corncostv!i!j. To see how it is used
// see the function corncost.

dedgecostm
dedgecostv
// edgecostm is a 24x24 matrix giving the cost of moving a
// piece from one edge position to another possibly flipping
// its orientation. Its structure is similar to corncostm.
// The ((i,j)th element of edgecostm can be accessed by
// edgecost!i!j. See the function edgecost.

fin_p; fin_l
spacev; spacep; spacet
spacevupb
hashtabsize
hashtabupb
mkvec
nodecount
hashtab
hashfn
findnode // Find a node in the hash table, creating one
    // if necessary.
cube // A packed cube -- 20 bytes = 5 words
colour // colour!0 .. colour!53
ersors // =TRUE if an error has occurred
moves // Initialising moves supplied by -m argument
bestnode
bestscore
initcostfn
costfn
score // (node) returns the node's score
scorenode
exploreroot
exploretree
try
prnode
tracing
compact // =TRUE for compact configuration output
randomise // Set by the -r or -s options
pieces2cube
cube2pieces
rotc
rota
flip
rotateUc; rotateUa
rotateDc; rotateDa
rotateFc; rotateFa
rotateBc; rotateBa
rotateRc; rotateRa
rotateLc; rotateLa
movecubep2q; movecubeq2p
cornrotate; edgerotate
ffloyd
prcornmat; predgemat
prmoves
corncost; edgecost
prcosts
prcorncost; predgecost
prsolution
wrcornerpiece; wredgepiece
prpieces
prnode; prnode
setface
corner; edge
cols2cube; cube2cols
setcornercols; setedgecols
}

LET hashfn(node) = VALOF
{ // Return a hash value in range 0 to hashtabupb
LET w = node!0 XOR node!1 XOR node!2 XOR node!3 XOR node!4
LET h = w MOD hashtabsize
UNLESS 0 <= h <= hashtabupb DO
{ prnode(node)
  writef("%x8 %x8 %x8 %x8 %x8*n",
         node!0, node!1, node!2, node!3, node!4)
  writef("w = %x8 => hashval = %n*n", w, h)
  abort(999)
}
RESULTIS h
}

AND findnode(cube, prev, move) = VALOF
{ // Find the node that matches the configuration in cube
  // prev=0 or is the immediate predecessor
  // move=0 or is the move to reach this node
  // These values are only used if the node has not been seen before.
  // It creates a new node if necessary.
  LET hashval = hashfn(cube)
  LET node = hashtab!hashval
  //writef("hashval=%n node=%n*n", hashval, node)
  WHILE node DO
  { IF cube!0=node!0 &
    cube!1=node!1 &
    cube!2=node!2 &
    cube!3=node!3 &
    cube!4=node!4 DO
    { //writef("node %n has been seen before*n", node)
      RESULTIS node // The node already exists
    }
  }
  IF hashval!=node DO
    node = hashval
  RESULTIS node
4.27. THE RUBIK CUBE

node := s_chain!node

//writef("Matching node not found so create one\n")

// The matching node has not been found so create one.

node := mkvec(nodeupb)
UNLESS node DO
  { writef("Mode space needed\n")
    stop(0, 0) //abort(999)
    RESULTIS 0
  }
// Fill in all its fields
node!0 := cube!0 // The corners
node!1 := cube!1
node!2 := cube!2 // The edges
node!3 := cube!3
node!4 := cube!4

// Fill in its remaining fields
s_prev!node := prev
s_move!node := move
s_maxdepth!node := 0

// Insert it into its hash chain
s_chain!node := hashtab!hashval
hashtab!hashval := node

nodecount := nodecount+1

IF tracing DO
  { writef("New node \n, nodecount=\n\n", node, nodecount)
    prnode(node)
  }

RESULTIS node

AND mkvec(upb) = VALOF
{ LET p = spacep
  spacep := spacep+upb+1
  IF spacep>spacet DO
    { writef("Insufficient space\n")
  }
}
longjump(fin_p, fin_l) //abort(999)
RESULTIS 0
}
RESULTIS p
}

LET start() = VALOF
{ LET argv = VEC 50
  LET root = 0

  fin_p := level()
  fin_l := fin

  // Allocate 75% of current Cintcode memory as work space.
  // All other space used by this program is taken out of
  // this allocation.
  spacevupb := rootnode!rtn_memsize*3/4
  hashtabsize := spacevupb/113
  hashtabupb := hashtabsize-1
  writef("Allocating %n words of work space, hashtabupb=\%n\n", spacevupb, hashtabupb)

  spacev := getvec(spacevupb)

  spacep, spacet := spacev, spacev+spacevupb

  UNLESS spacev DO
    { writef("Insufficient space available, cannot allocate spacev\n")
      GOTO fin
    }
  
  cube := mkvec(nodeupb) // Structure representing the current state of the cube
  colour := mkvec(6*9-1)
  corncostm := mkvec(corncostmupb)
  corncostv := mkvec(corncostvupb)
  edgecostm := mkvec(edgecostmupb)
  edgecostv := mkvec(edgecostvupb)

  UNLESS cube & colour &
    corncostm & edgecostm &
    corncostv & edgecostv DO
    { writef("Insufficient space available\n")
      GOTO fin
    }
4.27. THE RUBIK CUBE

```plaintext
errors := FALSE

UNLESS rdargs("W,R,B,O,G,Y,-m/K,-s/K/N,-r/S,-t/S,-c/S", argv, 50) DO
  { writef("Bad arguments for Rubik*n")
    GOTO fin
  }

  // Set default colours of the solved cube
  FOR i = 0 TO 8 DO colour!i := 'W'
  FOR i = 9 TO 17 DO colour!i := 'R'
  FOR i = 18 TO 26 DO colour!i := 'B'
  FOR i = 27 TO 35 DO colour!i := 'O'
  FOR i = 36 TO 44 DO colour!i := 'G'
  FOR i = 45 TO 53 DO colour!i := 'Y'

  // Set user specified colours
  IF argv!0 DO setface(0, 'W', argv!0) // W
  IF argv!1 DO setface(1, 'R', argv!1) // R
  IF argv!2 DO setface(2, 'B', argv!2) // B
  IF argv!3 DO setface(3, 'O', argv!3) // O
  IF argv!4 DO setface(4, 'G', argv!4) // G
  IF argv!5 DO setface(5, 'Y', argv!5) // Y

  moves := argv!6 // -m/K

  randomise := FALSE

  IF argv!7 DO // -s/K/N
    { //writef("calling setseed(%n)*n", !(argv!7))
      setseed(!(argv!7))
      randomise := TRUE
    }
  IF argv!8 DO // -r/S
    { LET day, msecs, filler = 0, 0, 0
      datstamp(@day)
      randomise := TRUE
      setseed(msecs) // Set seed based on time of day
    }
  tracing := argv!9 // -t/S
  compact := argv!10 // -c/S

  cols2cube(colour, cube)
  cube2pieces(cube, @pWRB)

  // Make initial moves, if any
```

---

211
IF moves FOR i = 1 TO moves%0 DO
{ SWITCHON moves%i INTO
  { DEFAULT: writef("Bad initial moves %s*n", moves)
      errors := TRUE
      BREAK
  CASE 'U': rotateUc(); ENDCASE
  CASE 'u': rotateUa(); ENDCASE
  CASE 'F': rotateFc(); ENDCASE
  CASE 'f': rotateFa(); ENDCASE
  CASE 'R': rotateRc(); ENDCASE
  CASE 'r': rotateRa(); ENDCASE
  CASE 'B': rotateBc(); ENDCASE
  CASE 'b': rotateBa(); ENDCASE
  CASE 'L': rotateLc(); ENDCASE
  CASE 'l': rotateLa(); ENDCASE
  CASE 'D': rotateDc(); ENDCASE
  CASE 'd': rotateDa(); ENDCASE
}
movecubeq2p()
}

// Possibly randomise the cube
IF randomise FOR i = 1 TO 200 DO
{ SWITCHON randno(15) INTO
  { DEFAULT: LOOP
    CASE 1: rotateUc(); ENDCASE
    CASE 2: rotateUa(); ENDCASE
    CASE 3: rotateFc(); ENDCASE
    CASE 4: rotateFa(); ENDCASE
    CASE 5: rotateRc(); ENDCASE
    CASE 6: rotateRa(); ENDCASE
    CASE 7: rotateBc(); ENDCASE
    CASE 8: rotateBa(); ENDCASE
    CASE 9: rotateLc(); ENDCASE
    CASE 10: rotateLa(); ENDCASE
    CASE 11: rotateDc(); ENDCASE
    CASE 12: rotateDa(); ENDCASE
  }
  movecubeq2p()
}

IF errors RESULTIS 0
4.27. THE RUBIK CUBE

// Pack the starting position in cube
pieces2cube(pWRB, cube)

newline()
newline()
initcostfn()
//prcosts()
//writef("*nThe starting position is:*n*n")
//prpieces(pWRB); newline()
//movecubep2q()
//writef("*n*score = %n*n", score()+goalscore(cube))
//prnode(cube)
//newline()
//abort(1000)

hashtab := mkvec(hashtabupb)
FOR i = 0 TO hashtabupb DO hashtab[i] := 0

nodecount := 0

// The starting node configuration is now in cube

//writef("*n*Creating the starting position*n")

// Create a new node with prev=0 and no move
root := findnode(cube, 0, 0, 0)

{ LET bestsc = bestscore
  root := exploreroot(root, 1)
  IF bestscore=0 | bestsc=bestscore BREAK
} REPEAT

writef("*n*Solution*n*n")
prsolution(root)

fin:
  writef("*n*nodecount = %n*n", nodecount)
writef("*n*space used: %n out of %n*n", spacep-spacev, spacet-spacev)

IF spacev DO freevec(spacev)
RESULTIS 0
}
AND exploreroot(root, maxdepth) = VALOF
{ // root is a new root node from which to start the search
  // to find a nearest node with minimum score no more than
  // maxdepth away. During the search nodes are put into the hash
  // table so that we can easily test whether a node has already
  // been visited.
  // The function returns a node with minimum score.
  // If the best node has the same score as root, exploreroot will
  // have to be called again with a larger maxdepth.

  LET rootscore = scorenode(root)

  // Initialise bestscore and bestnode
  bestscore, bestnode := rootscore, root

  //writef("exploreroot: score=%n space used = %n*n", rootscore, spacep-spacev)
  //prnode(root)
  IF bestscore=0 RESULTIS root
  //abort(5000)

  explretree(root, maxdepth)

  IF bestscore < rootscore RESULTIS bestnode

  maxdepth := maxdepth + 1
  //writef("bestscore = %n, trying exploreroot with new maxdepth = %n*n",
  //      bestscore, maxdepth)
  //abort(6000)
} REPEAT

AND explretree(node, maxdepth) BE
{ LET sc = score()+goalscore(node)

  IF sc < bestscore DO
    { bestscore, bestnode := sc, node
      writef("new bestscore=%n nodecount=%n*n", bestscore, nodecount)
      prnode(node)
      //abort(7000)
    }
  //writef("explretree: maxdepth=%n score=%n bestscore=%n nodecount=%n*n",
  //    maxdepth, sc, bestscore, nodecount)
  //prnode(node)
  //IF sc=0 DO abort(1000)

  IF maxdepth=0 RETURN // We have reached the depth limit
4.27. THE RUBIK CUBE

// Return is this node has already be processed at this maxdepth.
IF s_maxdepth!node >= maxdepth RETURN

// Try the 12 possible successors of this node
// in the list.
try(rotateUc, node, mUc, maxdepth)
try(rotateUa, node, mUa, maxdepth)
try(rotateFc, node, mFc, maxdepth)
try(rotateFa, node, mFa, maxdepth)
try(rotateRc, node, mRc, maxdepth)
try(rotateRa, node, mRa, maxdepth)
try(rotateBc, node, mBc, maxdepth)
try(rotateBa, node, mBa, maxdepth)
try(rotateLc, node, mLc, maxdepth)
try(rotateLa, node, mLa, maxdepth)
try(rotateDc, node, mDc, maxdepth)
try(rotateDa, node, mDa, maxdepth)
}

AND try(rotfn, prev, move, maxdepth) BE IF bestscore DO
{
  // Explore an immediate successor of node prev
  LET node = ?
  // First unpack prev in pWRB, etc
  cube2pieces(prev, @pWRB)

  // ppieces(@pWRB)
  rotfn() // q cube := p cube with one face rotated
  // newline()
  // ppieces(@qWRB)
  // abort(1000)
  pieces2cube(@qWRB, cube)
  node := findnode(cube, prev, move)

  exploretree(node, maxdepth-1) // Explore the successor nodes
}

AND pieces2cube(pieces, cube) BE
{
  cube%iWRB := pieces!iWRB
  cube%iWBO := pieces!iWBO
  cube%iWOG := pieces!iWOG
  cube%iWGR := pieces!iWGR
  cube%iYBR := pieces!iYBR
  cube%iYOB := pieces!iYOB
  cube%iYGO := pieces!iYGO
}
cube%iYRG := pieces!iYRG

cube%iWR := pieces!iWR
cube%iWB := pieces!iWB
cube%iWO := pieces!iWO
cube%iWG := pieces!iWG

cube%iBR := pieces!iBR
cube%iOB := pieces!iOB
cube%iGO := pieces!iGO
cube%iRG := pieces!iRG

cube%iYR := pieces!iYR
cube%iYB := pieces!iYB
cube%iYO := pieces!iYO
cube%iYG := pieces!iYG
}

AND cube2pieces(cube, pieces) BE
{  pieces!iWRB := cube%iWRB
  pieces!iWBO := cube%iWBO
  pieces!iWOG := cube%iWOG
  pieces!iWGR := cube%iWGR
  pieces!iYBR := cube%iYBR
  pieces!iYOB := cube%iYOB
  pieces!iYGO := cube%iYGO
  pieces!iYRG := cube%iYRG

  pieces!iWR := cube%iWR
  pieces!iWB := cube%iWB
  pieces!iWO := cube%iWO
  pieces!iWG := cube%iWG

  pieces!iBR := cube%iBR
  pieces!iOB := cube%iOB
  pieces!iGO := cube%iGO
  pieces!iRG := cube%iRG

  pieces!iYR := cube%iYR
  pieces!iYB := cube%iYB
  pieces!iYO := cube%iYO
  pieces!iYG := cube%iYG
  }

AND rotc(piece) = VALOF SWITCHON piece INTO
4.27. THE RUBIK CUBE

{ // Rotate a corner piece one position clockwise
    DEFAULT: writef("rotc: System error, piece=%n*n", piece)
        abort(999)
    RESULTIS piece

        CASE WRB1: CASE WRB2: CASE WBO1: CASE WBO2:
            CASE WOG1: CASE WOG2: CASE WGR1: CASE WGR2:
                CASE YBR1: CASE YBR2: CASE YOB1: CASE YOB2:
                    CASE YGO1: CASE YGO2: CASE YRG1: CASE YRG2:
                        RESULTIS piece-1

        CASE WRBO: CASE WBO0: CASE WOG0: CASE WGR0:
            CASE YBO0: CASE YBR0: CASE YGO0: CASE YRG0:
                RESULTIS piece+2
}

AND rota(piece) = VALOF SWITCHON piece INTO
{ // Rotate a corner piece one position anti-clockwise
    DEFAULT: writef("rot1: System error, piece=%n*n", piece)
        abort(999)
    RESULTIS piece

        CASE WRB0: CASE WRB1: CASE WBO0: CASE WBO1:
            CASE WOG0: CASE WOG1: CASE WGR0: CASE WGR1:
                CASE YBR0: CASE YBR1: CASE YOB0: CASE YOB1:
                    CASE YGO0: CASE YGO1: CASE YRG0: CASE YRG1:
                        RESULTIS piece+1

        CASE WRB2: CASE WBO2: CASE WGO2: CASE WGR2:
            CASE YOB2: CASE YBR2: CASE YGO2: CASE YRG2:
                RESULTIS piece-2
}

AND flip(piece) = piece XOR 1 // Flip an edge piece

AND rotateUc() BE
{ // Rotate the upper face clockwise by a quarter turn
    qWRB, qWBO, qWOG, qWGR := pWBO, pWOG, pWGR, pWRB // Rotated
    qYBR, qYOB, qYGO, qYRG := pYBR, pYOB, pYGO, pYRG // Not rotated
    qWR, qWB, qWO, qWG := pWB, pWO, pWG, pWR // Rotated
    qBR, qOB, qGO, qRG := pBR, pOB, pGO, pRG // Not rotated
    qYR, qYB, qYO, qYG := pYR, pYB, pYO, pYG // Not rotated
}
AND rotateUa() BE
{ // Rotate the upper face anti-clockwise by a quarter turn
  qWRB, qWBO, qWOG, qWGR := pWRB, pWBO, pWOG // Rotated
  qYBR, qYOB, qYGO, qYRG := pYBR, pYOB, pYGO, pYRG // Not rotated
  qWR, qWB, qWO, qWG := pWR, pWB, pWO // Rotated
  qBR, qBO, qGO, qRG := pBR, pBO, pGO, pRG // Not rotated
  qYR, qYB, qYO, qYG := pYR, pYB, pYO, pYG // Not rotated
}

AND rotateDc() BE
{ // Rotate the down face clockwise by a quarter turn
  qWRB, qWBO, qWOG, qWGR := pWRB, pWBO, pWOG, pWGR // Not rotated
  qYBR, qYOB, qYGO, qYRG := pYRG, pYBR, pYOB, pYGO //Rotated
  qWR, qWB, qWO, qWG := pWR, pWB, pWO, pWG // Not rotated
  qBR, qOB, qGO, qRG := pBR, pOB, pGO, pRG // Not rotated
  qYR, qYB, qYO, qYG := pYG, pYR, pYB, pYO // Rotated
}

AND rotateDa() BE
{ // Rotate the down face anti-clockwise by a quarter turn
  qWRB, qWBO, qWOG, qWGR := pWRB, pWBO, pWOG, pWGR // Not rotated
  qYBR, qYOB, qYGO, qYRG := pYBR, pYOB, pYGO, pYRG // Rotated
  qWR, qWB, qWO, qWG := pWR, pWB, pWO, pWG // Not rotated
  qBR, qOB, qGO, qRG := pBR, pOB, pGO, pRG // Not rotated
  qYR, qYB, qYO, qYG := pYG, pYR, pYB, pYO // Rotated
}

AND rotateFc() BE
{ // Rotate the front face clockwise by a quarter turn
  qWRB, qYBR, qYRG, qWGR := rotc(pWGR), rota(pWRB), rotc(pYBR), rota(pYRG) // Rotated
  qWBO, qYOB, qYGO, qWOG := pWBO, pYOB, pYGO, pWOG // Not rotated
  qWR, qBR, qYR, qRG := flip(pRG), pWR, pBR, flip(pYR) // Rotated
  qWB, qYB, qYG, qWG := pWB, pYB, pYG, pWG // Not rotated
  qWO, qOB, qGO, qGO := pWO, pOB, pGO, pGO // Not rotated
}

AND rotateFa() BE
{ // Rotate the front face anti-clockwise by a quarter turn
  qWRB, qYBR, qYRG, qWGR := rotc(pYBR), rota(pYRG), rotc(pWGR), rota(pWRB) // Rotated
  qWBO, qYOB, qYGO, qWOG := pWBO, pYOB, pYGO, pWOG // Not rotated
  qWR, qBR, qYR, qRG := pBR, pYR, flip(pRG), flip(pWR) // Rotated
  qWB, qYB, qYG, qWG := pWB, pYB, pYG, pWG // Not rotated
  qWO, qOB, qGO, qGO := pWO, pOB, pYO, pGO // Not rotated
}
AND rotateBc() BE
{
    // Rotate the back face clockwise by a quarter turn
    qWBO, qWOG, qYGO, qYOB := rota(pYOB), rotc(pWBO), rota(pWOG), rotc(pYGO) // Rotated
    qWBB, qWGR, qYRG, qYBR := pWBB, pWGR, pYRG, pYBR // Not rotated
    qWO, qGO, qYO, qOB := flip(pOB), pWO, pGO, flip(pYO) // Rotated
    qWB, qWG, qYG, qYB := pWB, pWG, pYG, pYB // Not rotated

    qWR, qRG, qYR, qBR := pWR, pRG, pYR, pBR // Not rotated
}

AND rotateBa() BE
{
    // Rotate the back face anti-clockwise by a quarter turn
    qWBO, qWOG, qYGO, qYOB := rota(pWOG), rotc(pYGO), rota(pYOB), rotc(pWBO) // Rotated
    qWBB, qWGR, qYRG, qYBR := pWBB, pWGR, pYRG, pYBR // Not rotated
    qWO, qGO, qYO, qOB := pGO, pYO, flip(pOB), flip(pWO) // Rotated
    qWB, qWG, qYG, qYB := pWB, pWG, pYG, pYB // Not rotated

    qWR, qRG, qYR, qBR := pWR, pRG, pYR, pBR // Not rotated
}

AND rotateRc() BE
{
    // Rotate the right face clockwise by a quarter turn
    qWRB, qWBO, qYOB, qYBR := rota(pYBR), rotc(pWBO), rota(pWOG), rotc(pYOG) // Rotated
    qWGR, qYRG, qYGO, qWOG := pWGR, pYRG, pYGO, pWOG // Not rotated
    qWB, qOB, qYB, qBR := flip(pBR), pWB, pOB, flip(pYB) // Rotated
    qWR, qWO, qYO, qYR := pWR, pWO, pYO, pYR // Not rotated
    qWG, qRG, qYG, qGO := pWG, pRG, pYG, pGO // Not rotated
}

AND rotateRa() BE
{
    // Rotate the right face anti-clockwise by a quarter turn
    qWRB, qWBO, qYOB, qYBR := rota(pWBO), rotc(pYOB), rota(pYBR), rotc(pWBR) // Rotated
    qWGR, qYRG, qYGO, qWOG := pWGR, pYRG, pYGO, pWOG // Not rotated
    qWB, qOB, qYB, qBR := pOB, pYB, flip(pBR), flip(pWB) // Rotated
    qWR, qWO, qYO, qYR := pWR, pWO, pYO, pYR // Not rotated
    qWG, qRG, qYG, qGO := pWG, pRG, pYG, pGO // Not rotated
}

AND rotateLc() BE
{
    // Rotate the left face clockwise by a quarter turn
    qWGR, qYRG, qYGO, qWOG := rotc(pWOG), rota(pWGR), rotc(pYRG), rota(pYGO) // Rotated
    qWBO, qYOB, qYBR, qWRB := pWBO, pYOB, pYBR, pWRB // Not rotated
    qWG, qRG, qYG, qGO := flip(pGO), pWG, pRG, flip(pYG) // Rotated
    qWR, qYR, qYO, qWO := pWR, pYR, pYO, pWO // Not rotated
    qWB, qOB, qYB, qBR := pWB, pOB, pYB, pBR // Not rotated
}
AND rotateLa() BE
  { // Rotate the left face anti-clockwise by a quarter turn
    qWGR, qYRG, qYG0, qWOG := rotc(pYRG), rota(pYG0), rotc(pWOG), rota(pWGR) // Rotated
    qWBO, qYOB, qYBR, qWBR := pWBO, pYOB, pYBR, pWBR // Not rotated
    qWB, qOB, qYB, qBR := pWB, pOB, pYB, pBR // Not rotated
  }

AND movecubep2q() BE
  { qWRB, qWBO, qWOG, qWGR := pWRB, pWBO, pWOG, pWGR
    qYBR, qYOB, qYGO, qYRG := pYBR, pYOB, pYGO, pYRG
    qWR, qWB, qWO, qWG := pWR, pWB, pWO, pWG
    qBR, qOB, qGO, qRG := pBR, pOB, pGO, pRG
    qYR, qYB, qYO, qYG := pYR, pYB, pYO, pYG
  }

AND movecubeq2p() BE
  { pWRB, pWBO, pWOG, pWGR := qWRB, qWBO, qWOG, qWGR
    pYBR, pYOB, pYGO, pYRG := qYBR, qYOB, qYGO, qYRG
    pWR, pWB, pWO, pWG := qWR, qWB, qWO, qWG
    pBR, pOB, pGO, pRG := qBR, qOB, qGO, qRG
    pYR, pYB, pYO, pYG := qYR, qYB, qYO, qYG
  }

AND initcostfn() BE
  { // Initialise corncostv
    FOR i = 0 TO corncostvupb DO corncostv!i := corncostm + i*corncostvsize
    // Set all elements of corncostm to 10
    FOR i = 0 TO corncostmupb DO corncostm!i := 10 // No cost will be as large as 10
    // Set all elements on the leading diagonal to 0
    FOR p = 0 TO corncostvupb DO { LET rowp = corncostm + corncostvsize*p
      rowp!p := 0 }
    // Set a cost of one for every single move
    cornrotate(0, 1, 0, mUa) // Corner 0 moves
    cornrotate(0, 3, 0, mUc)
    cornrotate(0, 3, 1, mFa)
    cornrotate(0, 4, 1, mFc)
    cornrotate(0, 4, 2, mRa)
    cornrotate(0, 1, 2, mRc)
    cornrotate(1, 2, 0, mUa) // Corner 1 moves
cornrotate(1, 0, 0, mUc)
cornrotate(1, 0, 1, mRa)
cornrotate(1, 5, 1, mRc)
cornrotate(1, 5, 2, mBa)
cornrotate(1, 2, 2, mBc)
cornrotate(2, 3, 0, mUa)  // Corner 2 moves
cornrotate(2, 1, 0, mUc)
cornrotate(2, 1, 1, mBa)
cornrotate(2, 6, 1, mBc)
cornrotate(2, 6, 2, mLa)
cornrotate(2, 3, 2, mLc)
cornrotate(3, 0, 0, mUa)  // Corner 3 moves
cornrotate(3, 2, 0, mUc)
cornrotate(3, 2, 1, mLa)
cornrotate(3, 7, 1, mLc)
cornrotate(3, 7, 2, mFa)
cornrotate(3, 0, 2, mFc)
cornrotate(4, 7, 0, mDa)  // Corner 4 moves
cornrotate(4, 5, 0, mDc)
cornrotate(4, 5, 1, mRa)
cornrotate(4, 0, 1, mRc)
cornrotate(4, 0, 2, mFa)
cornrotate(4, 7, 2, mFc)
cornrotate(5, 4, 0, mDa)  // Corner 5 moves
cornrotate(5, 6, 0, mDc)
cornrotate(5, 6, 1, mBa)
cornrotate(5, 1, 1, mBc)
cornrotate(5, 1, 2, mRa)
cornrotate(5, 4, 2, mRc)
cornrotate(6, 5, 0, mDa)  // Corner 6 moves
cornrotate(6, 7, 0, mDc)
cornrotate(6, 7, 1, mLa)
cornrotate(6, 2, 1, mLc)
cornrotate(6, 2, 2, mBa)
cornrotate(6, 5, 2, mBc)
cornrotate(7, 6, 0, mDa)  // Corner 7 moves
cornrotate(7, 4, 0, mDc)
cornrotate(7, 4, 1, mFa)
cornrotate(7, 3, 1, mFc)
cornrotate(7, 3, 2, mLa)
cornrotate(7, 6, 2, mLc)

// writef("*ncorner cost matrix before applying Ffloyd's algorithm*n")
// prcornmat(corncostm, corncostvsize)

// Apply Ffloyd's algorithm
ffloyd(corncostm, corncostvsize)

// writef("*ncorner cost matrix after applying Ffloyd's algorithm*n")
// prcornmat(corncostm, corncostvsize)
// abort(2000)

// Initialise edgecostv
FOR i = 0 TO edgecostvupb DO edgecostv!i := edgecostm + i*edgecostvsize

// Set all elements of edgecostm to 10
FOR i = 0 TO edgecostmupb DO
    edgecostm!i := 10 // No cost will be as large as 10

// Set all elements on the leading diagonal to 0
FOR p = 0 TO edgecostvupb DO
    LET rowp = edgecostm + edgecostvsize*p
    rowp!p := 0

// Set a cost of one for every single move
edgerotate( 0,  1,  0, mUa) // Edge 0 moves
edgerotate( 0,  3,  0, mUc)
edgerotate( 0,  7,  1, mFa)
edgerotate( 0,  4,  0, mFc)
edgerotate( 1,  2,  0, mUa) // Edge 1 moves
edgerotate( 1,  0,  0, mUc)
edgerotate( 1,  4,  1, mRa)
edgerotate( 1,  5,  0, mRc)
edgerotate( 2,  3,  0, mUa) // Edge 2 moves
edgerotate( 2,  1,  0, mUc)
edgerotate( 2,  5,  1, mBa)
edgerotate( 2,  6,  0, mBc)
edgerotate( 3,  0,  0, mUa) // Edge 3 moves
edgerotate( 3,  2,  0, mUc)
edgerotate( 3,  6,  1, mLa)
edgerotate( 3,  7,  0, mLc)
4.27. THE RUBIK CUBE

edgerotate(4, 0, 0, mFa) // Edge 4 moves
edgerotate(4, 8, 0, mFc)
edgerotate(4, 9, 1, mRa)
edgerotate(4, 1, 1, mRc)
edgerotate(5, 1, 0, mRa) // Edge 5 moves
edgerotate(5, 9, 0, mRc)
edgerotate(5, 10, 1, mBa)
edgerotate(5, 2, 1, mBc)
edgerotate(6, 2, 0, mBa) // Edge 6 moves
edgerotate(6, 10, 0, mBc)
edgerotate(6, 11, 1, mLc)
edgerotate(6, 3, 1, mLc)
edgerotate(7, 3, 0, mLc) // Edge 7 moves
edgerotate(7, 11, 0, mLc)
edgerotate(7, 8, 1, mFa)
edgerotate(7, 0, 1, mFc)
edgerotate(8, 11, 0, mDa) // Edge 8 moves
edgerotate(8, 9, 0, mDc)
edgerotate(8, 4, 0, mFa)
edgerotate(8, 7, 1, mFc)
edgerotate(9, 8, 0, mDa) // Edge 9 moves
edgerotate(9, 10, 0, mDc)
edgerotate(9, 5, 0, mRa)
edgerotate(9, 4, 1, mRc)
edgerotate(10, 9, 0, mDa) // Edge 10 moves
edgerotate(10, 11, 0, mDc)
edgerotate(10, 6, 0, mBa)
edgerotate(10, 5, 1, mBc)
edgerotate(11, 10, 0, mDa) // Edge 11 moves
edgerotate(11, 8, 0, mDc)
edgerotate(11, 7, 0, mLc)
edgerotate(11, 6, 1, mLc)

//writef("*nedge cost matrix before applying Ffloyd's algorithm*n")
//predgemat(edgecostm, edgecostvsize)

// Apply Ffloyd's algorithm
ffloyd(edgecostm, edgecostvsize)
//writef("*nedge cost matrix after applying Ffloyd’s algorithm*n")
//predgemat(edgecostm, edgecostvsize)
//abort(3000)
}

AND cornrotate(c1, c2, rot, move) BE
{ // rot = 0 no change in orientation, ie 0->0, 1->1 and 2->2
  // rot = 1 corner piece rotated anti-clockwise, ie 0->1, 1->2 and 2->0
  // rot = 2 corner piece rotated clockwise, ie 0->2, 1->0 and 2->1
  FOR o1 = 0 TO 2 DO // The three orientations of the piece at corner c1
  { LET o2 = (o1 + rot) MOD 3 // orientation when moved to corner c2
    LET p = c1*3 + o1
    LET rowp = corncostv!p
    LET q = c2*3 + o2
    // A piece at corner c1 with orientation o1 can be moved to
    // corner c2 with orientation o2 by a single move.
    rowp!q := 1
  }
  }

AND edgerotate(e1, e2, flip, move) BE
{ // flip = 0 no change in orientation, ie 0->0 and 1->1
  // flip = 1 edge piece flipped, ie 0->1 and 1->0
  FOR o1 = 0 TO 1 DO // The two orientations of the piece at edge e1
  { LET o2 = o1 XOR flip // orientation when moved to edge e2
    LET p = e1*2 + o1
    LET rowp = edgecostv!p
    LET q = e2*2 + o2
    // A piece at edge e1 with orientation o1 can be moved to
    // edge e2 with orientation o2 by a single move.
    rowp!q := 1
  }
  }

AND ffloyd(m, n) BE FOR k = 0 TO n-1 DO
{ LET rowk = m + k*n
  FOR i = 0 TO n-1 DO
  { LET rowi = m + i*n
    LET mik = rowi!k
    FOR j = 0 TO n-1 DO
    { LET mkj = rowk!j
      LET d = mik+mkj
      IF rowi!j > d DO rowi!j := d
    }
4.27. THE RUBIK CUBE

AND prcornmat(m, n) BE
{ newline() FOR i = 0 TO n-1 DO {
  LET rowi = m + i*n
  writeln("row %i2: ", i)
  FOR j = 0 TO n-1 DO {
    LET d = rowi!j
    TEST d=10 THEN writeln(".")
    ELSE writeln("%n", rowi!j)
    IF j MOD 3 = 2 DO wrch(' ')
  }
  IF i MOD 3 = 2 DO newline()
}
newline()

AND predgemat(m, n) BE
{ newline() FOR i = 0 TO n-1 DO {
  LET rowi = m + i*n
  writeln("row %i2: ", i)
  FOR j = 0 TO n-1 DO {
    LET d = rowi!j
    TEST d=10 THEN writeln(".")
    ELSE writeln("%n", rowi!j)
    IF j MOD 2 = 1 DO wrch(' ')
  }
  IF i MOD 2 = 1 DO newline()
}
newline()

AND prmoves(moves) BE IF moves DO
{ prmoves(moves>>8)
  wrch(moves&255)
}

AND corncost(piece, corner) = VALOF
{ LET d = piece MOD 3
  LET res = corncostv!(piece-d)!(3*corner+d)
  // writeln("corner piece = %n/%n corner = %n cost = %n*n",
  //     piece/3, piece MOD 3, corner, res)
RESULTIS res
}

AND edgecost(piece, edge) = VALOF
{ LET res = edgecostv!piece!(2*edge)
  //writef("edge piece = %i2/\n edge = %i2 cost = %\n*n", 
  // piece/2, piece MOD 2, edge, res)
  RESULTIS res
}

AND costfn() = VALOF
{ // Return the cost of the position in qWRB, etc
  // This is the sum of the minimum number of moves
  // required for each piece.
  LET c = ?
  //writef("costfn: entered*\n")
  c := corncost(qWRB, cWRB)
  c := c + corncost(qWBO, cWBO)
  c := c + corncost(qWOG, cWOG)
  c := c + corncost(qWGR, cWGR)
  c := c + corncost(qYBR, cYBR)
  c := c + corncost(qYOB, cYOB)
  c := c + corncost(qYGO, cYGO)
  c := c + corncost(qYRG, cYRG)
  c := c + edgecost(qWR, eWR)
  c := c + edgecost(qWB, eWB)
  c := c + edgecost(qWO, eWO)
  c := c + edgecost(qWG, eWG)
  c := c + edgecost(qBR, eBR)
  c := c + edgecost(qOB, eOB)
  c := c + edgecost(qGO, eGO)
  c := c + edgecost(qRG, eRG)
  c := c + edgecost(qYR, eYR)
  c := c + edgecost(qYB, eYB)
  c := c + edgecost(qYO, eYO)
  c := c + edgecost(qYG, eYG)
  //writef("costfn: cost = %n*n", c)
  //abort(4000)

  RESULTIS c * c // Square to discourage pieces many moves
  // from their required positions.
}
AND score(node) = VALOF
{ cube2pieces(node, @qWRB)
    RESULTIS score()+goalscore(node)
}

AND score() = costfn()

AND prcosts() BE
{ newline()
    prcorncost("WRB0: ", WRB0)
    prcorncost("WRB1: ", WRB1)
    prcorncost("WRB2: ", WRB2)
    newline()
    prcorncost("WBO0: ", WBO0)
    prcorncost("WBO1: ", WBO1)
    prcorncost("WBO2: ", WBO2)
    newline()
    prcorncost("WOG0: ", WOG0)
    prcorncost("WOG1: ", WOG1)
    prcorncost("WOG2: ", WOG2)
    newline()
    prcorncost("WGR0: ", WGR0)
    prcorncost("WGR1: ", WGR1)
    prcorncost("WGR2: ", WGR2)
    newline()
    prcorncost("YBR0: ", YBR0)
    prcorncost("YBR1: ", YBR1)
    prcorncost("YBR2: ", YBR2)
    newline()
    prcorncost("YOB0: ", YOB0)
    prcorncost("YOB1: ", YOB1)
    prcorncost("YOB2: ", YOB2)
    newline()
    prcorncost("YGO0: ", YGO0)
    prcorncost("YGO1: ", YGO1)
    prcorncost("YGO2: ", YGO2)
    newline()
    prcorncost("YRG0: ", YRG0)
    prcorncost("YRG1: ", YRG1)
    prcorncost("YRG2: ", YRG2)

    newline()

    predgecost("WR0: ", WRO)
predgecost("WR1: ", WR1)
newline()
predgecost("WB0: ", WB0)
predgecost("WB1: ", WB1)
newline()
predgecost("WO0: ", WO0)
predgecost("WO1: ", WO1)
newline()
predgecost("WG0: ", WG0)
predgecost("WG1: ", WG1)
newline()
predgecost("BR0: ", BR0)
predgecost("BR1: ", BR1)
newline()
predgecost("OB0: ", OB0)
predgecost("OB1: ", OB1)
newline()
predgecost("GO0: ", GO0)
predgecost("GO1: ", GO1)
newline()
predgecost("RG0: ", RG0)
predgecost("RG1: ", RG1)
newline()
predgecost("YR0: ", YR0)
predgecost("YR1: ", YR1)
newline()
predgecost("YB0: ", YB0)
predgecost("YB1: ", YB1)
newline()
predgecost("YO0: ", YO0)
predgecost("YO1: ", YO1)
newline()
predgecost("YG0: ", YG0)
predgecost("YG1: ", YG1)
newline()
}

AND prcorncost(str, piece) BE
{ writef("%s: ", str)
  FOR corner = 0 TO 7 DO writef(" %i3", corncost(piece, corner))
  newline()
}
AND predgecost(str, piece) BE
{ writef("%s: ", str)
   FOR edge = 0 TO 11 DO writef(" %i3", edgecost(piece, edge))
   newline()
}

AND prsolution(node) BE
{ IF s_prev!node DO
 { prsolution(s_prev!node)
   writef("move %c*n", s_move!node)
 }
   prcube(node)
}

AND wrcornerpiece(piece) BE
{ SWITCHON piece/3 INTO
 {  
   CASE cWRB: writef(" WRB"); END_CASE
   CASE cWBO: writef(" WBO"); END_CASE
   CASE cWOG: writef(" WOG"); END_CASE
   CASE cWGR: writef(" WGR"); END_CASE
   CASE cYBR: writef(" YBR"); END_CASE
   CASE cYOB: writef(" YOB"); END_CASE
   CASE cYGO: writef(" YGO"); END_CASE
   CASE cYRG: writef(" YRG"); END_CASE
 
   writef("%n", piece MOD 3)
 }
}

AND wredgepiece(piece) BE
{ SWITCHON piece/2 INTO
{  
   CASE eWR: writef(" WR"); END_CASE
   CASE eWB: writef(" WB"); END_CASE
   CASE eWO: writef(" WO"); END_CASE
   CASE eWG: writef(" WG"); END_CASE
   CASE eBR: writef(" BR"); END_CASE
   CASE eOB: writef(" OB"); END_CASE
   CASE eGO: writef(" GO"); END_CASE
   CASE eRG: writef(" RG"); END_CASE
   CASE eYB: writef(" YB"); END_CASE
   CASE eYO: writef(" YO"); END_CASE
   CASE eYG: writef(" YG"); END_CASE
}
CASE eYR: writef(" YR"); ENDCASE

writef("%n ", piece MOD 2)

AND prpieces(pieces) BE
{ LET c = VEC 4
  pieces2cube(pieces, c)
  wrcornerpiece(c%0)
  wrcornerpiece(c%1)
  wrcornerpiece(c%2)
  wrcornerpiece(c%3)
  wrcornerpiece(c%4)
  wrcornerpiece(c%5)
  wrcornerpiece(c%6)
  wrcornerpiece(c%7)
  newline()
  wredgepiece(c%8)
  wredgepiece(c%9)
  wredgepiece(c%10)
  wredgepiece(c%11)
  wredgepiece(c%12)
  wredgepiece(c%13)
  wredgepiece(c%14)
  wredgepiece(c%15)
  wredgepiece(c%16)
  wredgepiece(c%17)
  wredgepiece(c%18)
  wredgepiece(c%19)
  newline()
  prcube(c)
}

AND prnode(node) BE
{ //writef("node=%n prev=%n*n", 
   // node, s_prev!node)
   prcube(node)
}

AND prcube(cube) BE
{ /* Typical output is either

W W W W W W W G G G G G G G G R R R R R R R R B B B B B B B B B B B B O O O O O O O O O O O O O O O O O Y Y Y Y Y Y Y Y Y Y Y Y

or

W W W W W W W G G G G G G G G R R R R R R R R B B B B B B B B B B B B O O O O O O O O O O O O O O O O O Y Y Y Y Y Y Y Y Y Y Y Y

or

4.27. THE RUBIK CUBE

```c
/*
cube2cols(cube, colour)

IF compact DO
{ writef("%c%c%c%c%c%c%c%c%c ", colour!0, colour!1, colour!2,
colour!3, colour!4, colour!5,
colour!6, colour!7, colour!8)
writef("%c%c%c%c%c%c%c%c%c ", colour!36, colour!37, colour!38,
colour!39, colour!40, colour!41,
colour!42, colour!43, colour!44)
writef("%c%c%c%c%c%c%c%c%c ", colour! 9, colour!10, colour!11,
colour!12, colour!13, colour!14,
colour!15, colour!16, colour!17)
writef("%c%c%c%c%c%c%c%c%c ", colour!18, colour!19, colour!20,
colour!21, colour!22, colour!23,
colour!24, colour!25, colour!26)
writef("%c%c%c%c%c%c%c%c%c ", colour!27, colour!28, colour!29,
colour!30, colour!31, colour!32,
colour!33, colour!34, colour!35)
writef("%c%c%c%c%c%c%c%c%n", colour!45, colour!46, colour!47,
colour!48, colour!49, colour!50,
colour!51, colour!52, colour!53)
RETURN
}

writef(" %c %c %c%n", colour!0, colour!1, colour!2)
writef(" %c %c %c%n", colour!3, colour!4, colour!5)
writef(" %c %c %c%n", colour!6, colour!7, colour!8)
*/
```
CHAPTER 4. THE BCPL CINTCODE SYSTEM

writef(" %c %c %c ", colour!36, colour!37, colour!38)
writef(" %c %c %c ", colour! 9, colour!10, colour!11)
writef(" %c %c %c ", colour!18, colour!19, colour!20)
writef(" %c %c %c*n", colour!27, colour!28, colour!29)
writef(" %c %c %c ", colour!39, colour!40, colour!41)
writef(" %c %c %c ", colour!12, colour!13, colour!14)
writef(" %c %c %c ", colour!21, colour!22, colour!23)
writef(" %c %c %c*n", colour!30, colour!31, colour!32)
writef(" %c %c %c ", colour!42, colour!43, colour!44)
writef(" %c %c %c ", colour!15, colour!16, colour!17)
writef(" %c %c %c *n", colour!24, colour!25, colour!26)
writef(" %c %c %c*n", colour!33, colour!34, colour!35)
writef(" %c %c %c*n", colour!45, colour!46, colour!47)
writef(" %c %c %c*n", colour!48, colour!49, colour!50)
writef(" %c %c %c*n", colour!51, colour!52, colour!53)

AND setface(n, ch, str) BE
{ LET face = @colour!(9*n)
  UNLESS str%0=9 & capitalch(str%5)=ch DO
  { writef("Bad face colours %c %s*n", ch, str)
    errors := TRUE
  }
  FOR i = 1 TO str%0 DO face!(i-1) := capitalch(str%i)
}

AND corner(a, b, c) = VALOF SWITCHON a<<16 | b<<8 | c INTO
{ DEFAULT: writef("*nBad corner: %c%c%c*n", a, b, c)
  errors := TRUE
  RESULTIS 0
  CASE 'W'<<16 | 'R'<<8 | 'B': RESULTIS WRB0
  CASE 'B'<<16 | 'W'<<8 | 'R': RESULTIS WRB1
  CASE 'R'<<16 | 'B'<<8 | 'W': RESULTIS WRB2
  CASE 'W'<<16 | 'B'<<8 | 'O': RESULTIS WBO0
  CASE 'O'<<16 | 'W'<<8 | 'B': RESULTIS WBO1
  CASE 'B'<<16 | 'O'<<8 | 'W': RESULTIS WBO2
  CASE 'W'<<16 | 'O'<<8 | 'G': RESULTIS WOG0

4.27. THE RUBIK CUBE

CASE 'G'<<16 | 'W'<<8 | 'O': RESULTIS WOG1
CASE 'O'<<16 | 'G'<<8 | 'W': RESULTIS WOG2
CASE 'W'<<16 | 'G'<<8 | 'R': RESULTIS WGR0
CASE 'R'<<16 | 'W'<<8 | 'G': RESULTIS WGR1
CASE 'G'<<16 | 'R'<<8 | 'W': RESULTIS WGR2
CASE 'Y'<<16 | 'B'<<8 | 'R': RESULTIS YBR0
CASE 'R'<<16 | 'Y'<<8 | 'B': RESULTIS YBR1
CASE 'B'<<16 | 'R'<<8 | 'Y': RESULTIS YBR2
CASE 'Y'<<16 | 'O'<<8 | 'B': RESULTIS YOB0
CASE 'B'<<16 | 'Y'<<8 | 'O': RESULTIS YOB1
CASE 'O'<<16 | 'B'<<8 | 'Y': RESULTIS YOB2
CASE 'Y'<<16 | 'G'<<8 | 'O': RESULTIS YGO0
CASE 'G'<<16 | 'Y'<<8 | 'O': RESULTIS YGO1
CASE 'O'<<16 | 'G'<<8 | 'Y': RESULTIS YGO2
CASE 'Y'<<16 | 'R'<<8 | 'G': RESULTIS YRG0
CASE 'G'<<16 | 'Y'<<8 | 'R': RESULTIS YRG1
CASE 'R'<<16 | 'G'<<8 | 'Y': RESULTIS YRG2

AND edge(a, b) = VALOF SWITCHON a<<8 | b INTO
{ DEFAULT: writef("*nBad edge: %c%c*n", a, b)
  errors := TRUE
  RESULTIS 0

CASE 'W'<<8 | 'R': RESULTIS WR0
CASE 'R'<<8 | 'W': RESULTIS WR1
CASE 'W'<<8 | 'B': RESULTIS WB0
CASE 'B'<<8 | 'W': RESULTIS WB1
CASE 'W'<<8 | 'O': RESULTIS WO0
CASE 'O'<<8 | 'W': RESULTIS WO1
CASE 'W'<<8 | 'G': RESULTIS WG0
CASE 'G'<<8 | 'W': RESULTIS WG1
CASE 'B'<<8 | 'R': RESULTIS BR0
CASE 'R'<<8 | 'B': RESULTIS BR1
CASE 'O'<<8 | 'B': RESULTIS OB0
CASE 'B'<<8 | 'O': RESULTIS OB1
CASE 'G'<<8 | 'O': RESULTIS GO0
CASE 'O'<<8 | 'G': RESULTIS GO1
CASE 'R'<<8 | 'G': RESULTIS RG0
CHAPTER 4. THE BCPL CINTCODE SYSTEM

CASE 'G'<<8 | 'R': RESULTIS RG1

CASE 'Y'<<8 | 'R': RESULTIS YR0
CASE 'R'<<8 | 'Y': RESULTIS YR1
CASE 'Y'<<8 | 'B': RESULTIS YB0
CASE 'B'<<8 | 'Y': RESULTIS YB1
CASE 'Y'<<8 | 'O': RESULTIS YO0
CASE 'O'<<8 | 'Y': RESULTIS YO1
CASE 'Y'<<8 | 'G': RESULTIS YG0
CASE 'G'<<8 | 'Y': RESULTIS YG1

AND cols2cube(cv, cube) BE
{
    // Colour coordinates
    //
    // 0  1  2
    // 3  4  5
    // 6  7  8
    // 36 37 38  9 10 11 18 19 20 27 28 29
    // 39 40 41 12 13 14 21 22 23 30 31 32
    // 42 43 44 15 16 17 24 25 26 33 34 35
    // 45 46 47
    // 48 49 50
    // 51 52 53

cube%iWRB := corner(cv! 8, cv!11, cv!18)
cube%iWBO := corner(cv! 2, cv!20, cv!27)
cube%iWOG := corner(cv! 0, cv!29, cv!36)
cube%iWGR := corner(cv! 6, cv!38, cv! 9)
cube%iYBR := corner(cv!47, cv!24, cv!17)
cube%iYOB := corner(cv!53, cv!33, cv!26)
cube%iYGO := corner(cv!51, cv!42, cv!35)
cube%iYRG := corner(cv!45, cv!15, cv!44)

cube%iWR := edge(cv! 7, cv!10)
cube%iWB := edge(cv! 5, cv!19)
cube%iWO := edge(cv! 1, cv!28)
cube%iWG := edge(cv! 3, cv!37)

cube%iBR := edge(cv!21, cv!14)
cube%iOB := edge(cv!30, cv!23)
cube%iGO := edge(cv!39, cv!32)
cube%iRG := edge(cv!12, cv!41)

cube%iYR := edge(cv!46, cv!16)
setcornercols(cv, cube%iWRB, 8, 11, 18) // Corner pieces
setcornercols(cv, cube%iWBO, 2, 20, 27)
setcornercols(cv, cube%iWOG, 0, 29, 36)
setcornercols(cv, cube%iWGR, 6, 38, 9)
setcornercols(cv, cube%iYBR, 47, 24, 17)
setcornercols(cv, cube%iYOB, 53, 33, 26)
setcornercols(cv, cube%iYGO, 51, 42, 35)
setcornercols(cv, cube%iYRG, 45, 15, 44)

setedgecols(cv, cube%iWR, 7, 10) // edge piece, left sq, right sq
setedgecols(cv, cube%iWB, 5, 19)
setedgecols(cv, cube%iWO, 1, 28)
setedgecols(cv, cube%iW0, 3, 37)

setedgecols(cv, cube%iBR, 21, 14)
setedgecols(cv, cube%iOB, 30, 23)
setedgecols(cv, cube%iGO, 39, 32)
setedgecols(cv, cube%iRG, 12, 41)

setedgecols(cv, cube%iYR, 46, 16)
\texttt{setedgecols(cv, cube%iYB, 50, 25)}
\texttt{setedgecols(cv, cube%iY0, 52, 34)}
\texttt{setedgecols(cv, cube%iYG, 48, 43)}
\}

AND \texttt{setcornercols(cv, piece, i, j, k) BE}
\{
\hspace{1em} // i, j, k are corner face numbers in anti-clockwise order
\hspace{1em} //\texttt{writef("setcornercols %i2 %i2 %i2 %i2*n", piece, i, j, k)}
\hspace{1em} \texttt{SWITCHON piece INTO}
\hspace{2em} \{ \texttt{DEFAULT: writef("System error in setcornercols: piece=%n*n", piece)}
\hspace{3em} \}
\hspace{1em} CASE \texttt{WRB0: cv!i, cv!j, cv!k := 'W', 'R', 'B'; RETURN}
\hspace{1em} CASE \texttt{WRB1: cv!j, cv!k, cv!i := 'W', 'R', 'B'; RETURN}
\hspace{1em} CASE \texttt{WRB2: cv!k, cv!i, cv!j := 'W', 'R', 'B'; RETURN}
\hspace{1em} CASE \texttt{WBO0: cv!i, cv!j, cv!k := 'W', 'B', 'O'; RETURN}
\hspace{1em} CASE \texttt{WBO1: cv!j, cv!k, cv!i := 'W', 'B', 'O'; RETURN}
\hspace{1em} CASE \texttt{WBO2: cv!k, cv!i, cv!j := 'W', 'B', 'O'; RETURN}
\hspace{1em} CASE \texttt{WGO0: cv!i, cv!j, cv!k := 'W', 'O', 'G'; RETURN}
\hspace{1em} CASE \texttt{WGO1: cv!j, cv!k, cv!i := 'W', 'O', 'G'; RETURN}
\hspace{1em} CASE \texttt{WGO2: cv!k, cv!i, cv!j := 'W', 'O', 'G'; RETURN}
\hspace{1em} CASE \texttt{WGR0: cv!i, cv!j, cv!k := 'W', 'G', 'R'; RETURN}
\hspace{1em} CASE \texttt{WGR1: cv!j, cv!k, cv!i := 'W', 'G', 'R'; RETURN}
\hspace{1em} CASE \texttt{WGR2: cv!k, cv!i, cv!j := 'W', 'G', 'R'; RETURN}
\hspace{1em} CASE \texttt{YBR0: cv!i, cv!j, cv!k := 'Y', 'B', 'R'; RETURN}
\hspace{1em} CASE \texttt{YBR1: cv!j, cv!k, cv!i := 'Y', 'B', 'R'; RETURN}
\hspace{1em} CASE \texttt{YBR2: cv!k, cv!i, cv!j := 'Y', 'B', 'R'; RETURN}
\hspace{1em} CASE \texttt{YBO0: cv!i, cv!j, cv!k := 'Y', 'O', 'B'; RETURN}
\hspace{1em} CASE \texttt{YBO1: cv!j, cv!k, cv!i := 'Y', 'O', 'B'; RETURN}
\hspace{1em} CASE \texttt{YBO2: cv!k, cv!i, cv!j := 'Y', 'O', 'B'; RETURN}
\hspace{1em} CASE \texttt{YGO0: cv!i, cv!j, cv!k := 'Y', 'G', 'O'; RETURN}
\hspace{1em} CASE \texttt{YGO1: cv!j, cv!k, cv!i := 'Y', 'G', 'O'; RETURN}
\hspace{1em} CASE \texttt{YGO2: cv!k, cv!i, cv!j := 'Y', 'G', 'O'; RETURN}
\hspace{1em} CASE \texttt{YRG0: cv!i, cv!j, cv!k := 'Y', 'R', 'G'; RETURN}
\hspace{1em} CASE \texttt{YRG1: cv!j, cv!k, cv!i := 'Y', 'R', 'G'; RETURN}
\hspace{1em} CASE \texttt{YRG2: cv!k, cv!i, cv!j := 'Y', 'R', 'G'; RETURN}
\}
\}

AND \texttt{setedgecols(cv, piece, i, j) BE}
\{
\hspace{1em} //\texttt{writef("setedgecols(%i2 %i2 %i2*n", piece, i, j)}
\hspace{1em} \texttt{SWITCHON piece INTO}
\hspace{2em} \{ \texttt{DEFAULT: writef("System error in setedgecols: piece=%n*n", piece)}
\hspace{3em} \}
\hspace{1em} CASE \texttt{WRB0: cv!i, cv!j, cv!k := 'W', 'R', 'B'; RETURN}
\hspace{1em} CASE \texttt{WRB1: cv!j, cv!k, cv!i := 'W', 'R', 'B'; RETURN}
\hspace{1em} CASE \texttt{WRB2: cv!k, cv!i, cv!j := 'W', 'R', 'B'; RETURN}
\hspace{1em} CASE \texttt{WBO0: cv!i, cv!j, cv!k := 'W', 'B', 'O'; RETURN}
\hspace{1em} CASE \texttt{WBO1: cv!j, cv!k, cv!i := 'W', 'B', 'O'; RETURN}
\hspace{1em} CASE \texttt{WBO2: cv!k, cv!i, cv!j := 'W', 'B', 'O'; RETURN}
\hspace{1em} CASE \texttt{WGO0: cv!i, cv!j, cv!k := 'W', 'O', 'G'; RETURN}
\hspace{1em} CASE \texttt{WGO1: cv!j, cv!k, cv!i := 'W', 'O', 'G'; RETURN}
\hspace{1em} CASE \texttt{WGO2: cv!k, cv!i, cv!j := 'W', 'O', 'G'; RETURN}
\hspace{1em} CASE \texttt{WGR0: cv!i, cv!j, cv!k := 'W', 'G', 'R'; RETURN}
\hspace{1em} CASE \texttt{WGR1: cv!j, cv!k, cv!i := 'W', 'G', 'R'; RETURN}
\hspace{1em} CASE \texttt{WGR2: cv!k, cv!i, cv!j := 'W', 'G', 'R'; RETURN}
\hspace{1em} CASE \texttt{YBR0: cv!i, cv!j, cv!k := 'Y', 'B', 'R'; RETURN}
\hspace{1em} CASE \texttt{YBR1: cv!j, cv!k, cv!i := 'Y', 'B', 'R'; RETURN}
\hspace{1em} CASE \texttt{YBR2: cv!k, cv!i, cv!j := 'Y', 'B', 'R'; RETURN}
\hspace{1em} CASE \texttt{YBO0: cv!i, cv!j, cv!k := 'Y', 'O', 'B'; RETURN}
\hspace{1em} CASE \texttt{YBO1: cv!j, cv!k, cv!i := 'Y', 'O', 'B'; RETURN}
\hspace{1em} CASE \texttt{YBO2: cv!k, cv!i, cv!j := 'Y', 'O', 'B'; RETURN}
\hspace{1em} CASE \texttt{YGO0: cv!i, cv!j, cv!k := 'Y', 'G', 'O'; RETURN}
\hspace{1em} CASE \texttt{YGO1: cv!j, cv!k, cv!i := 'Y', 'G', 'O'; RETURN}
\hspace{1em} CASE \texttt{YGO2: cv!k, cv!i, cv!j := 'Y', 'G', 'O'; RETURN}
\hspace{1em} CASE \texttt{YRG0: cv!i, cv!j, cv!k := 'Y', 'R', 'G'; RETURN}
\hspace{1em} CASE \texttt{YRG1: cv!j, cv!k, cv!i := 'Y', 'R', 'G'; RETURN}
\hspace{1em} CASE \texttt{YRG2: cv!k, cv!i, cv!j := 'Y', 'R', 'G'; RETURN}
\}
4.27. THE RUBIK CUBE

CASE WR0: cv!i, cv!j := 'W', 'R'; RETURN
CASE WR1: cv!j, cv!i := 'W', 'R'; RETURN
CASE WB0: cv!i, cv!j := 'W', 'B'; RETURN
CASE WB1: cv!j, cv!i := 'W', 'B'; RETURN
CASE WO0: cv!i, cv!j := 'W', 'O'; RETURN
CASE WO1: cv!j, cv!i := 'W', 'O'; RETURN
CASE WG0: cv!i, cv!j := 'W', 'G'; RETURN
CASE WG1: cv!j, cv!i := 'W', 'G'; RETURN
CASE BR0: cv!i, cv!j := 'B', 'R'; RETURN
CASE BR1: cv!j, cv!i := 'B', 'R'; RETURN
CASE OB0: cv!i, cv!j := 'O', 'B'; RETURN
CASE OB1: cv!j, cv!i := 'O', 'B'; RETURN
CASE GO0: cv!i, cv!j := 'G', 'O'; RETURN
CASE GO1: cv!j, cv!i := 'G', 'O'; RETURN
CASE RG0: cv!i, cv!j := 'R', 'G'; RETURN
CASE RG1: cv!j, cv!i := 'R', 'G'; RETURN
CASE YR0: cv!i, cv!j := 'Y', 'R'; RETURN
CASE YR1: cv!j, cv!i := 'Y', 'R'; RETURN
CASE YB0: cv!i, cv!j := 'Y', 'B'; RETURN
CASE YB1: cv!j, cv!i := 'Y', 'B'; RETURN
CASE YO0: cv!i, cv!j := 'Y', 'O'; RETURN
CASE YO1: cv!j, cv!i := 'Y', 'O'; RETURN
CASE YG0: cv!i, cv!j := 'Y', 'G'; RETURN
CASE YG1: cv!j, cv!i := 'Y', 'G'; RETURN

AND goalscore(cube) = VALOF
{ LET k = ?
  LET piece = ?

  //printf("goalscore:*n")
  //prnode(cube)
  //printf("upper edges WR=\%n/%n WB=\%n/%n WO=\%n/%n WG=\%n/%n*n",
  //   cube%iWR, WR0,
  //   cube%iWB, WB0,
  //   cube%iWO, WO0,
  //   cube%iWG, WG0)

  // Upper edges
  // Penalties
  // right edge wrong orientation 900
  // wrong edge 1000
k := 4*1000
piece := cube%iWR
IF piece=WR0 DO k := k-1000
IF piece=WR1 DO k := k-100
piece := cube%iWB
IF piece=WB0 DO k := k-1000
IF piece=WB1 DO k := k-100
piece := cube%iWO
IF piece=WO0 DO k := k-1000
IF piece=WO1 DO k := k-100
piece := cube%iWG
IF piece=WG0 DO k := k-1000
IF piece=WG1 DO k := k-100

// If k=0 upper four edges are correct

// Upper corners
// Penalties
// right corner wrong orientation 700
// wrong corner 800

k := k + 4*800

piece := cube%iWBR
IF piece=WBR0 DO k := k-800
IF piece=WBR1 DO k := k-100
IF piece=WBR2 DO k := k-100
piece := cube%iWBO
IF piece=WBO0 DO k := k-800
IF piece=WBO1 DO k := k-100
IF piece=WBO2 DO k := k-100
piece := cube%iWOG
IF piece=WOG0 DO k := k-800
IF piece=WOG1 DO k := k-100
IF piece=WOG2 DO k := k-100
piece := cube%iWGR
IF piece=WRG0 DO k := k-800
IF piece=WRG1 DO k := k-100
IF piece=WRG2 DO k := k-100

// If k=0 upper layer is now correct

// Middle layer edges
// Penalties
// right edge wrong orientation 250
// wrong edge
300

k := k + 4*300

piece := cube%iBR
IF piece=BR0 DO k := k-300
IF piece=BR1 DO k := k- 50

piece := cube%iOB
IF piece=OB0 DO k := k-300
IF piece=OB1 DO k := k- 50

piece := cube%iGO
IF piece=GO0 DO k := k-300
IF piece=GO1 DO k := k- 50

piece := cube%iRG
IF piece=RG0 DO k := k-300
IF piece=RG1 DO k := k- 50

// If k=0 upper and middle layers are now correct

// Lower level edges
// Penalties
// right edge wrong orientation 30
// wrong edge 40

k := k + 4*40

piece := cube%iYR
IF piece=YR0 DO k := k-40
IF piece=YR1 DO k := k-10

piece := cube%iYB
IF piece=YB0 DO k := k-40
IF piece=YB1 DO k := k-10

piece := cube%iYO
IF piece=YO0 DO k := k-40
IF piece=YO1 DO k := k-10

piece := cube%iYG
IF piece=YG0 DO k := k-40
IF piece=YG1 DO k := k-10

// If k=0 upper and middle layers are now correct
// and down face edges are correct

// Lower level corners
// Penalties
// right edge wrong orientation 15
// wrong edge

k := k + 4 * 20

piece := cube%iYBR
IF piece=YBR0 DO k := k - 20
IF piece=YBR1 DO k := k - 5
IF piece=YBR2 DO k := k - 5

piece := cube%iYOB
IF piece=YOB0 DO k := k - 20
IF piece=YOB1 DO k := k - 5
IF piece=YOB2 DO k := k - 5

piece := cube%iYGO
IF piece=YGO0 DO k := k - 20
IF piece=YGO1 DO k := k - 5
IF piece=YGO2 DO k := k - 5

piece := cube%iYRG
IF piece=YRG0 DO k := k - 20
IF piece=YRG1 DO k := k - 5
IF piece=YRG2 DO k := k - 5

// If k=0 all positions are correct so the Rubik Cube has been solved
//writef("goalscore: returning %n*n", k)
//abort(9000)
RESULTIS k
}

4.28 Simple series

We have seen that the largest number we can represent in an unsigned 32-bit word is

\[ 1 + 2 + 2^2 + 2^3 + \ldots + 2^{31} \]

This is perfectly understandable and is called a series, but mathematicians do not normally like to use dots since they introduce possible misunderstandings of what is being omitted. They generally prefer the following notation.

\[ \sum_{i=0}^{31} 2^i \]
but in this document I will almost always use the dot notation. We can generalise this series to term $n$, replacing the constant 2 by some arbitrary value $x$ and call the sum $s$, namely

$$s = 1 + x + x^2 + x^3 + \ldots + x^n$$

We can easily make a simple formula for $s$ by considering $s$ multiplied by $(x - 1)$, that is

$$s(x - 1) = (1 + x + x^2 + x^3 + \ldots + x^n) \times x - (1 + x + x^2 + x^3 + \ldots + x^n)$$
$$= (x + x^2 + x^3 + \ldots + x^{n+1}) - (1 + x + x^2 + x^3 + \ldots + x^n)$$
$$= x^{n+1} - 1$$

So

$$s = \frac{x^{n+1} - 1}{x - 1}$$

So for our original series, $x = 2$ and $n = 31$ gives us

$$s = \frac{2^{32} - 1}{2 - 1} = 2^{32} - 1 = 4294967295$$

Notice that with $x = 2$ as $n$ gets larger so does the sum. When $x = 2$, the series is said to diverge as $n$ tends to infinity (an incredibly large number often represented by $\infty$). But what happens if $x < 1$. Let us try $x = \frac{1}{2}$ and $n = \infty$.

$$s = \left(\frac{1}{2}\right)^\infty - 1 = 0 - 1 = 2$$

In the above derivation, we took $\left(\frac{1}{2}\right)^\infty$ to be zero since multiplying 1 by $\frac{1}{2}$ a huge number of times gets so small its value can be ignored.

As a demonstration of the use of vectors and functions we will look a program called `eva12.b` that calculates $s$ to 2000 decimal places to show that it is indeed 2. It starts as follows.
GET "libhdr"

GLOBAL {
    sum:ug
    term
    upb
}

LET start() = VALOF
{ upb := 2004/4  // Each element holds 4 decimal digits
    // and there are 4 guard digits at the end.
    sum := getvec(upb)
    term := getvec(upb)

    settok(sum, 0)
    sum!upb := 5000  // Add 1/2 at digit position 2000 for rounding
    settok(term, 1)

    UNTIL iszero(term) DO
    { add(sum, term)
        divbyk(term, 2)
    }

    // Write out the sum to 40 decimal places
    writef("*nsum = %n.", sum!0)
    FOR i = 1 TO 10 DO writef("%4z ", sum!i)
    newline()
}

fin:
    freevec(sum)
    freevec(term)
    RESULTIS 0
}

It uses the vector $\text{sum}$ to hold the summation of all the terms and $\text{term}$ to hold the next term to add to $\text{sum}$. Both $\text{sum}$ and $\text{term}$ are vectors with upperbound $2004/4$ which is sufficient to hold numbers with 4 decimal digits before the decimal point and 2000 digits after the decimal point together with a further 4 guard digits at the end. $\text{sum}$ and $\text{term}$ are initialised by calls of $\text{settok}$, described later, and 5000 is placed in the last element of $\text{sum}$ which corresponds to adding $1/2$ at decimal digit position 2000. This causes appropriate rounding to take place. The $\text{UNTIL}$ loop adds $\text{term}$ to $\text{sum}$ dividing $\text{term}$ by 2 each time until $\text{term}$ represents zero. $\text{sum}$ is then output to 40 decimal places as follows:

$\text{sum} = 2.0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000$
as expected.

The rest of the program defines the functions `settok`, `add`, `divbyk` and `iszero` as follows.

```
AND settok(v, k) BE
{ v!0 := k
  FOR i = 1 TO upb DO v!i := 0
}

AND add(a, b) BE
{ LET c = 0
  FOR i = upb TO 0 BY -1 DO
    { LET d = c + a!i + b!i
      a!i := d MOD 10000
      c := d / 10000
    }
  }
}

AND divbyk(v, k) BE
{ LET c = 0
  FOR i = 0 TO upb DO
    { LET d = c*10000 + v!i
      v!i := d / k
      c := d MOD k
    }
  }
}

AND iszero(v) = VALOF
{ FOR i = upb TO 0 BY -1 IF v!i RESULTIS FALSE
  RESULTIS TRUE
}
```

The function `settok` is self explanatory. Notice that `add` performs the addition from the least significant end using the variable `c` to hold the carry. `divbyk` performs short division from the most significant end, again using `c` to hold the carry. Finally, `iszero` only returns `TRUE` if every element of `v` is zero.

### 4.29  \( e \) to 2000 decimal places

The constant \( e \) which has a value of approximately 2.71828 is one of the most important constants in mathematics. It can be defined in many ways, but the one we will use in this section is:

\[
e = 1 + 1 + \frac{1}{2} + \frac{1}{3!} + \ldots + \frac{1}{n!} + \ldots
\]
where \( n! \) stands for \( n \) factorial \((1 \times 1 \times 2 \times 3 \times \ldots \times n)\).

This section presents a simple program (evale.b) that computes \( e \) to 2000 decimal places. As with the previous program, it is primarily an example of the use of vectors and functions, and, as with the previous program, it uses high precision numbers using vectors whose elements each contain 4 decimal digits. It is convenient to think of these elements as digits of radix 10000. A radix of 10000 was chosen because 10000\(^2\) easily fits in a 32-bit word, but 100000\(^2\) does not. The program starts as follows.

GET "libhdr"

GLOBAL {
  sum:ug // The sum of terms so far
  term // The next term to add to sum
  tab // The frequency counts of the digits of \( e \)
  digcount
  digits // The number of decimal digits to calculate
  upb
}

LET start() = VALOF
  LET n = 1
  digits := 2000 // Calculate \( e \) to 2000 decimal places
  upb := (digits+10)/4 // add ten guard digits
  tab := getvec(9) // for digit frequency counts
  sum := getvec(upb) // will hold the sum of the series
  term := getvec(upb) // the next term in the series to add to sum

UNLESS tab & sum & term DO
  { writef("Unable to allocate vectors\n")
    GOTO fin
  }

settok(sum, 1) // Initial value of sum
settok(term, 1) // The first term to add

UNTIL iszero(term) DO // Until the term is zero
  { add(sum, term) // Add the term to sum
    n := n + 1
    divbyk(term, n) // Calculate the next term
  }

// Write out \( e \)
writes("\*ne = \*n")
print(sum)
4.29.  *E TO 2000 DECIMAL PLACES*

```plaintext

// Write out the digit frequency counts
writes("*nDigit counts*n")
FOR i = 0 TO 9 DO writef("%n:%i3 ", i, tab!i)
newline()

fin:
  freevec(tab)
  freevec(sum)
  freevec(term)
  RESULTIS 0
}

The program ends with the definitions of the functions used, most of which we have already seen.

AND settok(v, k) BE
{ v!0 := k // Set the integer part
  FOR i = 1 TO upb DO v!i := 0 // Clear all fractional digits
}

AND add(a, b) BE
{ LET c = 0
  FOR i = upb TO 0 BY -1 DO
    { LET d = c + a!i + b!i
      a!i := d MOD 10000
      c := d / 10000
    }
}

AND divbyk(v, k) BE
{ LET c = 0
  FOR i = 0 TO upb DO
    { LET d = c*10000 + v!i
      v!i := d / k
      c := d MOD k
    }
}

AND iszero(v) = VALOF
{ FOR i = upb TO 0 BY -1 IF v!i RESULTIS FALSE
  RESULTIS TRUE
}

The final two functions output the high precision number held in v as a sequence of decimal digits.
```
AND print(v) BE
{ FOR i = 0 TO 9 DO tab!i := 0 // Clear the frequency counts
digcount := 0
writef(" %i4.", v!0)
FOR i = 1 TO upb DO
{ IF i MOD 15 = 0 DO writes("*n ")
wrpn(v!i, 4)
wrch('*s')
}
newline()
}

AND wrpn(n, d) BE
{ IF d>1 DO wrpn(n/10, d-1)
IF digcount>=digits RETURN
n := n MOD 10
tab!n := tab!n + 1
wrch(n+'0')
digcount := digcount+1
}

When the program is run its output is as follows.

e =
  2.7182 8182 8459 0452 3536 0287 4713 5266 2497 7572 4709 3699 9595 7496
  6967 6277 2407 6630 3535 4759 4571 3821 7852 5166 4274 2746 6391 9320 0305
  9921 8174 1359 6629 0435 7290 0334 2952 6059 5630 7381 3232 8627 9434 9076
  3233 8298 8075 3195 2510 1901 1573 8341 8793 0702 1540 8914 9934 8841 6750
  9244 7614 6066 8082 2648 0016 8477 4118 5374 2345 4424 3710 7539 0777 4499
  2069 5517 0276 1838 6062 6133 1384 5830 0075 2044 9338 2656 0297 6067 3711
...  
  4995 8862 3428 1899 7077 3327 6171 7839 2803 4946 5014 3455 8897 0719 4258
  6396 7727 5471 0962 9537 4152 1115 1368 3506 2752 6023 2648 4728 7039 2076
  4310 0595 8411 6612 0545 2970 3023 6472 5492 9666 9381 1513 7322 7536 4509
  8889 0313 6020 5724 8176 5851 1806 3036 4428 1231 4965 5070 4751 0254 4650
  1172 7211 5551 9486 6850 8003 6853 2281 8315 2196 0037 3562 5279 4495 1582
  8418 8294 7876 1085 2639 8139

Digit counts

The frequency counts have been output because they have the remarkable
property of being very much closer to 200 that we should expect. There is a
simple statistical test (the $\chi^2$ test), covered in the next section, that shows just how unlikely these counts are assuming each digit is equally likely to be any digit in the range 0 to 9 and is independent of the other digits in the series.

4.30 The $\chi^2$ test

Feel free to skip this section if the formula below looks too frightening.

The program above showed us that, for $e$, the counts of each digit in the 2000 digits after the decimal point are 196, 190, 207, 202, 201, 197, 204, 198, 202 and 203. Since there are 2000 digits in all we would expect each to occur about 200 times, but, of course, we would also expect some random deviation from this average. Statisticians have devised a test (the $\chi^2$ test) that allows us to see if our collection of counts is reasonable. The method is as follows. First we calculate the quantity $\chi^2$ defined as follow.

$$\chi^2 = \sum_{i=1}^{k} \frac{(x_i - \mu_i)^2}{\mu_i}$$

where $k$ is the number of counts, $x_i$ is the $i^{th}$ count and $\mu_i$ is the expected value for $x_i$ which in our case is always 200. Putting our counts into the formula we obtain

$$\chi^2 = \frac{(196-200)^2}{200} + \frac{(190-200)^2}{200} + \frac{(207-200)^2}{200} + \frac{(202-200)^2}{200} + \frac{(201-200)^2}{200} +$$
$$+ \frac{(197-200)^2}{200} + \frac{(204-200)^2}{200} + \frac{(198-200)^2}{200} + \frac{(202-200)^2}{200} + \frac{(203-200)^2}{200} +$$
$$= \frac{16+100+49+4+4+4+9+16+4+4+9}{200}$$
$$= \frac{212}{200}$$
$$= 1.06$$

We had 10 counts but since they add up to 2000 the last count depends on the first 9, so for our collection the so called number of degrees of freedom is 9. We can lookup our value of $\chi^2$ in the table for 9 degrees of freedom to find the probability that $\chi^2$ would be greater than 1.06, assuming the digits are random and independent of one another. If you search the web using terms chi squared distribution calculator, you will find several web pages that will calculate the probability that $\chi^2$ should be greater than 1.06 for 9 degrees of freedom. The answer turns out to be 0.9993, so the chance that $\chi^2$ is 1.06 or smaller is less than one in a thousand.
4.31 \( e^x \)

The previous section defined \( e \) as the sum of a beautiful series whose \( n^{th} \) was \( \frac{1}{n!} \). Just for fun let us see what happens when we multiply this series by itself. Clearly the result should be a series representing \( e^2 \). So we have to simplify

\[
(1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \ldots) \times (1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \ldots)
\]

We can multiply each element of the left hand term by each element of the right hand term in a systematic way as follows

\[
\begin{align*}
1 \times 1 &= 1 = 1 \\
\frac{1}{1!} \times 1 + 1 \times \frac{1}{1!} &= \frac{1+1}{1!} = \frac{2}{1!} \\
\frac{1}{2!} \times 1 + \frac{1}{1!} \times \frac{1}{1!} + 1 \times \frac{1}{2!} &= \frac{1+2+1}{2!} = \frac{2^2}{2!} \\
\frac{1}{3!} \times 1 + \frac{1}{2!} \times \frac{1}{1!} + \frac{1}{1!} \times \frac{1}{2!} + 1 \times \frac{1}{3!} &= \frac{1+3+3+1}{3!} = \frac{2^3}{3!}
\end{align*}
\]

This shows that

\[
e^2 = 1 + \frac{2}{1!} + \frac{2^2}{2!} + \frac{2^3}{3!} + \ldots
\]

Seeing this equation leads us to thinking that

\[
e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots
\]

might be true. After all, it is certainly true when \( x \) is 0, 1 or 2. We can increase our believe that it is true by considering the product of the series for \( e^x \) and \( e^y \) to see if it yields the series for \( e^{x+y} \). We can do this by multiplying each element of the left hand term by each element of the right hand term in a systematic way as follows

\[
\begin{align*}
1 \times 1 &= 1 = 1 \\
\frac{x}{1!} \times 1 + 1 \times \frac{y}{1!} &= \frac{x+y}{1!} = \frac{(x+y)}{1!} \\
\frac{x^2}{2!} \times 1 + \frac{x}{1!} \times \frac{y}{1!} + 1 \times \frac{y}{2!} &= \frac{x^2+2xy+y^2}{2!} = \frac{(x+y)^2}{2!} \\
\frac{x^3}{3!} \times 1 + \frac{x^2}{2!} \times \frac{y}{1!} + \frac{x}{1!} \times \frac{y^2}{2!} + 1 \times \frac{y^3}{3!} &= \frac{x^3+3x^2y+3xy^2+y^3}{3!} = \frac{(x+y)^3}{3!}
\end{align*}
\]

This shows that
4.32. THE EXTRAORDINARY NUMBER $e^\pi\sqrt{163}$

\[ e^x \times e^y = 1 + \frac{(x+y)}{1!} + \frac{(x+y)^2}{2!} + \frac{(x+y)^3}{3!} + \ldots \]

which correctly represents the series for $e^{x+y}$, as expected.

So far we have assumed that $x$ and $y$ are integers, but the algebra we have just used works just as well when $x$ and $y$ are not whole numbers. Consider, for example, $e^{\frac{1}{2}}$. This clearly represents $\sqrt{e}$ since

\[ e^{\frac{1}{2}} \times e^{\frac{1}{2}} = e^{\frac{1}{2} + \frac{1}{2}} = e \]

Similarly, $e^{\frac{1}{q}}$ is the $q^{th}$ root of $e$. We can safely assume that our series works for any $x$ of the form $\frac{p}{q}$ where $p$ and $q$ are whole numbers. This leads us to believe the formula is correct even when $x$ cannot be represented as the ratio of two whole numbers. Examples of such numbers are $\sqrt{2}$, $\pi$ and even $e$ itself.

4.32 The extraordinary number $e^\pi\sqrt{163}$

This number is peculiar since it has 18 digits to the left of the decimal point, but a sequence of 12 nines to the right of the decimal point. The following program demonstrates this by computing its value to sufficient precision. The program is called epr163.b and starts as follows.

GET "libhdr"

MANIFEST
{ upb = 12
  upb1 = upb+1
}

LET start() = VALOF
{ LET pi = VEC upb
  AND root163 = VEC upb
  AND x = VEC upb
  AND ex = VEC upb
  LET exponent = 0

  numfromstr(pi, upb, "3.14159265358979323846264338327950*
  *288419716939937510582097494459230")
  writef("*nPi is*n")
  print(pi, 0)

  // Calculate root 163
sqrt163(root163)
writef("*nRoot 163 is*n")
print(root163, 0)

mult(x, pi, root163)
writef("*nPi times Root 163 is*n")
print(x, 0)

// Divide x by 2**10 (=1024) to make the computation
// e to the power x converge much more rapidly.
divbyk(x, 1024)
exp(ex, x)
// Now square the result 10 times.
FOR i = 1 TO 10 DO
{ exponent := 2*exponent
  mult(ex, ex, ex)
  IF ex!0>10000 DO
  { divbyk(ex, 10000)
    exponent := exponent + 1
  }
}
// Output the result
writef("*ne to the Pi root 163 is*n")
print(ex, exponent)
RESULTIS 0

A high precision number is represented by vector whose elements each contain four decimal digits. It is best to think of them as digits of radix 10000. The zeroth element is the integer part and the other elements contain the fractional digits. The upper bound of the vector is upb, set to 12, to allow a precision of over 40 decimal digits which is sufficient for our purposes. Four such vectors pi, root163, x, ex are declared to represent \( \pi \), \( \sqrt{163} \), \( \pi \times \sqrt{163} \) and \( e^{\pi \times \sqrt{163}} \), respectively. The function numfromstr is used to initialise pi from a string holding the digits of \( \pi \). The call sqrt163(root163) places a representation of \( \sqrt{163} \) in root163. The product of pi and root163 is placed in x using mult. Since x is about 40, the convergence of the series for \( e^x \) would be very slow, so x is reduced in size by dividing it by 1024 (= \( 2^{\text{10}} \)) before summing the series for \( e^x \), placing the result in ex by the call exp(ex, x). The result in ex is then squared 10 times to give a representation of \( e^{\pi \times \sqrt{163}} \). The only problem is that this value is outside the range of values our high precision numbers can hold. This is solved by maintaining an exponent value in exponent which specified that the number in ex should be multiplied by 10000^{exponent}. Each time ex is squared, exponent is doubled, and
4.32. THE EXTRAORDINARY NUMBER $E^{\pi \sqrt{163}}$

if $ex$ has become too large it is divided by 10000 and exponent incremented by one.

The additional functions used by this program are as follows.

```
AND numfromstr(v, upb, s) BE
{ LET p, k, val = 0, 0, k
  FOR i = 1 TO s%0 DO
    LET ch = s%i
    IF '0'<=ch<='9' DO val, k := 10*val + ch - '0', k+1
    IF ch='.' | k=4 DO
      IF p<=upb DO v!p := val
      p, k, val := p+1, 0, 0
    }  
  UNTIL k=4 DO val, k := 10*val, k+1
  IF p<=upb DO v!p := val
  // Pad on the right with zeroes
  UNTIL p>=upb DO { p := p+1; v!p := 0 } 
}
```

This take a character string in s and converts it into our high precision representation using the vector v whose upper bound is upb.

```
AND sqrt163(x) BE
{ // This is a simple but inefficient function to
  // calculate the square root of 163.
  LET w = VEC upb
  AND eps = VEC upb
  AND n163 = VEC upb
  numfromstr(x, upb, "13.") // Initial guess
  numfromstr(n163, upb, "163.")

  { mult(w, x, x)
    TEST w!0>=163 THEN { sub(eps, w, n163)
      divbyk(eps, 24)
      sub(x, x, eps)
    }
    ELSE { sub(eps, n163, w)
      divbyk(eps, 24)
      add(x, x, eps)
    }

    //print(x, 0) 
}
```
As the comment says this is a simple function to set \( x \) to a high precision representation of \( \sqrt{163} \). There was no need to use the much faster Newton-Raphson method.

\[ \text{AND mult}(x, y, z) \text{ BE} \]
\[ \{ \text{ LET res = VEC upb1} \]
\[ \quad \text{numfromstr(res, upb1, "0." )} \]
\[ \quad \text{// Round by adding a half to the last digit position.} \]
\[ \quad \text{res!upb1 := 5000} \]
\[ \quad \text{FOR i = 0 TO upb IF y!i FOR j = 0 TO upb1-i DO} \]
\[ \quad \quad \{ \text{ LET p = i + j // p is in range 0 to upb1} \]
\[ \quad \quad \quad \text{LET carry = y!i * z!j} \]
\[ \quad \quad \quad \text{WHILE carry DO} \]
\[ \quad \quad \quad \quad \{ \text{ LET w = res!p + carry} \]
\[ \quad \quad \quad \quad \quad \text{IF p=0 DO \{ res!0 := w; BREAK \}} \]
\[ \quad \quad \quad \quad \quad \text{res!p, carry := w MOD 10000, w/10000} \]
\[ \quad \quad \quad \quad \quad p := p-1 \}
\[ \quad \}
\[ \text{FOR i = 0 TO upb DO x!i := res!i} \]
\[ \} \]

This function multiplies the high precision numbers in \( y \) and \( z \) placing the rounded result in \( x \). It uses a temporary vector \( res \) that includes an extra digit to allow for rounding. Every pair of digits that can contribute to the result are multiplied together and added to the appropriate position in \( res \), dealing with carries as they arise.

\[ \text{AND exp(ex, x) \text{ BE}} \]
\[ \{ \text{ // This calculates e to the power x by summing the series} \]
\[ \quad \text{// whose nth term is x**n/n!} \]
\[ \quad \text{LET n = 0} \]
\[ \quad \text{LET term = VEC upb} \]
\[ \quad \text{numfromstr(term, upb, "1.")} \]
\[ \quad \text{numfromstr(ex, upb, "0.")} \]
\[ \text{UNTIL iszero(term) DO} \]
\[ \quad \text{add(ex, ex, term)} \]
\[ \quad \text{n := n+1} \]
\[ \quad \text{mult(term, term, x)} \]
4.32. THE EXTRAORDINARY NUMBER $e^{\pi \sqrt{163}}$

\[ e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \ldots \]

This computes $e^x$ using the series

The result is accumulated in $ex$ and $term$ holds the next term to be added. The summation stops when $term$ holds zero.

AND add(x, y, z) BE
{ LET c = 0
  FOR i = upb TO 0 BY -1 DO
    { LET d = c + y!i + z!i
      x!i := d MOD 10000
      c := d / 10000
    }
  }
}

This function adds the high precision numbers in $y$ and $z$ placing the result in $x$.

AND sub(x, y, z) BE
{ LET borrow = 0
  FOR i = upb TO 1 BY -1 DO
    { LET d = y!i - borrow - z!i
      borrow := 0
      UNTIL d>=0 DO borrow, d := borrow+1, d+10000
      x!i := d
    }
  }
x!0 := y!0 - borrow - z!0
}

This function subtracts the high precision number in $z$ from $y$ placing the result in $x$.

AND divbyk(v, k) BE
{ LET c = 0
  FOR i = 0 TO upb DO
    { LET d = c*10000 + v!i
      v!i := d / k
      c := d MOD k
    }
  }
}
This divides the high precision number in $v$ by $k$ which must be in the range 1 to 10000.

$$\text{AND iszero}(v) = \text{VALOF}$$

{ FOR $i = \text{upb} \text{ TO } 0 \text{ BY } -1$ IF $v!i$ RESULTIS FALSE
    RESULTIS TRUE
}

This returns TRUE is the high precision number in $v$ is zero.

$$\text{AND print}(v, \text{exponent}) \text{ BE}$$

{ writef("%i4", v!0)
  FOR $i = 1 \text{ TO } \text{upb} \text{ DO}$
  { wrch(exponent=0 \text{ \rightarrow} \ '.', '*s')
    exponent := exponent - 1
    IF $i \text{ MOD } 15 = 0 \text{ DO newline()}
    wrpn(v!i, 4)
  }
  newline()
}

$$\text{AND wrpn}(n, d) \text{ BE}$$

{ IF $d>1 \text{ DO wrpn}(n/10, d-1)$
  wrch(n \text{ MOD } 10 +'0')
}

These two functions combine to output a high precision number with a given exponent.

When this program runs, its output is as follows.

Pi is

3.1415 9265 3589 7932 3846 2643 3832 7950 2884 1971 6939 9375

Root 163 is

12.7671 4533 4803 7046 6171 0952 0097 8089 2347 3823 6377 9407

Pi times Root 163 is

40.1091 6999 1132 5197 5535 0083 6229 0414 0053 9005 3481 5142

e to the Pi root 163 is

26 2537 4126 4076 8743.9999 9999 9999 2500 7259 7198 1820 2936
4.33 Digits of $\pi$

This section is another illustration of the use of modulo arithmetic. It is entirely optional and can be skipped.

The ratio of the circumference of a circle to its diameter is a very important constant called $\pi$, and it has a value of about 3.14159, and some people like to use the approximations $\frac{22}{7}$ or $\frac{355}{113}$. In the mid 1930s, $\pi$ was known to about 700 decimal places but now, with the aid of computers and staggeringly cunning methods it can be calculated to billions (and even trillions) of decimal places. For more information do a web search on: digits of pi.

One intriguing method was discovered by David Bailey, Peter Borwein and Simon Plouffe and appears in section 10.7 of “Number Theory, A Programmer’s Guide” by Mark Herkommer. It is based on the totally remarkable formula:

$$\pi = \sum_{i=0}^{\infty} \left( \frac{4}{8i + 1} - \frac{2}{8i + 4} - \frac{1}{8i + 5} - \frac{1}{8i + 6} \right) \times \left( \frac{1}{16} \right)^i$$

The beauty of this formula is that it can be used to calculate the $n^{th}$ hexadecimal digit of $\pi$ using modulo arithmetic with the big advantage that the other digits are not computed. So how do we do it?

We multiply the right hand side by $16^n$ and split it into the first $n$ terms and the rest, namely

$$\sum_{i=0}^{n-1} \left( \frac{4 \times 16^{n-i}}{8i + 1} - \frac{2 \times 16^{n-i}}{8i + 4} - \frac{16^{n-i}}{8i + 5} - \frac{16^{n-i}}{8i + 6} \right)$$

and

$$\sum_{i=n}^{\infty} \left( \frac{4}{8i + 1} - \frac{2}{8i + 4} - \frac{1}{8i + 5} - \frac{1}{8i + 6} \right) \times \left( \frac{1}{16} \right)^{i-n}$$

If we add these two sums together, we obtain a huge number, and if we represent it using hexadecimal digits we find that the first digit to the right of the decimal point is the $n^{th}$ hex digit of $\pi$. If we are only interested in this digit all the digits to the left of the decimal point can be discarded and only a few to the right of the decimal point need to be retained during the calculation. Let us consider the first term in the first sum. The contribution this term makes to the result is

$$\sum_{i=0}^{n-1} 4 \left( \frac{16^{n-i}}{8i + 1} \right)$$

But we are only interested in the fractional part, so the following sum will do just as well.
\[ \sum_{i=0}^{n-1} 4^{16^n-i \mod (8i+1)} \]

Computing \(16^n-i \mod (8i+1)\) throws away all integer multiples of \((8i+1)\) leaving only the remainder, which is positive but less than \(8i+1\), so when this is divided by \(8i+1\) yields a value between 0 and 1. This trick is similar to calculating the fractional part of \(123/10\) as follow:

\[
\frac{123 \mod 10}{10} = \frac{3}{10} = 0.3
\]

A program to output the digits of \(\pi\) in hexadecimal and decimal is in `bcplprogs/raspi/pidigs.b`. It starts as follows:

```
GET "libhdr"

MANIFEST {
    // Define the scaled arithmetic parameters
    frclen = 28 // Number of binary digits after the decimal point
    // 28 allows numbers in the range -8.0 <= x < 8.0
    One = 1<<frclen // eg #x10000000
    Two = 2*One // eg #x20000000
    Four = 4*One // eg #x40000000
    fracmask = One - 1 // eg #x0FFFFFFF

    upb = 1000
}

LET start() = VALOF
{ LET hexdig = getvec(upb)
    writef("*nPi in hex*n")
    writef("*n 3.").
    hexdig!0 := 3
    FOR n = 1 TO upb DO {
        LET dig = pihexdig(n-1)
        IF n MOD 50 = 1 DO writef("*n%5i: ", n)
        writef("%x1", pihexdig(n)); deplete(cos)
    }
    newline()

    writef("*nPi in decimal*n")
    writef("*n 3.")
```
FOR i = 1 TO upb DO
{ IF i MOD 50 = 1 DO writef("*n%5i: ", i)
   hexdig!0 := 0 // Remove the integer part then
   mulby10(hexdig, upb) // multiply the fraction by 10 to obtain
   writef("%n", hexdig!0) // the next decimal digit in hexdig!0
   deplete(cos)
}
newline()
freevec(hexdig)
RESULTIS 0
}

The constant fraclen (=28) specifies the number of binary digits after the decimal point of the scaled numbers we will be using. This leaves 4 bits (or one hexadecimal digit) to the left of the decimal point. We will be using signed arithmetic, so this allows us to represent numbers greater than or equal to \(-8.000\) and less than \(8.000\) which is sufficient for our purposes. The constants One, Two and Four represent the numbers 1, 2 and 4 in this scaled representation, and fracmask is a bit pattern that will extract just the fractional bits of our numbers.

The main function start outputs the hexadecimal digits of \(\pi\) up to position 1000, placing 50 digits per line. Each digit is calculated by calls of pihexdig. These digit are saved in the vector hexdig to allow them to be converted to decimal. The conversion to decimal is simple. It just requires setting the integer part (held in hexdig!0) to zero before multiplying the fraction in hex by decimal 10 giving the next decimal digit in hexdig!0. The calculation is outlined below.

3.14159265 \Rightarrow 0.14159265 \times 10 \Rightarrow 1.4159265
1.4159265 \Rightarrow 0.4159265 \times 10 \Rightarrow 4.159265
4.159265 \Rightarrow 0.159265 \times 10 \Rightarrow 1.59265
1.59265 \Rightarrow 0.59265 \times 10 \Rightarrow 5.9265
5.9265 \Rightarrow 0.9265 \times 10 \Rightarrow 9.265

The multiplication by 10 is done by mulby10 defined as follows.

AND mulby10(v, upb) BE
{ // v contains one hex digit per element with the
   // decimal point between v!0 and v!1
   LET carry = 0
   FOR i = upb TO 0 BY -1 DO
   { LET d = v!i*10 + carry
     v!i, carry := d MOD 16, d/16
   }
}
The library function \texttt{muldiv} take three signed numbers and returns the mathematically correct result of dividing the third argument into the product of the first two. Thus \texttt{muldiv(x,y,z)=}(x*y)/z, but \(x\times y\) is computed as a double length quantity. The function \texttt{powmod(x,n,m)}, defined later, computes \(x^n \bmod m\) with reasonably efficiently. Note that

\[
muldiv(Four, powmod(16, n-i, 8*i+1), 8*i+1)
\]

will return the value of

\[
4 \left( \frac{16^{n-i} \bmod (8i+1)}{8i+1} \right)
\]

as a number using our scaled representation. The definition of \texttt{pihexdig} is as follows.

\texttt{AND pihexdig(n) = VALOF}

\{
  // By convention, the first hex digit after the decimal point
  // is at position n=0
  LET s = 0 // A scaled number with fraclen binary digits
  // after the decimal point.
  LET t = One

  FOR i = 0 TO n-1 DO
  { LET a = muldiv(Four, powmod(16, n-i, 8*i+1), 8*i+1)
    LET b = muldiv(Two, powmod(16, n-i, 8*i+4), 8*i+4)
    LET c = muldiv(One, powmod(16, n-i, 8*i+5), 8*i+5)
    LET d = muldiv(One, powmod(16, n-i, 8*i+6), 8*i+6)
    s := s + a - b - c - d & fracmask
  }

  // Now add the remaining terms until they are too small
  // to matter.
  { LET i = n
    WHILE t DO
    { LET a = 4 * t / (8*i+1)
      LET b = 2 * t / (8*i+4)
      LET c = t / (8*i+5)
      LET d = t / (8*i+6)
      s := s + a - b - c - d & fracmask
      i, t := i+1, t/16
    }
  }
\}
4.33. DIGITS OF $\pi$

} 

RESULTIS (s>>(fraclen-4)) & #xF // Extract the required digit 

} 

To complete the program, the definition of powmod is as on Page 65, namely 

AND powmod(x, n, m) = VALOF 
{ LET res = 1 
 LET p = x MOD m 
 WHILE n DO 
{ UNLESS (n & 1)=0 DO res := (res * p) MOD m 
 n := n>>1 
 p := (p*p) MOD m // DANGER: p*p must not overflow 
} 
RESULTIS res 
} 

The actual program in raspi/pidigs.b contains some optional tracing code as a debugging aid. The values of $a$, $b$, $c$, $d$, and $s$ can be output in decimal and hexadecimal as they are computed using the function tr, as in tr("a", a). The definition of tr is as follows.

AND tr(str, x) BE 
{ // Output scaled number x in decimal and hex 
 LET d = muldiv( 1_000_000, x, One) 
 LET h = muldiv(#x10000000, x, One) // Just in case fraclen is not 28 
 writef("%s = %9.6d %8x*n", str, d, h) 
} 

When pidigs runs it generates the following output.

0.000> pidigs 

Pi in hex 

3. 
1: 243F6A8885A308D313198A2E037073444A4093822299F31D008 
51: 2EFA98EC4E6C89452821E638D01377BE5466CF34E90C6CC0AC 
101: 29B7C97C50DD3F84D5B5B54709179216D5D98979FB1BD1310B 
151: A698DF85AC2FFD72BD01ADF87B8E1AFED6A267E96B7C9045 
201: F12C7F9924A19947B3916CF70801F2E2858EFC16636920D871 
251: 574E69A458FEA3F493D7E0D95748F728EB658718BCD588215 
301: 4AEE7B54A41DC25A5B959C30D5392AF26013C5D1B023286085 
351: F0CA417918B8DB38EF8E79DCB0603A180E6C9E0E8BB01E8A3E
By changing to bounds of the FOR loop in start and disabling the decimal conversion, you can discover that the hexadecimal digit at position one million is 6, which I think is remarkable for such a small program. But beware, 28 fractional bits does not have sufficient precision to guarantee all digits from position zero to one million are correct. Try reducing fraclen to see where errors begin to creep in. For instance, if fraclen=22 the first error is at position 1269, and 25 gives an
error at 3708. 28 gives correct digits at least up to position 5000. Unfortunately, if you want more than 28 bits the program will need substantial modification.

4.34 More commands

The programs given so far have included examples of most of the constructs available in BCPL. This section just describes a few of them in more detail.

We should now be familiar with the IF and UNLESS statements that allow the conditional execution of commands based on the values returned by expressions. The convention is that a value of zero represents false and any non zero value represents true. For convenience, the keywords FALSE and TRUE have values zero and -1. Note that the bit pattern operators &, | and ~ work well with this representation of truth values. For instance, (TRUE & FALSE) = FALSE and (FALSE | ~FALSE) = TRUE. However, there is one subtlety which is as follows.

When an expression is used in a conditional statement controlling the flow of execution, the operators &, | and ~ are evaluated slightly differently. For instance, in the command IF x=0 & y>3 RESULTIS 13, if the value of x is non zero the condition y>3 will not be evaluated since it is already known that the RESULTIS statement will not be executed since it is already known that the RESULTIS statement will not be executed. The expression x=0 & y>3 in this example is being evaluated in what is called Boolean context. Whereas in the assignment sw := x=0 & y>3 both x=0 and y>3 are evaluated before being anded together. The only places where expressions are evaluated in a Boolean contexts are those used in IF, UNLESS, TEST, WHILE, UNTIL, REPEATWHILE, REPEATUNTIL, and the expression to the left of -> in a conditional expression. It is important to know when an expression is being evaluated in a Boolean context since, for instance, the following two statements are not equivalent.

\[
\begin{align*}
\text{IF } x & \& 7 \text{ RESULTIS 12} \\
\text{IF } (x & 7) = 0 \text{ RESULTIS 12}
\end{align*}
\]

The first will execute the RESULTIS statement whenever x is non zero, but the second will only do so if the least significant three bits of x are not all zero.

The IF and UNLESS commands allow for the conditional execution of a command. If you wish to conditionally execute one of two commands you should use the TEST commands, as in

\[
\begin{align*}
\text{TEST tracing} \\
\text{THEN writef("*nSignal tracing now on*n")} \\
\text{ELSE writef("*nSignal tracing turned off*n")}
\end{align*}
\]

It is sometimes necessary to select one of many alternative command based on the value of an expression. This is often done using the SWITCHON command as in:

\[
\begin{align*}
\text{SWITCHON } x \\
\text{WHEN y=1 THEN writef("*nHello, World!*n")} \\
\text{ELSE writef("*nGoodbye, World!*n")}
\end{align*}
\]
CHAPTER 4. THE BCPL CINTCODE SYSTEM

SWITCHON op INTO
{ DEFAULT: writeln("Unknown operator \%n\*n", op)
    abort(999)
   ENDCASE
   CASE Pos: ENDCASE
   CASE Neg: a := - a; ENDCASE
   CASE Add: a := b + a; ENDCASE
   CASE Sub: a := b - a; ENDCASE
   CASE Mul: a := b * a; ENDCASE
   CASE Div: a := b / a; ENDCASE
   CASE Mod: a := b MOD a; ENDCASE
}

Here the value of op is inspected and compared with Pos, Neg, Add, Sub, Mul, Div and Mod, all of which must have been declared as MANIFEST constant. If op is not equal to any of them control passed to the default label, otherwise execution continues at the appropriate CASE label. The ENDCASE statement cause a jump to just after the SWITCHON command. Although MANIFEST constants are often used in CASE label, numerical and character constants are frequently used.

In addition to ENDCASE, there are several other special jump commands. BREAK causes a jump out of the current repetitive command. The repetitive commands are those with keywords WHILE, UNTIL, REPEATWHILE, REPEATUNTIL, REPEAT and FOR. LOOP causes a jump to end of the body of a repetitive command normally to where the repetition condition is re-evaluated. For a REPEAT command, it jumps to the start of the body and for a FOR command it jumps to where the control variable is incremented. The other jump commands are RESULTIS which jumps to the end of the current VALOF expression carrying with it the result, and, finally, RETURN causes a return from the current function. Careful use of these commands almost eliminates the need to ever use the GOTO command.

4.35 The VSPL Compiler

As a final example we will look at a somewhat more substantial program.

BCPL was originally written to help with the implementation of programming language compilers, and its own compiler is a good example. It is, however, too long and complicated to be used as an introduction to compiler writing. A much simpler language called VSPL (Very Simple Programming Language) was designed as an educational tool showing how a compiler can be written in several languages using different programming styles. If you are interested, look at the VSPL distribution available from my home page. The standard BCPL distribution includes the BCPL version of the VSPL compiler in com/vspl.b together with two example programs primes.vs and demo.vs in the BCPL root directory. When printed vspl.b is only 21 pages long, but does contain a lexical
4.36. SUMMARY OF BCPL

analyser, a parser, a translation phase and an interpreter to execute the compiled code. It also contains debugging aids to help you understand how the compiler works.

To explore the VSPL system, try typing the following commands.

cd $BCPLROOT  -- Enter the BCPLROOT directory
cintsys  -- Start the BCPL system
c bc vspl  -- Compile the VSPL compiler
type primes.vs  -- Look at a typical VSPL program
vspl primes.vs  -- Compile and run it
type demo.vs  -- Look at a tiny demo program
vspl -l demo.vs  -- Look at the result of lexical analysis
vspl -p demo.vs  -- Look at the parse tree
vspl -c demo.vs  -- Look at the compiled code
vspl -t demo.vs  -- Trace the execution of the compiled code

For more information look at the VSPL distribution available via my home page.

4.36 Summary of BCPL

This section gives a brief summary of BCPL. For a full description of the language look at the BCPL Manual (bcplman.pdf) given in my home page.

In the syntactic forms given below

\[ E \] denotes an expression,
\[ K \] denotes a constant expression,
\[ C \] denotes a command,
\[ D \] denotes a definition,
\[ A \] denotes a function argument list,
\[ N \] denotes a variable name,

4.36.1 Comments and GET

Text between // and the end of the line is ignored. The symbols /* and */ are called comment brackets. These brackets and the text enclosed between them are ignored. Such comments may be nested.

A GET directive of the form GET "filename" as in GET "libhdr" is replaced by the contents of the specified file. GET first searches the current directory and then the directories specified by the BCPLHDRS environment variable. If the file name does not end with .h or .b, .h is appended.
4.36.2 Sections

A section is a sequence of declarations optionally preceded by a SECTION directive of the form SECTION "name". Several sections can occur in one file separated by dots.

4.36.3 Declarations

LET D AND ... AND D

AND joins simultaneous definitions together. All the variables defined have a scope starting at the word LET.

MANIFEST { N = K ; ... ; N = K }

The “= K”s are optional. When omitted the next available integer is used.

STATIC { N = K ; ... ; N = K }

The “= K”s are optional. When omitted the corresponding variables have undefined initial values.

GLOBAL { N : K ; ... ; N : K }

The “: K”s are optional. When omitted the next available integer is used.

4.36.4 Definitions

Definitions are used in declarations after the word LET or AND. They are as follows.

N , ... , N = E , ... , E

This is a simultaneous definition defining a list of local variables with specified initial values. They are allocated consecutive locations in memory.

N = VEC K

This is a local vector definition. It defines a local variable N with an initial value that points to the zeroth element of a local vector whose upper bound is the constant K.

N ( N , ... , N ) = E

This defines a function that returns a result specified by the expression E. It has zero or more arguments.

N ( N , ... , N ) BE C

This defines a function just like the one above but has no specified result.

4.36.5 Expressions

N

Eg: abc v1 a s_err

These are used to name functions, variables and constants.
4.36. SUMMARY OF BCPL

numb

Eg: 1234 #x7F_0001 #377 #b_0111_1111_0000
These yield specified constant values.

? 
This yields an undefined value.

TRUE FALSE
These represent the two truth values -1 and 0, respectively.

char

Eg: 'A' '*n'
These character constants are encoded as numbers in the range 0 to 255.

string

Eg: "abc" "Hello*n"
A string is represented by a pointer to where the characters of the string are packed. The individual characters are encoded as 8-bit bytes and can be accessed using the percent operator %. The zeroth character of a string holds its upper bound.

TABLE K, ..., K
This yields an initialised static vector. The elements of the vector are initialised to the given compile time constants.

VALOF C
This introduces a new scope for locals and defines the context for RESULTIS commands within C.

( E )
Parentheses are used to override the normal precedence of the expression operators.

E ( E, ..., E )
This is a function call.

® E
This returns the address of E which must be either a variable name or of the form E!E or !E.

E ! E ! E
This is the subscription operator. The left operand is a pointer to the zeroth element of a vector and the right hand operand is an integer subscript. The form !E is equivalent to E!0.

E % E
This is the byte subscription operator. The left operand is a pointer to the zeroth element of a byte vector and the right hand operand is an integer subscript.

+ E - E ABS E
These are monadic operators for plus, minus and absolute value, respectively.

E * E E / E E MOD E
These are dyadic operators for multiplication, division, remainder after division, respectively.
These are dyadic operators for addition and subtraction, respectively.

\[ E + E \quad E - E \]

These are dyadic operators for addition and subtraction, respectively.

\[ E \operatorname{relop} E \operatorname{relop} \ldots \operatorname{relop} E \]
where \( \operatorname{relop} \) is any of \( =, \sim=, <, \leq, >, \geq \). It returns \( \text{TRUE} \) only if all the individual relations are satisfied.

\[ E \ll E \quad E \gg E \]
These are logical left and right shift operators, respectively.

\( \sim E \)
This returns the bitwise complement of \( E \).

\[ E \& E \]
This returns the bitwise AND of its operands.

\[ E | E \]
This returns the bitwise OR of its operands.

\[ E \operatorname{XOR} E \]
This returns the bitwise exclusive OR of its operands.

\[ E \rightarrow E, E \]
This is the conditional expression construct.

### 4.36.6 Commands

\[ E, \ldots, E := E, \ldots, E \]
This is the simultaneous assignment operator. The order in which the expressions are evaluated is undefined.

TEST \( E \) THEN \( C \) ELSE \( C \)

IF \( E \) DO \( C \)

UNLESS \( E \) DO \( C \)
These are the conditional commands. They are less binding than assignment.

SWITCHON \( E \) INTO \( C \)

DEFAULT:

CASE \( K \):

ENDCASE
The \( \text{DEFAULT} \) label and \( \text{CASE} \) labels identify positions within the body of a SWITCHON command. The effect of a SWITCHON command is to evaluate \( E \) and then transfer control to the matching \( \text{CASE} \) label. If no \( \text{CASE} \) label matches control is passed to the \( \text{DEFAULT} \) label, but if there is no \( \text{DEFAULT} \) label control exits from the SWITCHON command. ENDCASE causes an exit from the SWITCHON command. It normally occurs at the end of the code for each case.

WHILE \( E \) DO \( C \)
UNTIL \( E \) DO \( C \)
4.36. SUMMARY OF BCPL

C REPEATWHILE E
C REPEATUNTIL E
C REPEAT
FOR $N = E$ TO $E$ BY $K$ DO $C$
FOR $N = E$ TO $E$ DO $C$

These are the repetitive commands. The FOR command introduces a new scope for locals, and $N$ is a new variable within this scope.

RESULTIS $E$

This returns from current VALOF expression with the given value.

RETURN

Return from current function with an undefined value.

BREAK LOOP

Respectively, exit from, or loop in the current repetitive command.

$N$:

GOTO $E$:

The construct $N$: sets a label to this point in the program, and the GOTO command can be used to transfer to this point. However, the GOTO and the label must be in the same function.

$C ; \ldots ; C$

Evaluate the commands from left to right.

$\{C ; \ldots ; C\}$

This construct is called a compound command and is treated syntactically as a single command. It can, for instance, be the operand of an IF statement. A sequence of declaration is permitted immediately after the open section bracket ($\{\}$). This causes it to be called a block. The declared names have a scope limited to the block.

4.36.7 Constant expressions

These are used in MANIFEST, STATIC and GLOBAL declarations, in VEC definitions, and in the step length of FOR commands.

The syntax of constant expressions is the same as that of ordinary expressions except that only constructs that can be evaluated at compile time are permitted.

These are:

$N$, $\text{numb}$, $\text{?}$, TRUE, FALSE, $\text{char}$,
($K$),
$+ K$, $- K$, $\text{ABS } K$,
$K * K$, $K / K$, $K \mod K$
$K + K$, $K - K$,
$K \text{ relop } K \text{ relop } \ldots \text{ relop } K$,
\[ K \ll K, K \gg K, \]
\[ \sim K, \]
\[ K \& K, \]
\[ K \mid K, \]
\[ K \text{ XOR } K, \]
\[ K \rightarrow K, K \]
Young Persons Guide to BCPL Programming on the Raspberry Pi Part 2

by

Martin Richards
mr@cl.cam.ac.uk
http://www.cl.cam.ac.uk/~mr10/

Computer Laboratory
University of Cambridge
Revision date: Mon Aug 1 14:46:24 BST 2016
Chapter 5

Interactive Graphics in BCPL using SDL

5.1 Introduction

If your system does not already have the SDL libraries and header files installed, you should fetch them using commands such as the following.

```
sudo apt-get update
sudo apt-get install libsdl1.2-dev libSDL-image1.2-dev
sudo apt-get install libsdl-mixer1.2-dev libSDL-ttf2.0-dev
```

The `apt-get update` command stops some annoying error messages being generated by the two `install` commands.

As a test to see if they have been installed examine the directory `/usr/include/SDL`. It should contain several files relating to SDL.

Having installed the SDL libraries you should rebuild the BCPL system telling it to use the libraries. To do this type the following.

```
cd ~/distribution/BCPL/cintcode
make clean
make -f MakefileRaspiSDL
```

This should rebuild the BCPL system from its source incorporating and interface with SDL.

Although all the programs in this chapter can be controlled from the keyboard, you may find it useful to plug a USB joystick into your Raspberry Pi. I bought a Logitech Attack 3 Joystick which is cheap, well made and works well. It is shown below. Although it provides elevator, aileron and throttle control together with
11 buttons, it does not provide a convenient rudder control, so you might wish to buy a more expensive model.

To test whether you have installed the SDL graphics library correctly, try compiling and running the demonstration program `bcplprogs/raspi/engine.b` by typing the following commands.

```bash
cd ~/distribution/BCPL/bcplprogs/raspi
cintsys
c b engine
engine
```

This should create and display the following window for about 20 seconds.

![First SDL Demo](image)

The program starts as follow.

```bcpl
GET "libhdr"
GET "sdl.h"
GET "sdl.b" // Insert the library source code
.
GET "libhdr"
GET "sdl.h"
```

The first four lines consisting of three `GET` directives and a dot, cause a BCPL interface to the SDL library to be compiled as a separate section at the head of the program. The source is in `cintcode/g/sdl.b` and it uses a header file called `cintcode/g/sdl.h`. In due course you should look at these files to see what is
provided, but that can wait. The program goes on to declare some global variables that will be used to hold the various colours.

GLOBAL {
    col_black: ug
    col_blue
    col_green
    col_yellow
    col_red
    col_majenta
    col_cyan
    col_white
    col_darkgray
    col_darkblue
    col_darkgreen
    col_darkyellow
    col_darkred
    col_darkmajenta
    col_darkcyan
    col_gray
    col_lightgray
    col_lightblue
    col_lightgreen
    col_lightyellow
    col_lightred
    col_lightmajenta
    col_lightcyan
}

The rest of the program just contains the definition of the main program start, and is as follows.

LET start() = VALOF
{ initsdl()
    mkscreen("First SDL Demo", 600, 400)
    col_black := maprgb( 0, 0, 0)
    col_blue := maprgb( 0, 0, 255)
    col_green := maprgb( 0, 255, 0)
    col_yellow := maprgb( 0, 255, 255)
    col_red := maprgb(255, 0, 0)
    col_majenta := maprgb(255, 0, 255)
    col_cyan := maprgb(255, 255, 0)
    col_white := maprgb(255, 255, 255)
    col_darkgray := maprgb( 64, 64, 64)
CHAPTER 5. INTERACTIVE GRAPHICS IN BCPL USING SDL

col_darkblue := maprgb( 0, 0, 64)
col_darkgreen := maprgb( 0, 64, 0)
col_darkyellow := maprgb( 0, 64, 64)
col_darkred := maprgb(128, 0, 0)
col_darkmajenta := maprgb( 64, 0, 64)
col_darkcyan := maprgb( 64, 64, 0)
col_gray := maprgb(128, 128, 128)
col_lightblue := maprgb(128, 128, 255)
col_lightgreen := maprgb(128, 255, 128)
col_lightyellow := maprgb(128, 255, 255)
col_lightred := maprgb(255, 128, 128)
col_lightmajenta := maprgb(255, 128, 255)
col_lightcyan := maprgb(255, 255, 128)

fillscreen(col_darkgreen)

setcolour(col_cyan)
plotf(250, 30, "First Demo")

setcolour(col_red)  // Rails
moveto( 100, 80)
drawby( 400, 0)
drawby( 0, -10)
drawby(-400, 0)
drawby(0, 10)

setcolour(col_black)  // Wheels
drawfillcircle(250, 100, 25)
drawfillcircle(350, 100, 25)
setcolour(col_green)
drawfillcircle(250, 100, 20)
drawfillcircle(350, 100, 20)

setcolour(col_blue)  // Base
drawfillrect(200, 110, 400, 130)

setcolour(col_majenta)  // Boiler
drawfillrect(225, 135, 330, 170)

setcolour(col_darkred)  // Cab
drawfillroundrect(340, 135, 400, 210, 15)
setcolour(col_lightyellow)
drawfillroundrect(350, 170, 380, 200, 10)

setcolour(col_lightred)  // Funnel
The call `initsdl()` initialises the SDL system allowing the program to create a window, draw a picture in it, interact with the keyboard, mouse, and joystick, if any, and even generate sounds. The call of `mkscreen` creates a window that is 600 pixels wide and 400 pixels high. It is given the title First SDL Demo.

Then follows a sequence of calls to `maprgb` to create values representing colours in the pixel format used by the system. These calls can only be made after `mkwindow` has been called. There are several possible pixel formats and is more efficient to use the one that the system is currently using. It turns out that the pixel format on my laptop is different from the one used by the Raspberry Pi.

The next call `fillscreen(col_dkgreen)` fills the entire window with the specified colour. The call `setcolour(...)` selects the colour to use in subsequent drawing operations. The first of which is to draw the string First Demo starting 250 pixels from the left of the window and 30 pixels from the bottom. The convention often adopted in windowing systems is to measure the vertical displacement from the top, but I have adopted the convention that the vertical displacement increases as you move upwards as is typical when drawing graphs on graph paper. If my choice turns out to be too problematic, I will change it and all your pictures will suddenly be upside down.

Lines can be drawn in the selected colour by calls such as `moveto`, `drawto`, `moveby` and `drawby`, which each take a pair of arguments giving either the absolute or relative pixel locations. More complicated shapes can be drawn using functions such as `drawcircle(ox, oy, r)`, `drawfillcircle(ox, oy, r)`, `drawrect(x1, y1, x2, y2)`, `drawfillrect(x1, y1, x2, y2)`, `drawroundrect(x1, y1, x2, y2, r)` and `drawfillroundrect(x1, y1, x2, y2, r)`. In these calls `ox` and `oy` are the coordinates of the centre of the circle and `r` is its radius. If the function name includes `fill`, the edge and inside of the shape is filled with the selected colour, otherwise only the edge is drawn. Rectangles can have rounded corners with a radius in pixels given by `r`.

After drawing the picture it can be sent to the display hardware by the call `updatescreen()`. The call `sdldelay(20_000)` causes a real time delay of 20 sec-
onds so that the image can be viewed, and the final call `closesdl()` causes the graphics system to close down.

## 5.2 The dragon curve

This next demonstration draws the well known dragon curve. The idea is simple. To draw the curve from point $A$ to $B$, if the distance is less than a certain limit, the curve is just a straight line from $A$ to $B$, otherwise a detour is made travelling along two sides of a square whose diagonal is $AB$. If the sides of the square is still too long, detours are again taken, and so on. The detours alternate in direction, the first being to the right, the second being to the left and so on. Surprisingly this generates a rather beautiful picture. The following program generates a dragon curve containing 1024 short line segments with a short delay as each is drawn so you can see the picture being built up. The program is in the file `bcplprogs/raspi/dragon.b` and is as follows.

```bcpl
GET "libhdr"
GET "sdl.h"
GET "sdl.b"
.
GET "libhdr"
GET "sdl.h"

GLOBAL {
    col_blue: ug
    col_white
    col_lightcyan
}

LET start() = VALOF
{ initsdl()
    mkscreen("Dragon Curve", 600, 600)

    col_blue := maprgb( 0, 0, 255)
    col_white := maprgb(255, 255, 255)
    col_lightcyan := maprgb(255, 255, 64)

    fillscreen(col_blue)

    setcolour(col_lightcyan)
    plotf(240, 50, "The Dragon Curve")

    setcolour(col_white)
    moveto(260, 200)
```

dragon(1024, 6)

updatescreen()
sdldelay(20_000)
closesdl()
RESULT IS 0
}

AND gray(n) = n XOR n>>1

AND bits(w) = w=0 -> 0, 1 + bits(w & w-1)

AND dragon(n, size) BE FOR i = 0 TO n-1 DO
{ LET dir = bits(gray(i))
    SWITCHON dir & 3 INTO
    { CASE 0: drawby(size, 0); ENDCASE // Right
        CASE 1: drawby(0, size); ENDCASE // Up
        CASE 2: drawby(-size, 0); ENDCASE // Left
        CASE 3: drawby(0, -size); ENDCASE // Down
    }
    updatescreen() // Show the curve as it is drawn
    sdldelay(20)
}

When this program runs, it creates a window like the following.
The program uses a cunning trick to determine the direction the \( i^{th} \) line segment based on the number of one bits in the gray code representation of \( i \). The gray code corresponding to the binary number 0110 is shown as follows.

<table>
<thead>
<tr>
<th>number in binary</th>
<th>0 1 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>corresponding gray code</td>
<td>1 0 1</td>
</tr>
</tbody>
</table>

Notice that each digit of the gray code is computed by comparing adjacent digits of the number. The gray code digit is 0 if the adjacent digits are the same, otherwise it is a 1. This conversion is done by the function `gray` whose body is `n XOR (n>>1)`. The gray codes for the integers 000 to 111 are shown in the following table.
5.3. COLLATZ REVISITED

<table>
<thead>
<tr>
<th>n</th>
<th>n XOR (n&gt;&gt;1)</th>
<th>ones</th>
<th>direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>000</td>
<td>0</td>
<td>right</td>
</tr>
<tr>
<td>001</td>
<td>001</td>
<td>1</td>
<td>up</td>
</tr>
<tr>
<td>010</td>
<td>011</td>
<td>2</td>
<td>left</td>
</tr>
<tr>
<td>011</td>
<td>010</td>
<td>1</td>
<td>up</td>
</tr>
<tr>
<td>100</td>
<td>110</td>
<td>2</td>
<td>left</td>
</tr>
<tr>
<td>101</td>
<td>111</td>
<td>3</td>
<td>down</td>
</tr>
<tr>
<td>110</td>
<td>101</td>
<td>2</td>
<td>left</td>
</tr>
<tr>
<td>111</td>
<td>100</td>
<td>1</td>
<td>up</td>
</tr>
</tbody>
</table>

Notice that Gray code has the property that only one digit changes as you move from one number to the next. The function `bits` counts the number of ones in it argument using a trick involving the expression `w&(w-1)` as explained on page 50. The sequence of counts for consecutive Gray codes can be regarded as a sequence of directions taken as a curve is drawn, and the following diagrams help to show why this scheme generates the dragon curve.

Notice that the shape of the lines from P to B in diagram (4) is the same as that from A to P, but rotated clockwise through 90 degrees about P and drawn backward.

5.3 Collatz Revisited

The program described in this section concerns the Collatz Conjecture which was introduced in Section 4.16 but has been delayed until this point since it generates a graphical image. It draws a graph showing, on the vertical axis, the length in the range 1 to 250 of the Collatz sequences for starting values in the range 1 to 10000 placed on the horizontal axis. The program is called `collatzgraph.b` and is as follows.

```source
GET "libhdr"
GET "sdl.h"
GET "sdl.b"
GET "libhdr"
```
GET "sdl.h"

MANIFEST {
   nlim = 10000
   clim = 250
}

GLOBAL {
   col_red: ug
   col_green
   col_blue
   col_lightgray
   col_black
}

LET start() = VALOF
{ initsdl()
  mkscreen("Collatz Diagram", 700, 500)

  col_red := maprgb(180, 0, 0)
  col_green := maprgb( 0, 255, 0)
  col_blue := maprgb( 0, 0, 255)
  col_lightgray := maprgb(180, 180, 180)
  col_black := maprgb( 0, 0, 0)

  fillsurf(col_lightgray)

  // Draw the axes
  setcolour(col_black)

  cmoveto( 0, 0)
  cdrawto(nlim, 0)
  cdrawto(nlim, clim)
  cdrawto( 0, clim)
  cdrawto( 0, 0)

  FOR x = 1 TO nlim DO
  { LET y = try(x)
    TEST y>=0
    THEN setcolour(col_red)
    ELSE { setcolour(col_blue)
      y := -y
    }
    cdrawpoint(x, y)
  updatescreen()
\[5.3. \textit{COLLATZ REVISITED}\]

\[
\begin{align*}
\text{AND \ \text{cdrawpoint}(x,y) \ BE} & \{ \text{ // Convert to screen coordinates} \\
& \text{LET sx} = 10 + \text{muldiv(screenxsize-20, x, nlim)} \\
& \text{LET sy} = 10 + \text{muldiv(screenysize-20, y, clim)} \\
& \text{drawfillcircle(sx, sy, 1)} \}
\end{align*}
\]

\[
\begin{align*}
\text{AND \ \text{cmoveto}(x,y) \ BE} & \{ \text{ // Convert to screen coordinates} \\
& \text{LET sx} = 10 + \text{muldiv(screenxsize-20, x, nlim)} \\
& \text{LET sy} = 10 + \text{muldiv(screenysize-20, y, clim)} \\
& \text{moveto(sx, sy)} \}
\end{align*}
\]

\[
\begin{align*}
\text{AND \ \text{cdrawto}(x,y) \ BE} & \{ \text{ // Convert to screen coordinates} \\
& \text{LET sx} = 10 + \text{muldiv(screenxsize-20, x, nlim)} \\
& \text{LET sy} = 10 + \text{muldiv(screenysize-20, y, clim)} \\
& \text{drawto(sx, sy)} \}
\end{align*}
\]

\[
\begin{align*}
\text{AND \ \text{try}(n) \ = \ \text{VALOF}} & \{ \text{ LET count} = 0 \\
& \text{LET lim} = (\text{maxint-1})/3 \\
& \{ \text{ count} := \text{count}+1 \\
& \quad \text{IF n} = 1 \text{ RESULTIS count} \\
& \quad \text{TEST n MOD 2 = 0} \\
& \quad \text{THEN} \{ \text{ n} := \text{n}/2 \\
& \quad \} \\
& \quad \text{ELSE} \{ \text{ IF n} \ > \ \text{lim RESULTIS} \ -\text{count} \\
& \quad \text{n} := 3*\text{n}+1 \\
& \quad \} \} \text{ REPEAT} \\
\}
\end{align*}
\]

When this program is run it generates the following window.
5.4  sdlinfo.b

This section presents a simple program that displays some details of the graphics system. It also displays information about any joysticks that are connected to the system. The program is called sdlinfo.b and is as follows.

```bcpl
/*
This program outputs some information about the current SDL interface.

Implemented by Martin Richards (c) February 2013
*/

GET "libhdr"
GET "sdl.h"
GET "sdl.b"   // Insert the library source code
.
GET "libhdr"
GET "sdl.h"

GLOBAL {
  done:ug
}

LET plotscreen() BE
{ LET maxy = screenysize-1
```
5.4. SDLINFO.B

// Surface info structure
LET flags, fmt, w, h, pitch, pixels, cliprect, refcount = 0, 0, 0, 0, 0, 0, 0, 0

// Format info structure
LET palette, bitsperpixel, bytesperpixel, Rmask, Gmask, Bmask, Amask, Rshift, Gshift, Bshift, Ashift, Rloss, Gloss, Bloss, Aloss, colorkey, alpha = 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

// Video info structure
LET videoflags, blit_fill, video_mem, videoformat = 0,0,0,0

fillsurf(maprgb(120,120,120))

setcolour(maprgb(255,255,255))

sys(Sys_sdl, sdl_getsurfaceinfo, screen, @flags)
sys(Sys_sdl, sdl_getfmtinfo, format, @palette)
sys(Sys_sdl, sdl_videoinfo, @videoflags)

// Screen surface info
plotf(20, maxy- 20, "Screen Surface Info")
plotf(30, maxy- 40,
    "flags=%%x w=%%n h=%%n pitch=%%n",
    flags, w, h, pitch)

// Screen format info
plotf(20, maxy- 80, "Screen Format Info")
plotf(30, maxy-100,
    "palette=%%n bitsperpixel=%%n bytesperpixel=%%n",
    palette, bitsperpixel, bytesperpixel)
plotf(30, maxy-120,
    "Rmask=%%8x Gmask=%%8x Bmask=%%8x Amask=%%8x",
    Rmask, Gmask, Bmask, Amask)
plotf(30, maxy-140,
    "Rshift=%%n Gshift=%%n Bshift=%%n Ashift=%%n",
    Rshift, Gshift, Bshift, Ashift)
plotf(30, maxy-160,
    "Rloss=%%n Gloss=%%n Bloss=%%n Aloss=%%n",
    Rloss, Gloss, Bloss, Aloss)
plotf(30, maxy-180,
    "colorkey=%%8x alpha=%%n",
    colorkey, alpha)
// Video info
plotf(20, maxy-220, "Video Info")
plotf(30, maxy-240,
    "videoflags=%8x blit_fill=%8x video_mem=%n",
    videoflags, blit_fill, video_mem)

{ LET n = sys(Sys_sdl, sdl_numjoysticks)
  plotf(20, maxy-280, "Number of joysticks %2i", n)
  FOR j = 0 TO n-1 DO
    { LET joystick = sys(Sys_sdl, sdl_joystickopen, j)
      LET axes = sys(Sys_sdl, sdl_joysticknumaxes, joystick)
      LET buttons = sys(Sys_sdl, sdl_joysticknumbuttons, joystick)
      LET hats = sys(Sys_sdl, sdl_joysticknumhats, joystick)
      plotf(20, maxy-300-80*j, "Joystick %n", j+1)
      plotf(30, maxy-320-80*j,
        "Number of axes %2i", axes)
      FOR a = 0 TO axes-1 DO
        plotf(250+60*a, maxy-320-80*j,
          "%i7", sys(Sys_sdl, sdl_joystickgetaxis, joystick, a))
        plotf(30, maxy-340-80*j,
          "Number of buttons %2i", buttons)
      FOR b = 0 TO buttons-1 DO
        plotf(250+20*b, maxy-340-80*j,
          "%i2", sys(Sys_sdl, sdl_joystickgetbutton, joystick, b))
        plotf(30, maxy-360-80*j,
          "Number of hats %2i", hats)
    FOR h = 0 TO hats-1 DO
      plotf(250+20*h, maxy-360-80*j,
        "%b4", sys(Sys_sdl, sdl_joystickgethat, joystick, h))
      sys(Sys_sdl, sdl_joystickclose, joystick)
    }
  }
}

AND proessevents() BE WHILE getevent() SWITCHON eventtype INTO
{ CASE sdle_keydown:
  CASE sdle_quit:   done := TRUE
  DEFAULT: LOOP
}

LET start() = VALOF
{ initsdl()
  mkscreen("SDL Info", 800, 500)
5.5. *GRAPHS*

```c

done := FALSE

UNTIL done DO
    { processevents()
        plotscreen()
        updatescreen()
        sdlDelay(50)
    }

writef("*nQuitting*n")
closesdl()
RESULTIS 0
}
```

The main function `start` initialises the SDL interface and then makes a window of size 800x500. It then enters an event loop which it repeatedly executes until `done` is set to `TRUE`. Within the event loop the call of `processevents` sets `done` to `TRUE` if any key is pressed or if the user clicks on the window’s close button.

The call of `plotscreen` interrogates the SDL system and displays some of the information it obtains. It then displays axis, button and hat information about any joysticks that are attached to the system. The call `updatescreen()` sends the window to the display hardware. The loop ends by delaying for 50 milli-seconds.

5.5 *Graphs*

A useful aid to understanding a numerical function is to plot its graph. On graph paper the point \((x, y)\) is located at a distance \(x\) along the horizontal \((x\)-axis) and a distance \(y\) along the vertical \((y\)-axis). The collection of points with coordinates \((x, x^2)\) gives a curve that shows how \(x^2\) changes as we increase \(x\). The following diagram shows the curves for the three functions \(y = x^2\), \(y = x^3 - x\) and \(y = x^3 - x^2 - x\) displayed in red, green and blue, respectively. The program to draw the graph is as follow.

```c
GET "libhdr"
GET "sdl.h"
GET "sdl.b"
.
GET "libhdr"
GET "sdl.h"

GLOBAL {
col_red: ug
col_green
col_blue
col_lightgray
col_black
}

LET start() = VALOF
{ initsdl()
   mkscreen("Three curves", 500, 500)
   col_red := maprgb(255, 0, 0)
   col_green := maprgb(0, 255, 0)
   col_blue := maprgb(0, 0, 255)
   col_lightgray := maprgb(180, 180, 180)
   col_black := maprgb(0, 0, 0)

   fillsurf(col_lightgray)

   // We will use scales numbers with three digits after the
   // decimal point and the $x$ and $y$ ranges will both be
   // between -3.000 and +3.000

   // Draw the axes
   setcolour(col_black)
   FOR x = -3_000 TO 3_000 BY 1_000 DO
   { cmoveto(x, -3_000)
     cdrawto(x, 3_000)
   }
   FOR y = -3_000 TO 3_000 BY 1_000 DO
   { cmoveto(-3_000, y)
     cdrawto(3_000, y)
   }

   plotfn(f1, -3_000, 3_000, col_red)
   plotfn(f2, -3_000, 3_000, col_green)
   plotfn(f3, -3_000, 3_000, col_blue)

   updatescreen()
   sdldelay(20_000)
   closesdl()
   RESULT IS 0
}

AND plotfn(f, x1, x2, col) BE
5.5. GRAPHS

{ setcolour(col)
  cmoveto(x1, f(x1))
  FOR i = 1 TO 100 DO
  { LET x = (x1*(100-i) + x2*i)/100
    cdrawto(x, f(x))
  }
}

AND f1(x) = x*x/3_000

AND f2(x) = f1(x)*x/3_000 - x

AND f3(x) = f1(x) - f2(x)

AND cmoveto(x,y) BE
{ // Convert to screen coordinates
  LET sx = screenxsize/2 + x/15
  LET sy = screenysize/2 + y/15
  moveto(sx, sy)
}

AND cdrawto(x,y) BE
{ // Convert to screen coordinates
  LET sx = screenxsize/2 + x/15
  LET sy = screenysize/2 + y/15
  drawto(sx, sy)
}

This program displays the following window for 20 seconds.
5.6 Gradients

The gradient of a function for a given value of $x$ is a measure of how much it changes when $x$ is changed by a tiny amount. Mathematically, we say that the gradient of $f(x)$ is the limit of $(f(x + dx) - f(x))/dx$ as $dx$ becomes closer and closer to zero. Mathematicians call the gradient the differential of $f(x)$ and represent it using the notation:

$$\frac{d}{dx} f(x)$$

Luckily, for many simple functions there are simple formulae allowing us to compute the differential. For instance, consider the following program (bcplprogs/raspi/slopes.b).

GET "libhdr"
5.6. GRADIENTS

// This program outputs the approximate slope of y = x^n for
// various values of x and n, using scaled numbers with 8 digits
// after the decimal point.

LET start() = VALOF
{ writef(" x n dx slope n**pow(x,n-1)*n^n")
  try( 1.12345678, 0); try( 1.12345678, 1); try( 1.12345678, 2)
  try( 1.12345678, 3); try( 1.12345678, 4)
  newline()
  try( 0.87654321, 0); try( 0.87654321, 1); try( 0.87654321, 2)
  try( 0.87654321, 3); try( 0.87654321, 4)
  newline()
  try(-0.12345678, 0); try(-0.12345678, 1); try(-0.12345678, 2)
  try(-0.12345678, 3); try(-0.12345678, 4)

  RESULTIS 0
}

AND try(x, n) BE
{ LET dx = 0.00010000
  LET slope = muldiv(pow(x+dx,n) - pow(x,n), 1.00000000, dx)
  writef("%11.8d %n %11.8d %11.8d %11.8d*n", x, n, dx, slope, n * pow(x, n-1))
}

AND pow(x, n) = VALOF
{ LET xn = 1.00000000
  FOR i = 1 TO n DO xn := muldiv(xn, x, 1.00000000)
  RESULTIS xn
}

When run, it outputs the following.

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>n</td>
<td>dx</td>
<td>slope</td>
<td>n*pow(x,n-1)</td>
</tr>
<tr>
<td>1.12345678</td>
<td>0</td>
<td>0.00010000</td>
<td>0.000000000</td>
<td>0.00000000</td>
</tr>
<tr>
<td>1.12345678</td>
<td>1</td>
<td>0.00010000</td>
<td>1.000000000</td>
<td>1.00000000</td>
</tr>
<tr>
<td>1.12345678</td>
<td>2</td>
<td>0.00010000</td>
<td>2.247000000</td>
<td>2.24691356</td>
</tr>
<tr>
<td>1.12345678</td>
<td>3</td>
<td>0.00010000</td>
<td>3.786800000</td>
<td>3.78646539</td>
</tr>
<tr>
<td>1.12345678</td>
<td>4</td>
<td>0.00010000</td>
<td>5.672600000</td>
<td>5.67190692</td>
</tr>
<tr>
<td>0.87654321</td>
<td>0</td>
<td>0.00010000</td>
<td>0.000000000</td>
<td>0.00000000</td>
</tr>
</tbody>
</table>
This seems to imply that
\[
\frac{d}{dx} x^n = n \times x^{n-1}
\]

We can convince ourselves that this is indeed correct by the following derivation.
\[
\begin{align*}
\frac{d}{dx} x^n &= \frac{((x+dx)\times(x+dx)\times\ldots\times(x+dx))-x^n}{dx} \\
&= \frac{x^n+n\times x^{n-1}dx+O(dx^2)-x^n}{dx} \\
&= \frac{x^{n-1}dx+O(dx^2)}{dx} \\
&= n \times x^{n-1} + O(dx) \\
&= n \times x^{n-1} + O(dx)
\end{align*}
\]
where the notation \(O(dx)\) stands for terms that all have \(dx\) as a factor, so tend to zero as \(dx\) becomes smaller and smaller.

Using this formula we can easily see that
\[
\frac{d}{dx} (\frac{x^n}{n!}) = \frac{n \times x^{n-1}}{n!} = \frac{x^{n-1}}{(n-1)!}
\]
This allows us to deduce a remarkable property of \(e^x\), namely
\[
\frac{d}{dx} e^x = \frac{d}{dx} (1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \ldots) \\
= 0 + 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots \\
= e^x
\]
5.7 Events

This section demonstrates how input from the keyboard, mouse and joystick can be handled. The program displays a coloured circle in a window. Its colour may be changed to red, green or blue by pressing R, G or B on the keyboard, or by buttons on the joystick. It can be moved up, down, left or right by pressing the arrow keys, and it may be dragged using the mouse with a mouse button pressed. It may also be moved using the joystick. You can exit from the program by pressing Q. The program starts as follows.

GET "libhdr"
GET "sdl.h"
GET "sdl.b"

GET "libhdr"
GET "sdl.h"

GLOBAL {
  done:ug
  xpos; ypos; xdot; ydot
  col_blue; col_green; col_red
  col_cyan; col_white; col_gray
}

LET start() = VALOF
{ initSDL()
  mkscreen("Events Test", 600, 400)
  runtest()
  closesdl()
  RESULTIS 0
}

AND runtest() = VALOF
{ // Declare a few colours in the pixel format of the screen
  col_blue := maprgb(0, 0, 255)
  col_green := maprgb(0, 255, 0)
  col_red := maprgb(255, 0, 0)
CHAPTER 5. INTERACTIVE GRAPHICS IN BCPL USING SDL

```
col_cyan := maprgb(255, 255, 0)
col_white := maprgb(255, 255, 255)
col_gray := maprgb(128, 128, 128)

fillscreen(col_gray)

xpos, ypos := 1000*screenxsize/2, 1000*screenysize/2
xdot, ydot := 0, 0
setcolour(col_red) // Set the initial circle colour
done := FALSE

UNTIL done DO
{ step()
displayall()
sdldelay(20)
}

RESULTIS 0
}

runtest creates a few colours, fills the screen with a gray colour and initialises, xpos, ypos, xdot, ydot and done. The first two are scaled numbers with three digits after the decimal point representing the coordinates on the screen of the location of the small coloured circle. Mathematicians often use the notation $\dot{x}$ and $\dot{y}$ to represent the rate at which $x$ and $y$ change with time. In this program we use the names xdot and ydot to hold the rate of change of xpos and ypos. These rates depend on the joystick position. The variable done is set to TRUE when the user wishes to exit from the program.

The program now enters an UNTIL loop that repeatedly reads and processes events from the keyboard, mouse and joystick. These events may change the colour and position of the coloured circle, so the window is redrawn by the call displayall() each time round the loop. The call sdldelay(20) causes a real time delay of 20 milli-seconds so that the screen is updated about 50 times per second independent of the CPU speed of the computer. The program thus has a similar timing behaviour even when run on computers of different processing power.

Finally the definition of step is as follows.

```

AND step() BE
{ WHILE getevent() SWITCHON eventtype INTO
{ DEFAULT: LOOP

CASE sdl_keydown:
    SWITCHON capitalch(eventa2) INTO
```

runtest creates a few colours, fills the screen with a gray colour and initialises, xpos, ypos, xdot, ydot and done. The first two are scaled numbers with three digits after the decimal point representing the coordinates on the screen of the location of the small coloured circle. Mathematicians often use the notation $\dot{x}$ and $\dot{y}$ to represent the rate at which $x$ and $y$ change with time. In this program we use the names xdot and ydot to hold the rate of change of xpos and ypos. These rates depend on the joystick position. The variable done is set to TRUE when the user wishes to exit from the program.

The program now enters an UNTIL loop that repeatedly reads and processes events from the keyboard, mouse and joystick. These events may change the colour and position of the coloured circle, so the window is redrawn by the call displayall() each time round the loop. The call sdldelay(20) causes a real time delay of 20 milli-seconds so that the screen is updated about 50 times per second independent of the CPU speed of the computer. The program thus has a similar timing behaviour even when run on computers of different processing power.

Finally the definition of step is as follows.

```

AND step() BE
{ WHILE getevent() SWITCHON eventtype INTO
{ DEFAULT: LOOP

CASE sdl_keydown:
    SWITCHON capitalch(eventa2) INTO
```
When the user presses a key on the keyboard, moves the mouse or joystick, or
presses a mouse or joystick button, the system creates an event held in an event queue. These events can be inspected, one at a time, by calling `getevent()`. If there are no outstanding events `getevent` returns `FALSE`, otherwise it updates the global variable `eventtype` and possibly some event arguments `eventa1`, `eventa2`, `eventa3`, etc. As we will see later, which event arguments are set depends on the event type. The possible event types are declared in `sdl.h` and have names starting with `sdle_`, such as `sdle_keydown` or `sdle_joyaxismotion`.

If the type was `sdle_keydown`, the argument `eventa2` will identify which key pressed. As can be seen, the program is only interested in the arrow keys and the letters R, G, B and Q. The arrow keys cause the coordinates `xpos` and `ypos` to change, R, G, B cause the colour of the circle to change and Q sets `done` to `TRUE` causing execution of the program to terminate.

If the type was `sdle_mousebuttondown`, the arguments `eventa2` and `eventa3` give the coordinates of the mouse. These are used to set the coordinates of the centre of the coloured circle.

If the type was `sdle_mousemotion`, the arguments `eventa2` and `eventa3` give the coordinates of the mouse. `eventa1` is a bit pattern identifying which of the mouse buttons are currently pressed, and if any are, the coloured circle is moved to the cursor position.

If the type was `sdle_joyaxismotion`, the arguments `eventa2` and `eventa3` identify which axis has moved and what it new value is. With the Logitech Attack 3 joystick there are three axes, elevator, aileron and throttle and their values range from -32768 to +32767. The elevator and aileron values are used to control how fast our coloured circle moves across the screen.

The event type `sdle_quit` occurs when the user clicks on the little cross at the top right hand corner of the window indicating that the program should terminate. All that `step` does in this case is to set `done` to `TRUE` causing execution to leave the event loop.

The final function, `displayall`, just fills the screen with gray, draws the coloured circle in it new position, ensuring that it is still within the window, and finally `displayall` calls `updatescreen` to update the video hardware. Its definition is as follows.

```bcpl
AND displayall() BE
{ LET x, y = xpos/1000, ypos/1000
  LET minx, miny = 20, 20
  LET maxx, maxy = screenxsize-20, screenysize-20
  fillscreen(col_gray)

  IF x<minx DO x, xpos := minx, minx*1000
  IF y<miny DO y, ypos := miny, miny*1000
  IF x>maxx DO x, xpos := maxx, maxx*1000
  IF y>maxy DO y, ypos := maxy, maxy*1000
}
```
5.8 $e^{ix}$ and rotation

We all know that when we square a number the result is positive. For example, $2^2 = 4$ and $(-3)^2 = 9$. But mathematicians are not satisfied with this since they sometimes find it useful to take the square root of negative numbers. You might think they are mad but let us see what they do and why it is useful. The trick is to postulate a new number $i$ having the property that $i^2 = -1$. Such a number, of course, cannot exist so they call it an imaginary number. They let it obey all the normal algebraic rules that ordinary (real) numbers have. Using $i$ we can make complex numbers such as $2 + 3i$, and these also obey the normal rules of algebra. For instance, we can multiply them as in

$$(a + ib) \times (c + id) = ac + i^2bd + aid + ibc = (ac - bd) + i(ad + bc)$$

We have seen the series for $e^x$ in Section 4.31 which was as follows

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots$$

If we substitute $ix$ for $x$ in this equation we get an equation with some very interesting properties.

$$e^{ix} = 1 + ix + \frac{i^2x^2}{2!} + \frac{i^3x^3}{3!} + \frac{i^4x^4}{4!} + \frac{i^5x^5}{5!} + \ldots$$

$$= 1 + ix - \frac{x^2}{2!} - \frac{ix^3}{3!} + \frac{x^4}{4!} + \frac{ix^5}{5!} + \ldots$$

$$= (1 - \frac{x^2}{2!} + \frac{x^4}{4!} + \ldots) + i(x - \frac{x^3}{3!} + \frac{x^5}{5!} + \ldots)$$

The real and imaginary parts of $e^{ix}$ are so important they are given the names cosine and sine, normally written as $\cos x$ and $\sin x$.

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} + \ldots$$

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} + \ldots$$

Notice that if we change the sign of $x$, all the terms in the cos series remain unchanged, but those in the sin series are all negated, so
$\cos(-x) = \cos x$

$\sin(-x) = -\sin x$

Notice, also, that

$$e^{ix} \times e^{-ix} = e^{ix-ix} = e^0 = 1$$

But

$$e^{ix} \times e^{-ix} = (\cos x + i \sin x) \times (\cos (-x) + i \sin (-x))$$

$$= (\cos x \times \cos (-x) - \sin x \times \sin (-x)) + i(\cos x \times \sin (-x) - \sin x \times \cos (-x))$$

$$= (\cos^2 x + \sin^2 x) + i(-\cos x \times \sin x + \sin x \times \cos x)$$

$$= \cos^2 x + \sin^2 x$$

So

$$\cos^2 x + \sin^2 x = 1$$

Using the formula

$$\frac{d}{dx} \left( \frac{x^n}{n!} \right) = \frac{x^{n-1}}{(n-1)!}$$

that we derived earlier, we can easily obtain the following two results.

$$\frac{d}{dx} \sin x = \frac{d}{dx} (x - \frac{x^3}{3!} + \frac{x^5}{5!} + \ldots)$$

$$= 1 - \frac{x^2}{2!} + \frac{x^4}{4!} + \ldots$$

$$= \cos x$$

and

$$\frac{d}{dx} \cos x = \frac{d}{dx} (1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots)$$

$$= -x + \frac{x^3}{3!} - \frac{x^5}{5!} + \ldots$$

$$= -\sin x$$
It turns out that the arguments of cos and sin are best thought of as angles and, since mathematicians like to use greek letters for angles, we will use letters such as $\theta$ and $\phi$ in place of $x$ and $y$, saving $x$ and $y$ for horizontal and vertical coordinates on graph paper.

It is instructive to see how $\cos \theta$ and $\sin \theta$ change as $\theta$ varies from 0 to $2\pi$. The following program plots them with the curve for $\cos \theta$ in red and the curve for $\sin \theta$ in green. It also plots the points with coordinate $(\cos \theta, \sin \theta)$ in blue centred on the graph. The program uses variants of several of the functions used in the evaluation of $e^{\pi \sqrt{163}}$ given in Section 4.32, and as with the previous program we use multi digit numbers of radix 10000 held in vectors, but this time the upper bound is 4 which is sufficient for a precision of nearly 16 decimal digits after the decimal point. If $v$ is such a number, then $10000*v!0+v!1$ is the equivalent scaled fixed point number with 4 decimal digits after the decimal point. The digit in $v!0$ is signed, but all the other digits are positive in the range 0 to 9999. This convention is somewhat analogous to the interpretation of the bits in a 2s complement signed binary numbers.

The program (which is in \texttt{bcplprogs/raspi/cossin.b}) starts as follows.

```
// Insert the SDL library source code as a separate section

GET "libhdr"
GET "sdl.h"
GET "sdl.b".
GET "libhdr"
GET "sdl.h"

GLOBAL {
  x0:ug // The scaling parameters
  y0
  scale
    col_white; col_blue; col_green; col_red; col_gray; col_black
}

MANIFEST { upb = 4 }

LET start() = VALOF
{ initsdl()
  mkscreen("Cosine and sine curves", 800, 400)

    // Declare a few colours in the pixel format of the screen
    col_white := maprgb(255, 255, 255)
    col_black := maprgb( 0, 0, 0)
    col_blue := maprgb( 0, 0, 225)

```
col_green := maprgb(0, 185, 0)
col_red := maprgb(195, 0, 0)
col_gray := maprgb(228, 228, 228)

fillscreen(col_gray)
updatescreen() // Update the screen hardware

setscaling() // Set the scaling parameters for smoveto etc.

setcolour(col_black); plotgraphpaper()
setcolour(col_red); plot_fn(cosine)
setcolour(col_green); plot_fn(sine)
setcolour(col_blue); plotcircle()

updatescreen() // Update the screen hardware
sdldelay(20_000) // Pause for 20 secs

closedsdl()
RESULTIS 0
}

All that remains is to define the plotting functions and the one that sets the scaling parameters so that the graph will appear appropriately sized and centred in the window.

The graph paper ranges from 0.0000 to 2 × 3.1415 in the x (horizontal) direction and from -1.0000 to +1.0000 in the y (vertical) direction with (0, -1) being the bottom left corner of the graph. Lines will be drawn using the functions smoveto and sdrawto which both take scaled fixed point numbers with 4 digits after the decimal point to specify the coordinate on the graph paper. They are defined as follows.

AND smoveto(x, y) BE
{ LET screenx = x0 + muldiv(x, scale, 1_000_000)
AND screeny = y0 + muldiv(y, scale, 1_000_000)
moveto(screenx, screeny)
}

AND sdrawto(x, y) BE
{ LET screenx = x0 + muldiv(x, scale, 1_000_000)
AND screeny = y0 + muldiv(y, scale, 1_000_000)
drawto(screenx, screeny)
updatescreen() // Update the screen
sdldelay(20) // So we can see the curves being drawn
}
5.8. $e^{ix}$ AND ROTATION

Both these functions use the scaling parameters $x_0$, $y_0$ and $scale$ to transform the graph paper coordinates to coordinates on the window. Notice also that `sdrawto` updates the screen and has a slight real time delay so that we can watch the graphs being drawn. The scaling parameters are set by the next function

```plaintext
AND setscaling() BE
{ // Set the scaling parameters x0, y0 and scale used by smoveto
    // and sdrawto so that the drawing area from x = 0 to 2 pi and
    // y = -1.0 to +1.0 appears centered in the window.
    // The conversion from graph coordinates (x, y) to
    // screen coordinates will be as follows

    // screenx = x0 + muldiv(x, scale, 1_000_000)
    // screeny = y0 + muldiv(y, scale, 1_000_000)

    x0 := screenxsize / 20
    y0 := screenysize / 2
    scale := muldiv(screenxsize*9/10, 1_000_000, 2 * 3_1415)
}
```

Next comes the plotting functions. The first draws the graph paper consisting of lines for the edges, the $x$ axis and vertical lines at $\pi/2$, $\pi$ and $3\pi/2$.

```plaintext
AND plotgraphpaper() BE
{ FOR i = -1 TO +1 DO
    { // Draw horizontal lines at -1.0000, 0 and 1.0000
        smoveto( 0, i * 1_0000)
        sdrawto( 2*3_1415, i * 1_0000)
    }
    FOR i = 0 TO 4 DO
    { // Draw vertical lines at 0, pi/2, pi 3pi/2 and 2pi
        smoveto( i*3_1415/2, -1_0000)
        sdrawto( i*3_1415/2, +1_0000)
}
}
```

The next function `plot_fn` is used to plot the cosine and sine curves. It takes an argument $f$ which is either `cosine` or `sine` and draws the curve as a sequence of 100 short line segments. It uses a multi digit representation of the angle $\theta$ which it passes to $f$ each time a new value is to be computed. The values of $\theta$ are of the form $2n\pi/100$ for $n$ in the range 0 to 100. It uses `mulbyk` and `divbyk` defined later.
AND plot_fn(f) BE FOR n = 0 TO 100 DO
{ // Plot f(theta) from theta = 0 to 2 pi
  LET theta = VEC upb
  LET pi = TABLE 3, 1415, 9265, 3589, 7932
  FOR j = 0 TO upb DO theta!j := pi!j // Set theta = pi
  mulbyk(theta, 2*n)
  divbyk(theta, 100)
  TEST n=0
  THEN smoveto(10000*theta!0+theta!1, f(theta))
  ELSE sdrawto(10000*theta!0+theta!1, f(theta))
}

The function plotcircle has much in common with plot_fn but draws short
line segments between points with coordinates (cos(θ), sin(θ)). A scaled number
representing 3.1415 is added to the x coordinate to place the circle at the center
of the graph.

AND plotcircle() BE FOR n = 0 TO 100 DO
{ LET theta = VEC upb
  LET pi = TABLE 3, 1415, 9265, 3589, 7932
  FOR i = 0 TO upb DO theta!i := pi!i // Set theta = pi
  mulbyk(theta, 2*n)
  divbyk(theta, 100)
  TEST n=0
  THEN smoveto(cosine(theta)+3_1415, sine(theta))
  ELSE sdrawto(cosine(theta)+3_1415, sine(theta))
}

The functions cosine and sine compute multi digit representations of cos θ
and sin θ using the two series we have already seen, namely:

\[
\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} + \ldots
\]
\[
\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} + \ldots
\]

Since these series have much in common, cosine and sine both use an auxiliary
function sumseries(theta, n) to perform the summation. theta is a multi
digit representation of θ and n=0 for cosine and n=1 for sine. The function is
defined as follows.

AND sumseries(theta, n) = VALOF
{ // n=0 return cosine theta as a scaled number with 4 decimal
  // digits after the decimal point
// n=1 return sine theta as a scaled number with 4 decimal
digits after the decimal point
LET sum = VEC upb
LET term = VEC upb // Next term to add, x^n/n!
LET negt2 = VEC upb // To hold -theta^2

FOR i = 0 TO upb DO sum!i, term!i := 0, 0 // Set sum and term to zero
term!0 := 1 // Set sum to 1.0000
IF n DO mult(term, term, theta) // Set term for sine
FOR i = 0 TO upb DO negt2!i := theta!i // Set negt2 = theta
mult(negt2, negt2, negt2) // negt2 now holds theta^2
neg(negt2, negt2) // negt2 now hold -theta^2
UNTIL iszero(term) DO
{ add(sum, sum, term) // Accumulate the current term
  mult(term, term, negt2) // Calculate the next term in the series
  divbyk(term, n+1)
  divbyk(term, n+2)
  n := n+2
}

RESULTIS 1_0000*sum!0 + sum!1 // Return a fix point scaled number
}

AND iszero(v) = VALOF
{ FOR i = 0 TO upb IF v!i RESULTIS FALSE
  RESULTIS TRUE
}

The definition of sumseries should be reasonably understandable. It accumulates the result in sum by adding the next term (held in term) until term represents zero. The next term is computed from the previous one by multiplying by \(-\frac{\theta^2}{(n+1)(n+2)}\) incrementing n by 2 each time. The initial value of term represents either 1 for cosine or \(\theta\) for sine. Once the series has been summed, it is converted to a scaled fixed point number with 4 decimal digits after the decimal point by the expression 1_0000*sum!0 + sum!1. Finally cosine as sine are defined by suitable calls of sumseries.

AND cosine(theta) = sumseries(theta, 0)
AND sine(theta) = sumseries(theta, 1)
All that remains is to define the low level functions to perform arithmetic on our multi digit representation of signed numbers. The first of these is `mult` which computes the product of the numbers in `y` and `z` storing the result in `x`. The comments explain how it works.

```bcpl
AND mult(x, y, z) BE
{ // Set x to the product of y and z
  // x, y and z need not be distinct, so copies are made.
  LET res = VEC upb+3 // res includes some guard digits
  LET cy = VEC upb  // cy and cz will hold copies of y and z
  LET cz = VEC upb
  LET resneg = FALSE

  // Make copies of y and z
  FOR i = 0 TO upb DO cy!i, cz!i := y!i, z!i
  // Set res to zero
  FOR i = 0 TO upb+3 DO res!i := 0
  // Rounding of the result is done by adding 1/2 to the last digit
  res!(upb+1) := 5000

  IF cy!0<0 DO { neg(cy, cy); resneg := ~resneg }
  IF cz!0<0 DO { neg(cz, cz); resneg := ~resneg }

  // cy and cz now both represent positive numbers
  FOR i = 0 TO upb IF cy!i FOR j = 0 TO upb+3-i DO
    { LET p = i + j // Destination in range 0 to upb+3
      LET d = res!p + cy!i * cz!j
      LET carry = d / 10000
      IF p=0 DO { res!0 := d; LOOP } // res!0 is allowed to be >= 10000
      res!p := d MOD 10000
    }
  }

  TEST resneg
  THEN neg(x, res) // Set x = -res
  ELSE FOR i = 0 TO upb DO x!i := res!i // Set x = res
}
The next function copies the negated value of y into x. It is perhaps best understood by considering the operation on a number with only one digit (of radix 10000) after the decimal point. Suppose num represents 1.2345, then num\!0=1 and num\!1=2345. Our representation -1.2345 has num\!0=-2 and num\!1=7655 since the fractional part is positive. This result can be computed as follows. First negate both the integer and fractional parts giving num\!0=-1 and num\!1=-2345, then correct the fractional part by adding 10000 to it and subtracting 1 from the integer part in compensation. The addition 10000 can be done by adding 9999 and then incrementing the result. The fractional part thus becomes 9999-2345+1 = 7654+1 = 7655. Note that the addition of 1 causes a carry of 1 into the integer part, if the original fractional part was zero.

\begin{verbatim}
AND neg(x, y) BE
{ // Set x to -y
  LET carry = 1
  FOR i = upb TO 1 BY -1 DO
  { LET d = 9999 - y!i + carry
    x!i := d MOD 10000
    carry := d / 10000
  }
  x!0 := carry - y!0 -1
}
\end{verbatim}

The \texttt{add} function adds corresponding digits of y and z starting from the least significant end, dealing with carries as it goes. The result is placed in x. Note that the fraction digits are all positive but the integer part (in element zero) is signed and need not be in the range -9999 to +9999.

\begin{verbatim}
AND add(x, y, z) BE
{ LET carry = 0
  FOR i = upb TO 1 BY -1 DO
  { LET d = y!i + z!i + carry
    x!i := d MOD 10000
    carry := d / 10000
  }
  x!0 := y!0 + z!0 + carry
}
\end{verbatim}

Subtraction is performed by negating z then calling \texttt{add}.

\begin{verbatim}
AND sub(x, y, z) BE
{ // Set x = y - z
  // Copy z because it might be the same as y
}
\end{verbatim}
LET cz = VEC upb
    neg(cz, z)
    add(x, y, cz)
}

The function `mulbyk` multiplies the multi digit signed number in v by the integer k placing the result back in v. It conditionally changes the signs of v and k so the multiplication is performed on positive values. It then changes the sign of v again at the end, if needed.

AND mulbyk(v, k) BE
{ LET carry = 0
    LET resneg = FALSE
    IF v!0<0 DO { neg(v, v); resneg := ~resneg }
    IF k<0 DO { k := -k; resneg := ~resneg }

    FOR i = upb TO 1 BY -1 DO
    { LET d = v!i * k + carry
      v!i := d MOD 10000
      carry := d / 10000
    }
    v!0 := v!0 * k + carry

    IF resneg DO neg(v, v)
}

The function `divbyk` divides the multi digit signed number in v by the integer k placing the result back in v.

AND divbyk(v, k) BE
{ LET carry = 0
    LET resneg = FALSE
    IF v!0<0 DO { neg(v, v); resneg := ~resneg }
    IF k<0 DO { k := -k; resneg := ~resneg }

    FOR i = 0 TO upb DO
    { LET d = carry*10000 + v!i
      v!i := d / k
      carry := d MOD k
    }
    IF resneg DO neg(v, v)
}
When the above program runs, it creates the window shown below containing the curves for $\cos \theta$ in red, $\sin \theta$ in green and a circle in blue. The short delay in \texttt{sdrawto} allows you to see these curves being drawn.

Before leaving this section, there is one last formula we need to derive. Looking at the blue circle drawn by the previous program, it is clear the coordinates $(\cos \theta, \sin \theta)$ lie on a circle of radius one. $\theta$ is not measured in degrees but in \textit{radians} which is the distance around the circumference of the unit circle from the point $(1, 0)$. Thus $\theta = 2\pi$ corresponds to an angle of $360^\circ$.

Let us assume a point $P$ on the unit circle is at an angle $\phi$ from the $x$ axis and that its coordinates are $(x, y) = (\cos \phi, \sin \phi)$. If we wanted to rotate $P$ anti-clockwise by an angle $\theta$ to point $Q$, it would move to $(X, Y) = (\cos(\theta + \phi), \sin(\theta + \phi))$. It would be really useful to have formulae that compute these coordinates in terms of the old ones and $\theta$, and this can easily be done by considering $e^{i(\theta+\phi)}$ as follows
\[
e^{i(\theta+\phi)} = \cos(\theta + \phi) + i \sin(\theta + \phi)
\]

But also
\[
e^{i(\theta+\phi)} = e^{i\theta} \times e^{i\phi} = (\cos \theta + i \sin \theta) \times (\cos \phi + i \sin \phi) = (\cos \theta \cos \phi - \sin \theta \sin \phi) + i(\sin \theta \cos \phi + \cos \theta \sin \phi)
\]

So
\[
\cos(\theta + \phi) = \cos \theta \cos \phi - \sin \theta \sin \phi \\
\sin(\theta + \phi) = \sin \theta \cos \phi + \cos \theta \sin \phi
\]
Remembering the old coordinates were \((x, y) = (\cos \phi, \sin \phi)\), we can calculate the new coordinates \((X, Y) = (\cos(\theta + \phi), \sin(\theta + \phi))\) as follows

\[
\begin{align*}
X &= \cos \theta \times x - \sin \theta \times y \\
Y &= \sin \theta \times x + \cos \theta \times y
\end{align*}
\]

Mathematicians usually prefer to write these two equations as a single equation have exactly the same meaning using what is called matrix notation.

\[
\begin{pmatrix}
X \\
Y
\end{pmatrix} =
\begin{pmatrix}
\cos \theta & -\sin \theta \\
\sin \theta & \cos \theta
\end{pmatrix}
\begin{pmatrix}
x \\
y
\end{pmatrix}
\]

It is easy to see that these formulae work just as well when \((x, y)\) is not on the unit circle but on a circle of radius \(r\), say. These formulae will be used later when we wish to rotate, for example, the moon lander space craft. To see a geometric proof of the \(\cos(\theta + \phi)\) equation do a web search on: cos a plus b geometric proof.

Note that when \(\theta\) is small enough to allow us to ignore terms such as \(\frac{\theta^2}{2!}\) and \(\frac{\theta^3}{3!}\) then from the series we can deduce that \(\cos \theta\) is approximately 1 and \(\sin \theta\) is approximately \(\theta\). We take advantage of these approximations when dealing with small rotations in implementation of the flight simulator given later.

To summarise this section, we started by considering the impossible number \(i\) whose square is \(-1\) and then thought of the equally mind boggling idea of computing \(e^{ix}\), that is multiplying 1 by \(e\), \(ix\) times. This resulted in two functions, \(\cos\) and \(\sin\), which, when plotted, looked beautiful and rather similar. We even showed that \(\cos^2 \theta + \sin^2 \theta = 1\) which was confirmed by plotting points of the form \((\cos \theta, \sin \theta)\) showing they all lay on the unit circle. We went on to deduce formulae for \(\cos(\theta + \phi)\) and \(\sin(\theta + \phi)\) which we will be used later in this chapter. What this tells us is that mathematics is not just about learning multiplication tables and doing tedious numerical sums, but is more to do with extraordinary ideas and beautiful results obtained with the aid of a little simple algebra. Some of the results turn out to be very useful, while others, like Euler’s identity \(e^{i\pi} + 1 = 0\), are just wonderous to observe. (Try a web search on: e to the i pi plus one equals zero.)

If you have reached this far in this section you are either already a mathematician or well on the way to becoming one. Well done!
5.9 Polar Coordinates

We saw in the previous section that complex numbers can be thought of as points on a two dimensional graph, with the horizontal and vertical axes representing the real and imaginary components, respectively. Such a graph is often called an Argand diagram and is useful in helping to understand how complex numbers behave. A complex number \( z = x + iy \) can be represented by the point in the Argand diagram with cartesian coordinates \((x, y)\). However, we can also describe it by the pair \( (r, \theta) \) where \( r \) is the distance between \( z \) and the origin, and \( \theta \) is the angle between the line from the origin to \( z \) and the real axis. The quantities \( r \) and \( \theta \) are called polar coordinates, and this representation turns out to be very useful. The conversion from polar coordinate \( (r, \theta) \) to cartesian coordinates \( (x, y) \) is easy, since \( x = r \cos \theta \) and \( y = r \sin \theta \). So, \( z = r \cos \theta + ir \sin \theta \) which, as we saw in the previous section, can also be written as \( re^{i\theta} \).

The product of two complex numbers \( re^{i\theta} \) and \( se^{i\phi} \) is \( rs e^{i(\theta+\phi)} \). So, using polar coordinates, the product of \( (r, \theta) \) and \( (s, \phi) \) is \( (rs, \theta + \phi) \). It is thus clear that when we multiply two complex numbers together, the polar distance of the result is the product of the polar distances of the two operands, and the polar angle is the sum of the angles of the two operands.

If we consider a number \( (r, \theta) \) on or inside the unit circle in the Argand diagram then \( r \) will be less than or equal to one and the square of \( (r, \theta) \) will still be within the unit circle. If, on the other hand, \( r > 1 \) the square will be further away from the origin and repeatedly squaring the result will cause it to diverge to infinity. Thus if we apply this repeated squaring process to arbitrary initial values, we only avoid divergence for all initial values on or inside the unit circle. This mechanism defines the set of points inside or on the unit circle and the boundary of this set is the unit circle itself.

5.10 The Mandelbrot Set

Benoit Mandelbrot considered a slight variation of the repeated squaring process. Every time \( z \) is squared a small complex constant \( c \) is added to the result. So the process involves repeated performing \( z := z^2 + c \). He chose to start with \( z = 0 \). For some values of \( c \), such as \( c = 3 \), the process diverges, and for other settings, such as \( c = 0 \) or \( c = -1 \), the values of \( z \) remain bounded. The possible values of \( c \) that cause the process to remain bounded is called the Mandelbrot set, and it turns out to have some extraordinarily unexpected properties. The program presented here displays a specified square region of the Mandelbrot set by performing the iteration a limited number of times for all possible values of \( c \) in the square. If \( z \) remains within three units of the origin throughout all the iterations, \( c \) is in or close to the Mandelbrot set and is plotted as a black pixel. If, on the other hand, \( z \) moves further than three units from the origin, the process
is clearly going to diverge and the corresponding pixel is given a colour depending on how many iterations were required for $z$ to escape. The resulting picture is sometimes rather surprising.

The program is called `bcplprogs/raspi/mandset.b` and starts as follows.

```bcpl
// Insert the SDL library source code as a separate section

GET "libhdr"
GET "sdl.h"
GET "sdl.b"
.
GET "libhdr"
GET "sdl.h"

GLOBAL {
   a:ug
   b
   size
   limit    // The iteration limit
   v
   col_white; col_gray; col_black
}

MANIFEST {
   One = 100_000_000 // The number representing 1.00000000
   width=512
   height=width     // Ensure the window is square
}

The global variables $a$, $b$ and $size$ will hold the details of a square region to display with sides of length $2\times size$ centred at position $(a,b)$, and $limit$ is the upper limit of the number of iterations to use.

The manifest constant $One$ gives the integer value of the scaled numbers used in the calculation. This allow for number in about the range $-20.0$ to $+20.0$ to be represented with 8 decimal digits after the decimal point. The main program is as follows.

```
5.10. THE MANDELBROT SET

v := 0

// Default settings
a, b, size := -50_000_000, 0, 180_000_000
limit := 38

IF argv!0 DO s := !argv!0 // s/n
IF argv!1 DO a := !argv!1 // a/n
IF argv!2 DO b := !argv!2 // b/n
IF argv!3 DO size := !argv!3 // size/n
IF argv!4 DO limit := !argv!4 // limit/n

IF 1<=s<=7 DO
{ LET limtab = TABLE 38, 38, 38, 54, 70, // 0
    80, 90, 100, 100, 110, // 5
    120, 130, 140, 150, 160, // 10
    170, 180, 190, 200, 210, // 15
    220 // 20

    limit := limtab!s
    a, b, size := -52_990_000, 66_501_089, 50_000_000
    FOR i = 1 TO s DO size := size / 10
}

initsdl()
mkscreen("Mandlebrot Set", width, height)

// Declare a few colours in the pixel format of the screen
col_white := maprgb(255, 255, 255)
col_gray := maprgb(128, 128, 128)
col_black := maprgb(0, 0, 0)

v := getvec(width*height-1)
// Initialise v the vector of random pixel addresses.
FOR i = 0 TO width*height - 1 DO v!i := i
// Random shuffle v so that the screen pixels are filled in
// in random order.
FOR i = width*height - 1 TO 1 BY -1 DO
{ LET j = randno(i+1) - 1 // Random number in range 0 .. i
    LET t = v!j
    v!j := v!i
    v!i := t
}
plotset()

setcolour(col_white)
plotf(5, 50, "s = "); plotf(50, 50, " %i4\ast n", s)
plotf(5, 35, "a = %11.8d b = %11.8d size = %11.8d", a, b, size)
plotf(5, 20, "limit = "); plotf(50, 20, " %i4\ast n", limit)
updatescreen()

sdldelay(60_000) //Pause for 60 secs
closesdl()
IF v DO freevec(v)
RESULTIS 0
}

The program takes five possible arguments $s$, $a$, $b$, $size$ and $limit$ all of
which are numeric. The first argument can be used to select one of 7 interesting
regions to display, the next three can specify other regions, and $limit$ can be
used to set the iteration limit.

The vector $v$ is initialised to the integers 0 to $width\ast height$-1 stored in
random order. Each element holds the $x$, $y$ coordinates of a pixel position packed
as two adjacent 9-bit values. The $i^{th}$ pixel to be drawn will be at $(x, y)$ position
$(v!i\&#x1FF,(v!i>>9)&#x1FF)$. This vector holds the random order in which the
pixels are drawn.

The mandelbrot set is then plotted by the call plotset(). Some text is then
written to the screen specifying the position and size of the region displayed. A
colour bar is then drawn near the bottom of the screen to show the mapping be-
tween iteration count and the corresponding colour. Finally the screen is updated
and displayed for 60 seconds.

AND colfill(p, m, col1, col2) BE
{ //writef("colfill: p=%i5 m=%i3 col1=%o9 col2=%o9\*n", p, m, col1, col2)
 //abort(1000)
TEST m<=1
THEN { putcolour(p, 0, col1)
}
ELSE { // Fill p!0 to p!(m-1) with colours using linear
 // interpolation.
LET m2 = m/2 // Midpoint
LET midcol = (col1+col2)/2 // Midpoint colour
colfill(p, m2, col1, midcol)
colfill(p+m2, m-m2, midcol, col2)
}
}

AND putcolour(p, i, col) BE
5.10. THE MANDELBROT SET

{ LET r, g, b = (col>>18)&255, (col>>9)&255, col&255
  // writef("putcolour: p=%i6 i=%i3 col=%o9 r=%i3 g=%i3 b=%i3*n",
  // p, i, col, r, g, b)
  //abort(1000)
  p!i := maprgb(r, g, b)
}

AND setpalette(p, lim, colv, n) BE
{ // Fill in colours in p!0 to p!lim based on
  // the colours in colv!0 to colv!n
  //writef("setpalette: p=%i5 lim=%i3 colv=%i5 n=%i3*n", p, lim, colv, n)
  //abort(1000)
  IF lim<=n DO
    { FOR i = lim TO 0 BY -1 DO { putcolour(p, i, colv!n); n := n-1 }
      RETURN
    }
  IF lim - lim/4 >= n DO
    { LET m = lim/4
      colfill(p, m, colv!0, colv!1)
      setpalette(p+m, lim-m, colv+1, n-1)
      RETURN
    }
  // Copy colours from colv! to colv!n to p!(lime-n+1) to p!lim
  WHILE n>0 DO
    { putcolour(p, lim, colv!n)
      lim, n := lim-1, n-1
    }
  colfill(p, lim+1, colv!0, colv!1)
}

These few functions construct a colour palette in colourv that depends on the selected iteration limit. The colours are chosen so that they move from yellow through white to various shaded of green, and for points most distant from the Mandelbrot the various shades of blue are chosen.

AND plotset() BE
{ // The following table hold 8-bit rgb colours packed
  // in three 9-bit fields. It is used to construct a palette
  // of colours depending on the current limit setting.
  LET coltab = TABLE
    #300_300_377, #200_200_377, #100_100_377, #000_000_377, // 0
    #040_040_300, #070_140_300, #070_110_260, #100_170_260, // 4
    #120_260_260, #150_277_240, #120_310_200, #120_340_200, // 8
    #120_377_200, #100_377_150, #177_377_050, #270_377_070, // 12
    #350_377_200, #350_300_200, #340_260_200, #377_260_140, // 16
#377_220_100, #377_170_100, #347_200_100, #360_100_000, // 20
#240_300_000, #100_277_000, #000_377_000, #230_350_230, // 24
#340_340_377, #377_377_377, #377_377_200, #377_377_100, // 28
#377_377_000 // 32

LET mina = a - size
LET minb = b - size

LET colourv = VEC 500
setpalette(colourv, limit, coltab, 32)

fillsurf(col_gray)

// Draw a small white square at the centre
setcolour(col_white)
drawrect(width*45/100, height*45/100,
        width*55/100, height*55/100)

// Draw the colour bar
FOR x = 0 TO width-1 DO
{ LET i = ((limit+1) * x) / width
  LET p, q = x, 6
  setcolour(colourv!i)
moveto(p, q)
drawby(0, 6)
}
updatescreen()}

FOR i = 0 TO width*height - 1 DO // Number of points to plot
{ LET vi = v!i
  LET colour = ?
  LET intercount = ?
  LET x, y, p, q = ?, ?, ?, ?

  // Periodically update the screen as the pixels are drawn
  IF i MOD 100 = 0 DO updatescreen()

  x := vi & #x1FF // 0 .. 511
  y := (vi>>9) & #x1FF // 0 .. 511

  // Calculate c = p + iq corresponding to pixel (x,y)
p := mina + muldiv(2*size, x, 511)
q := minb + muldiv(2*size, y, 511)
5.10. THE MANDELBROT SET

itercount := mandset(p, q, limit)
TEST itercount<0
THEN colour := col_black
ELSE colour := colourv!itercount

setcolour(colour)
drawpoint(x, y)
}

// Draw the palette of colours
FOR x = 0 TO width DO
{ LET i = (limit * x) / width
  LET p, q = x, 6
  setcolour(colourv!i)
moveto(p, q)
drawby(0, 6)
}
updatescreen()
}

The function plotset plots the requested region of the Mandelbrot set. It does this by plotting each pixel in the requested region in random order. For each point, if mandset returns -1 it is in or close to the Mandelbrot set and so is coloured black, otherwise it is given a colour depending on the number of iterations needed before $z$ is more than three units away from the origin. The palette of colours is placed in the vector colourv.

AND mandset(p, q, n) = VALOF
{ LET x, y = 0, 0 // $z = x + iy$ is initially zero
  // $c = a + ib$ is the point we are testing
  FOR i = 0 TO n DO
  { LET t = ?
    LET x3, y3 = x/3, y/3 // To avoid possible overflow
    LET rsq = muldiv(x3, x3, One) + muldiv(y3, y3, One)

    // Test whether $z$ is diverging, ie is $x^2+y^2 > 9$
    IF rsq > One RESULTIS i
    // Square $z$ and add $c$
    // Note that $(x + iy)^2 = (x^2-y^2) + i(2xy)$
    t := muldiv(2*x, y, One) + b
    x := muldiv(x, x, One) - muldiv(y, y, One) + a
    y := t
  }
// z did not diverge after n iterations
RESULTIS -1
}

This function initially sets $z = x + iy$ to zero and then repeatedly performs the assignment $z := z^2 + c$ up to $n$ times. If at any stage $z$ move further than three units from the origin, the function returns the iteration count at that moment, otherwise it returns -1 indicating that $c$ is in or close to the Mandelbrot set.

The following three diagrams show the result of running this program with a first argument of 0, 2 and 4, respectively, using an appropriate iteration limit for each.
These images were saved using the shell command `gnome-screenshot -i` and converted to `.jpg` format using `gimp`. If these commands are not yet installed on your machine type the following.

```bash
sudo apt-get install gnome-screenshot
sudo apt-get install gimp
```
This program uses only 8 digits of precision after the decimal point and this limits the detail that can be displayed when really small regions are selected. By selecting \( s = 6 \) or 7 you will see that 8 decimal digits of precision is insufficient for this level of detail. If the program were rewritten in C, the calculation could easily be done using double length floating point numbers giving a precision of about 15 decimal digits. If one unit corresponds to a distance of a metre, we would be able to display regions as small as, say, a hydrogen atom (which has a diameter of about \( 10^{-10} \) metres. However the iteration limit would have to be increased somewhat. We could, of course, go for much higher precision using the mechanism used is Section 4.32. By doing this, it would be possible to explore much tinier regions of the Mandelbrot set that have never been seen before by anyone. A high definition version of this program called \texttt{raspi/hdmandset.b} is available. Its first argument \( s \) selects different magnifications of an interesting point in the Mandelbrot set. The image displayed is square with a side length of \( 10^{-s} \). Currently \( s \) can be between 1 and 20 but this range can easily be extended. The program currently uses 40 decimal digits after the decimal point which is certainly sufficient for all settings of \( s \) from 1 to 20. Since the images require huge amounts of computation, it is best to run the program using the native
5.10. THE MANDELBROT SET

code version of BCPL. The following two images were generated in a Pentium based laptop machine running Linux using the following shell command sequence.

cd $BCPLROOT/../natbcpl
make -f MakefileSDL clean
make -f MakefileSDL hdmandset
./hdmandset 10
./hdmandset 15

To do this on the Raspberry Pi just replace MakefileSDL by MakefileRaspiSDL. The first image is as follows.

![Mandelbrot Set](image)

The above image is a detailed display of a square region with a side length of $10^{-10}$ close to the point $c = -0.53 + 0.66i$. If one unit corresponds to one metre, the side length of this image is one Angstrom which is about the size of a hydrogen atom.

It is tempting to think of the black area as land surrounded by sea coloured to indicate its depth. Indeed, the colours have been chosen so that sea close to the coast is yellow indicating sand, then there is white representing breaking
waves and foam followed by various shades to green. At a greater distance the sea is dark blue becoming lighter as the distance increases. In between other colours are used to make the image more interesting. But thinking of this image as land surrounded by sea in unrealistic since, at this magnification, the image is one Angstrom across and a single water molecule which has a diameter of 3.2 Angstoms far too large to fit in the window.

The next image increases the magnification by a factor of 100,000 giving a detailed display of a region about the size of a proton (the nucleus of a hydrogen atom).

These images help to confirm that the boundary of the Mandelbrot set remains just as wiggly at whatever magnification we use. It also helps to confirm that the Mandelbrot set is simply connected, that is between any two points in the set there is a path lying entirely in the set that joining them.

Since computing these images take considerable time, I include thumbnail pictures of the 20 images corresponding to \( s=1 \) to \( 20 \). These images are all centred at \( c = -0.529999999998948805 + 0.66501088950000000000000000000000000000i \).

They are also available as files with names from hdmandset01.jpg to
This is a simple game in which the user can hit three coloured balls with a bat in an enclosed room containing a bucket placed near the ceiling. The balls bounce off each other, the walls, the floor, the ceiling and the bat, and feel the effect of gravity. The bat can only move horizontally along the floor and its motion is controlled by the left and right arrow keys. Pressing R puts all three balls in the bucket and pressing S starts the game by removing the base of the bucket until all the balls fall out. Pressing P pauses the game, and Q terminates the game. Pressing H will toggle the display of some help information, and pressing D or U causes debugging and CPU usage information to be displayed. Pressing B toggles between the user having control of the bat or the computer moving the
bat randomly. The aim of the game is to return the balls to the bucket as quickly as possible. A typical screen shot is the following.

The source of the program is in `bplprogs/raspi/bucket.b` and, although quite long, most of it is easy to understand. There is code to display the static parts of the scene, namely, the bucket walls with their rounded ends and the base of the bucket. There is code to display the three balls and the bat in their current positions. There is code to deal with bouncing of the balls off each other and the bat as well as bounces off fixed surfaces such as the walls and the bucket. The game is controlled by input from the keyboard, handled by the function `processevents`. The program starts as follows.

```c
/* This is a simple bat and ball game

Implemented by Martin Richards (c) February 2013

History:

17/02/2013
Successfully reimplemented the first version, bucket0.b, to make it much more efficient.
*/

SECTION "sdllib"
GET "libhdr"
GET "sdl.h"
GET "sdl.b"  // Insert the library source code
.
SECTION "bucket"
```
GET "libhdr"
GET "sdl.h"

MANIFEST {
One = 1_00000 // The constant 1.000 scaled with 5 decimal
// digits after the decimal point.
OneK = 1000_00000

batradius = 12_00000
ballradius = 25_00000
endradius = 15_00000
bucketthickness = 2 * endradius

ag = 50_00000 // Gravity acceleration
}

GLOBAL {
  done:ug

  help // Display help information
  stepping // =FALSE if not stepping
  starting // Trap door open
  started
  finished
  randombat // If TRUE the bat is given random accelerations
  randbattime
  randbatx

  starttime // Set when starting becomes FALSE
  displaytime // Time to display
  usage
  displayusage
  debugging

  sps // Steps per second, adjusted automatically

  bucketwallsurf // Surface for the bucket walls
  bucketbasesurf // Surface for the bucket base
  ball1surf // Surfaces for the three balls
  ball2surf
  ball3surf
  batsurf // Surface for the bat

  backcolour // Background colour
  bucketcolour
The first few lines insert the BCPL interface with the SDL library. This is followed by the declarations of the constants and global variables used in the program. Many quantities in this program use scaled numbers with 5 decimal digits after the decimal point. These numbers are used for the location of the fixed surfaces on the screen, the centre of gravity of the balls and bat, and their velocities and accelerations. The constant One represents 1.00000 in this representation. The radii of the balls and bat are held in ballradius and batradius. The bucket has circular corners whose radius is in endradius. The thickness of the bucket walls and the base is twice endradius and is held in bucketthickness.
5.11. BALL AND BUCKET GAME

The balls feel the effect of gravity whose acceleration is held in \( \mathbf{ag} \), typically set to 50.00000 representing 50 pixels per second per second.

The player can terminate the program by pressing Q or clicking on the little cross at the top right hand corner of the window. This sets the variable done to TRUE.

Various variables, such as starting, started and finished, describe the state of the game. For instance, starting=TRUE after the player presses \( \text{S} \) to place the balls in the bucket and remove its base allowing them to fall out. When the bucket becomes empty the base is re-instated and started becomes TRUE. This is the moment when the timer starts and begins to be displayed. When all three balls are returned to the bucket, finished is set to TRUE and the timer is stopped.

Pressing \( \text{B} \) causes the program to move the bat randomly causing the balls to be eventually returned to the bucket. It is implemented using the variables randombat, randbattime and randbatx. Details are given later.

Pressing \( \text{P} \) causes the program to pause. It is implemented by setting stepping to FALSE. Pressing \( \text{D} \) or \( \text{U} \) turn on and off the display of some debugging information.

The colour of the various objects on the screen such as the bucket, bat and balls are held in suitably mnemonic variables such as bucketcolour and batcolour.

Many variables are initialised to hold the geometry of the objects in the game. For instance wall_lx and wall_rx hold the \( \mathbf{x} \) coordinates of the left and right wall. The \( \mathbf{y} \)-coordinates of the ceiling and floor are held in ceiling_yb and floor_yt. The \( \mathbf{x} \)-coordinate of the centre of the screen is held in screen_xc.

Variables starting bucket hold the coordinates of the surfaces of the bucket.

Global variables with names starting with \( \text{xlim} \) or \( \text{ylim} \) are used to determine efficiently whether a ball is in contact with a fixed surface such as the side of the bucket.

The position, velocity and acceleration of the balls are held in variables such as cgx1, cgy1, cgx1dot, cgy1dot, ax1 and ay1. It is important that these six values are in consecutive global locations since \( @\mathbf{cgx1} \) is sometimes used as a pointer to all six values.

The bat is constrained to move horizontally in contact with the floor, but it is convenient to represent its position and velocity using the variables batx, baty, batxdot and batydot. When the bat is being moved randomly, the variable abatx holds its current acceleration.

An important feature of the game is how the balls bounce. Bouncing off flat surfaces such as the floor or sides of the bucket is straightforward since they are all either horizontal or vertical. Details of such bounces are covered later. When a ball collides with another ball, the bat or a circular corner of the bucket, the computation is more difficult. The two functions incontact and cbounce help to deal with these collisions. incontact is defined as follows.
LET incontact(p1, p2, d) = VALOF
{ LET x1, y1 = p1!0, p1!1
  LET x2, y2 = p2!0, p2!1
  // (x1,y1) and (x2,y2) are the centres of two circles
  // The result is TRUE if these centres are less than d apart.
  LET dx, dy = x1-x2, y1-y2
  IF ABS dx > d | ABS dy > d RESULTIS FALSE
  IF muldiv(dx, dx, One) + muldiv(dy, dy, One) >
    muldiv(d, d, One) RESULTIS FALSE
  RESULTIS TRUE
}

The variable x1, y1, x2 and y2 are declared to hold the centres of the two circles, and the function returns TRUE if these circles are less than a distance d apart. The argument d is the sum of the radii of the two circles involved, and so is batradius+ballradius, endradius+ballradius, ballradius+ballradius. With the current settings d can be no larger than 50_00000. The function first checks whether the horizontal and vertical separations of the two objects are no greater than d. This is a cheap test and has the merit that the more detailed measurement of separation cannot suffer from overflow. The distance between the two centres is the length of the hypotenuse of a right angled triangle whose shorter sides have lengths dx and dy. Using Pythagorus’ theorem the square of this length is the sum of squares of dx and dy, and so we compare this sum with the square of d, dividing both sides of the relation by One=1_00000 to avoid overflow. Notice that both dx and dy are less than or equal to 50_00000 and so muldiv(dx, dx, One) + muldiv(dy, dy, One) can be no greater than twice 2500_00000 which is well within the range of 32-bit signed numbers. A bounce between these two objects can only occur if incontact returns TRUE. The effect of the collision is calculated by a call of cbounce whose definition is as follows.

AND cbounce(p1, p2, m1, m2) BE
{ // p1!0 and p1!1 are the x and y coordinates of a ball, bat or bucket end.
  // p1!2 and p1!3 are the corresponding velocities
  // p2!0 and p2!1 are the x and y coordinates of a ball.
  // p2!2 and p2!3 are the corresponding velocities
  // m1 and m2 are the masses of the two objects in arbitrary units
  // m2 = 0 if p1 is a bucket end.
  // m1=m2 if the collision is between two balls
  // m1=5 and m2=1 is for collisions between the bat and ball assuming the bat
  // has five times the mass of the ball.

  LET c = cosines(p2!0-p1!0, p2!1-p1!1) // Direction p1 to p2
  LET s = result2
5.11. BALL AND BUCKET GAME

IF m2=0 DO
  \{ // Object 1 is fixed, ie a bucket corner
  LET xdot = p2!2
  LET ydot = p2!3
  // Transform to (t,w) coordinates
  // where t is in the direction of the two centres
  LET tdot = inprod(xdot, ydot, c, s)
  LET wdot = inprod(xdot, ydot, -s, c)

  IF tdot>0 RETURN
  // Object 2 is getting closer so reverse tdot (but not wdot)
  // and transform back to world (x,y) coordinates.
  tdot := rebound(tdot) // Reverse tdot with some loss of energy
  // Transform back to real world (x,y) coordinates
  p2!2 := inprod(tdot, wdot, c, -s)
  p2!3 := inprod(tdot, wdot, s, c)

  RETURN
  \}

IF m1=m2 DO
  \{ // Objects 1 and 2 are both balls of equal mass
  // Find the velocity of the centre of gravity
  LET cgxdot = (p1!2+p2!2)/2
  LET cgydot = (p1!3+p2!3)/2
  // Calculate the velocity of object 1
  // relative to the centre of gravity
  LET rx1dot = p1!2 - cgxdot
  LET ry1dot = p1!3 - cgydot
  // Transform to (t,w) coordinates
  LET t1dot = inprod(rx1dot, ry1dot, c, s)
  LET w1dot = inprod(rx1dot, ry1dot, -s, c)

  IF t1dot<=0 RETURN
  // Reverse t1dot with some loss of energy
  t1dot := rebound(t1dot)

  // Transform back to (x,y) coordinates relative to cg
  rx1dot := inprod(t1dot, w1dot, c, -s)
  ry1dot := inprod(t1dot, w1dot, s, c)

  // Convert to world (x,y) coordinates
  p1!2 := rx1dot + cgxdot

  RETURN
  \}
\[ p1!3 := ry1dot + cgydot \]
\[ p2!2 := -rx1dot + cgxdot \]
\[ p2!3 := -ry1dot + cgydot \]

// Apply a small repulsive force between balls
\[ p1!0 := p1!0 - muldiv(0_40000, c, One) \]
\[ p1!1 := p1!1 - muldiv(0_40000, s, One) \]
\[ p2!0 := p2!0 + muldiv(0_40000, c, One) \]
\[ p2!1 := p2!1 + muldiv(0_40000, s, One) \]

RETURN

\\}

// Object 1 is the bat and object 2 is a ball
// Find the velocity of the centre of gravity
LET cgxdot = (p1!2*m1+p2!2*m2)/(m1+m2)
LET cgydot = (p1!3*m1+p2!3*m2)/(m1+m2)
// Calculate the velocities of the two objects
// relative to the centre of gravity
LET rx1dot = p1!2 - cgxdot
LET ry1dot = p1!3 - cgydot
LET rx2dot = p2!2 - cgxdot
LET ry2dot = p2!3 - cgydot
// Transform to (t,w) coordinates
LET t1dot = inprod(rx1dot,ry1dot, c,s)
LET w1dot = inprod(rx1dot,ry1dot, -s,c)
LET t2dot = inprod(rx2dot,ry2dot, c,s)
LET w2dot = inprod(rx2dot,ry2dot, -s,c)

IF t1dot<=0 RETURN

// Reverse t1dot and t2dot with some loss of energy
t1dot := rebound(t1dot)
t2dot := rebound(t2dot)

// Transform back to (x,y) coordinates relative to cg
rx1dot := inprod(t1dot,w1dot, c,-s)
ry1dot := inprod(t1dot,w1dot, s, c)
rx2dot := inprod(t2dot,w2dot, c,-s)
ry2dot := inprod(t2dot,w2dot, s, c)

// Convert to world (x,y) coordinates
p1!2 := rx1dot + cgxdot
p1!3 := ry1dot + cgydot // The bat cannot move vertically
p2!2 := rx2dot + cgxdot
5.11. BALL AND BUCKET GAME

\[
p_2!3 := r_2ydot + cgydot
\]

// Apply a small repulsive force
\[
p_1!0 := p_1!0 - \text{muldiv}(0.05, c, \text{One})
\]
\[
p_1!1 := p_1!1 - \text{muldiv}(0.05, s, \text{One})
\]
\[
p_2!0 := p_2!0 + \text{muldiv}(0.05, c, \text{One})
\]
\[
p_2!1 := p_2!1 + \text{muldiv}(0.05, s, \text{One})
\]

RETURN

This function may look complicated but is, in fact, quite easy to understand. It takes four arguments. The first, \(p_1\) is a pointer to the locations holding the \((x, y)\) coordinates and velocity of the first object involved in the collision, and \(p_2\) points to the coordinates and velocity of the second object. Pointers are used since \texttt{cbounce} may need to update both the position and velocity of each object after the collision. The masses of the two objects are given in arbitrary units in \(m_1\) and \(m_2\). If object 1 is a bucket corner it is given infinite mass by setting \(m_1=1\) and \(m_2=0\). If the collision is between two balls, they are given equal mass by setting \(m_1=1\) and \(m_2=1\), and if object 1 is the bat and object 2 is a ball, \(m_1\) is set to 5 and \(m_2\) is set to 1, indicating that the mass of the bat is five times that of a ball.

The direction from the centre of object 1 to the centre of object 2 is calculated by a call of \texttt{cosines} whose arguments are the horizontal and vertical displacements between the two centres. On return, the result is the cosine of the direction relative to the \(x\) axis, and \texttt{result2} holds the corresponding sine. The implementation of \texttt{cosines} is described later.

When object 1 is a bucket corner, the calculation is simple since the corner is fixed and the ball’s velocity in the direction of the centres is reversed with some energy loss. This velocity is calculated using the direction cosines by the call \texttt{inprod(xdot, ydot, c, s)}. The transverse velocity (orthogonal to the line between the centres) is calculated by the call \texttt{inprod(xdot, ydot, -s, c)}. The results are placed in \(tdot\) and \(wdot\), respectively. If the ball is approaching the corner \(tdot\) will be negative, a bounce will take place implemented by replacing \(tdot\) with the result of \texttt{rebound(tdot)}. The inverse transformation is performed to convert the velocities back to world \((x, y)\) coordinates.

The case when \(m_1=m_2\) is two balls of equal mass collide and its implementation is a straightforward optimisation of the general case given at the end of \texttt{cbounce} that deals with objects with different masses. We will look at this general case first.

The principles underlying this kind of collision were worked out by Isaac Newton and described in 1687 in \textit{Principia Mathematica}. His second law states that
the acceleration $\mathbf{a}$ of a body is parallel and directly proportional to the net force $\mathbf{F}$ acting on the body, is in the direction of the force and is inversely proportional to the mass $m$ of the body, i.e. $\mathbf{F} = m\mathbf{a}$. Note that we are using the standard mathematical convention that quantities that have both magnitude and direction, such as $\mathbf{F}$ and $\mathbf{a}$ appear in bold while those such a $m$ that only have magnitude are non bold.

Suppose $\mathbf{F}$ and $m$ are such as to cause an acceleration of one foot per second per second, then applying the force for one second would increase the speed of the body by one foot per second. Applying it for two seconds would increase the speed by two feet per second. Thus if $t$ was the length of time the force was applied and $\mathbf{v}$ was the resulting change in velocity then $\mathbf{F}t = m\mathbf{v}$. The term $\mathbf{F}t$ is called the *impulse*, and $m\mathbf{v}$ is called the change in *momentum*. When two bodies collide they receive equal and opposite impulses so their changes in momentum are equal and opposite. The total momentum of two colliding bodies is thus unchanged by the collision. It is easy to see that the velocity of the combined centre of gravity of two objects in unaffected by the collision.

We calculate the velocity of the combined centre of gravity by declaring $\mathbf{cgxdot}$ to have value $(p1!2*m1+p2!2*m2)/(m1+m2)$ and $\mathbf{cgydot}$ to have value $(p1!3*m1+p2!3*m2)/(m1+m2)$. We then subtract this velocity from the velocities of the two objects, declaring $\mathbf{rx1dot}$, $\mathbf{ry1dot}$, $\mathbf{rx2dot}$ and $\mathbf{ry2dot}$ to be the velocities of the two object relative to the centre of gravity. Even though we are now in a moving frame of reference the behaviour of the objects are unchanged. After all, if you play billiards or snooker the behaviour of the balls is not affected by the fact we are travelling at a more or less uniform rate of 15 miles per second around the sun, and further more, if you play again on the same table six months later when we are on the other side of the sun, even though we are now traveling at 15 miles per second in the opposite direction.

Viewing the situation relative to the centre of gravity is a great simplification, since the centre of gravity now appears to be stationary, and the two objects are moving toward the centre of gravity until they bounce, when the will then begin moving away. At the moment of collision the each receive impulses that are equal and opposite along the line joining their centres. If there is some loss of energy during the collision the component of velocity in the direction between the centre will be reversed with its magnitude slightly reduced. We assume that the component orthogonal to this direction will be unchanged. If we call these two directions $\mathbf{t}$ and $\mathbf{w}$, we can compute the velocity component of object 1 in direction $\mathbf{t}$ by evaluating $\text{inprod}(\mathbf{rx1dot}, \mathbf{ry1dot}, \mathbf{c}, \mathbf{s})$, calling the result $\mathbf{t1dot}$. The component orthogonal to this in computed by $\text{inprod}(\mathbf{rx1dot}, \mathbf{ry1dot}, -\mathbf{s}, \mathbf{c})$ and given the name $\mathbf{w1dot}$. The velocity components of the other object are computed similarly and given names $\mathbf{t2dot}$ and $\mathbf{w2dot}$. At the moment of collision the components in direction $\mathbf{t}$ are reversed using calls of $\text{rebound}$ which also simulates a slight loss in energy. The inverse transformation is then performed to obtain the velocities after the collision of the two objects relative the centre of gravity, and finally
the velocities in real world coordinates are obtained by adding the velocity of
the centre of gravity to each object. The results are the assigned to the velocity
components pointed to by \( p1 \) and \( p2 \). To make the packing of the balls in the
bucket realistic, a small repulsive force is applied to both objects when they are
in contact.

As stated earlier, the case when two balls collide (\( m1=m2 \)) is an optimisation
of this code taking advantage that the masses of the two balls are the same.

Whenever a ball bounces it loses some energy and this loss is implemented by
the function \texttt{rebound}, defined below.

\[
\text{AND rebound(vel)} = \frac{\text{vel}}{7} - \text{vel} // \text{Returns the rebound speed of a bounce}
\]

It negates the given velocity and reduces its magnitude slightly. The implementa-
tion does this by subtracting one seventh to avoid possible overflow.

When a ball collides with another ball, the bat or a round corner of the bucket,
it is necessary to calculate the direction of the line joining the centres of the two
objects. This direction could be represented by the angle between this line and
the \( x \)-axis, but it is more convenient to represent it as the cosine and sine of this
angle. These two values are often called direction cosines, and can be thought of
as the coordinates of a point at the required angle on a unit circle. The function
\texttt{cosines} computes them from given displacements \( dx \) and \( dy \) of the two centres
in the \( x \) and \( y \) directions. This calculation could have been done by taking the
inverse tangent of \( dy/dx \) and then computing the cosine and sine of the resulting
angle, but for this program an alternative method is used.

If you think of a right angled triangle whose two shorter sides are of length \( dx \)
and \( dy \) lying parallel the \( x \) and \( y \)-axes, by Pythagoras’ theorem the hypotenuse
will be of length \( \sqrt{dx^2 + dy^2} \), and so the required cosine and sine will be
\[
\frac{dx}{\sqrt{(dx^2 + dy^2)}} \quad \text{and} \quad \frac{dy}{\sqrt{(dx^2 + dy^2)}}.
\]
The function \texttt{cosines}, defined below, first reduces the
size of the triangle by dividing \( dx \) and \( dy \) by the so called Manhatten distance
\( \text{ABS } dx + \text{ABS } dy \). This will cause the hypotenuse to have a length somewhere
between about 0.7 and 1. The square of this length is placed in \( a \) and the
approximate values of cosine and sine are held in \( c \) and \( s \). To correct these values
they must be divided by the square root of \( a \) which is computed to sufficient
precision by just three iterations of Newton-Raphson using a well chosen initial
guess. The Newton-Raphson iteration is illustrated by the following diagram.
The iteration is based on the function \( f(x) = x^2 - a \) which has the property that \( x = \sqrt{a} \) when \( f(x) = 0 \). As shown in Section 5.6, the slope of \( f(x) \) is its differential which, in this case, is \( 2x \). To find a value of \( d \) for which \( f(d) = 0 \) we can make a guess, say \( d = 1 \), corresponding to point A in the diagram, and improve it by reducing \( d \) by \( f(d) \) divided by the slope of \( f(x) \) at \( x = d \). The new guess is then \( d - \left( d^2 - a \right) / 2d \) which simplifies to \( (d + a)/2 \). This step is encoded by the statement \( d := (d + \text{muldiv}(d^2, 1, d))/2 \). The new value of \( d \) corresponds to point B in the diagram. If you uncomment the \texttt{writef} statements you will see how rapidly this process converges. In fact, each iteration approximately doubles the number of significant digits, so if we started with a guess that was correct to one significant place, the successive iterations would be correct to about 2, 4, 8 and 16 places. Indeed, if we did the calculation to sufficient precision, 10 iterations would give us an answer correct to about 1000 places. However, for our purposes the 4 digits of precision obtained by three iterations is sufficient. To understand this mechanism in more detail, do a web search on \texttt{newton raphson}.

The definition of \texttt{cosines} is as follows.

\begin{verbatim}
AND cosines(dx, dy) = VALOF
{ LET d = ABS dx + ABS dy
  LET c = muldiv(dx, One, d) // Approximate cos and sin
  LET s = muldiv(dy, One, d) // Direction good, length not.
  LET a = muldiv(c,c,One)+muldiv(s,s,One) // 0.5 <= a <= 1.0
  d := 1.00000 // With this initial guess only 3 iterations
  // of Newton-Raphson are required.
}
\end{verbatim}
5.11. BALL AND BUCKET GAME

The cosine is returned as the result of cosines and the sine is returned in the global result2.

If a point has coordinates \((x, y)\) then its component in the direction specified by cosines \((c, s)\) is \(xc + ys\). This value is sometimes called the inner product of the two pairs \((x, y)\) and \((c, s)\). For our scaled numbers with 5 digits after the decimal point, this calculation and be performed by calling inprod\((x, y, c, s)\).

The definition of inprod is as follows.

\[
\text{AND inprod}(dx, dy, c, s) = \text{muldiv}(dx, c, \text{One}) + \text{muldiv}(dy, s, \text{One})
\]

As the game proceeds, the window is repeatedly redrawn perhaps more often as 20 times per second to give the illusion that the bat and balls are moving smoothly. The function step is used to calculate the new the positions of the bat and balls for each image frame. This function uses ballbounces to deal with bounces between balls and the bat or fixed surfaces such as the walls or bucket. Most of ballbounces is easy to understand, but since it is rather long it will be described a few lines at a time. It starts as follows.

\[
\text{AND ballbounces}(pv) \text{ BE}
\]

\[
\{ // This function deals with bounces between the ball whose position
// and velocity is specified by pv and the bat or any fixed surface.
// It does not deal with ball on ball bounces.
\text{LET cx, cy, vx, vy = pv!0, pv!1, pv!2, pv!3}
\text{TEST x1lim_bucket_ll \leq cx \leq x1lim_bucket_rr &}
\text{ ylim_baseb \leq cy \leq ylim_topt}
\text{THEN \{ // The ball cannot be in contact with the cieling, floor or}
// either wall so we only need to check for contact with
// the bucket
\]
The argument $pv$ points to consecutive locations holding the $(x, y)$ coordinates of a ball and its velocities in the $x$ and $y$ directions. These are extracted and placed in the variables $cx$, $cy$, $vx$ and $vy$. The TEST command then determines whether the ball might bounce off the bucket or the walls. The THEN case deals with possible bounces off the bucket.

```c
IF cy > bucket_tyc DO
{ LET ecx, ecy, evx, evy = bucket_lxc, bucket_tyc, 0, 0
  IF incontact(@ecx, pv, endradius+ballradius) DO
  { cbounce(@ecx, pv, 1, 0)
    // No other bounces possible
    RETURN
  }
  ecx := bucket_rxc
  IF incontact(@ecx, pv, endradius+ballradius) DO
  { cbounce(@ecx, pv, 1, 0)
    // No other bounces possible
    RETURN
  }
  // No other bounces possible
  RETURN
}

If $cy$ is greater $bucket_tyc$, the only possible bounces are with the two rounded tops of each side of the bucket. These are tested for and dealt with using appropriate calls of `incontact` and `cbounce`.

```c
IF cy >= bucket_byc DO
{ // Possibly bouncing with bucket walls

  IF cx <= bucket_lxc DO
  { // Bounce with outside of bucket left wall
    pv!0 := xlim_bucket_ll
    IF vx>0 DO pv!2 := rebound(vx)
  }
  IF bucket_lxc < cx <= xlim_bucket_lr DO
  { // Bounce with inside of bucket left wall
    pv!0 := xlim_bucket_lr
    IF vx<0 DO pv!2 := rebound(vx)
  }
  IF xlim_bucket_rl <= cx < bucket_rxc DO
  { // Bounce with inside of bucket right wall
    pv!0 := xlim_bucket_rl
    IF vx>0 DO pv!2 := rebound(vx)
  }

  IF bucket_rxc < cx <= bucket_rxc DO
  { // Bounce with outside of bucket right wall
    pv!0 := xlim_bucket_rr
    IF vx<0 DO pv!2 := rebound(vx)
  }
}
```

```
If the boundary conditions for bouncing are satisfied, the only possible bounces are with the inside or outside of the bucket walls. These four possibilities are straightforward and dealt with in turn.

// Bounce with base
UNLESS starting DO
{ // The bucket base is present
  IF bucket_lxc <= cx <= bucket_rxc DO
    { // Bounce on the outside of the base
      pv!1 := ylim_baseb
      IF vy>0 DO pv!3 := rebound(vy)
      // No other bounces are possible
      RETURN
    }
  IF bucket_byc <= cy <= ylim_baset DO
    { // Bounce on the top of the base
      pv!1 := ylim_baset
      IF vy<0 DO pv!3 := rebound(vy)
      // No other bounces are possible
      RETURN
    }
}

If starting is FALSE the base of the bucket is present, and so bouncing is possible of its top or bottom surfaces. The above code deals with these two cases. If either bounce occurs no other bounces are possible, so the function returns.

// Bounces with the bottom corners
IF cy < bucket_byc DO
{ LET ecx, ecy, evx, evy = bucket_lxc, bucket_byc, 0, 0
  IF incontact(@ecx, pv, endradius+ballradius) DO
    { // Bounce with bottom left corner
      cbounce(@ecx, pv, 1, 0)
    }
// No other bounces are possible
RETURN
}
ecx := bucket_rxc
IF incontact(@ecx, pv, endradius+ballradius) DO
{ // Bounce with bottom right corner
cbounce(@ecx, pv, 1, 0)
// No other bounces are possible
RETURN
}

The above code deals with bounces off the bottom two corners of the bucket, but is only reached if the ball did not bounce off the bucket base, if present. As before, these corner bounces are easy to implement using suitable calls of \texttt{incontact} and \texttt{cbounce}.

The rest of \texttt{ballbounces} deals with bounces known not to be off the bucket, and since ball on ball bounces are not performed by \texttt{ballbounces} the only possibilities are with the bat, wall, ceiling or floor. The following code deals with them all.

ELSE { // The ball can only be in contact with the bat, side walls, // ceiling or floor

// Bouncing with the bat
IF incontact(@batx, pv, batradius+ballradius) DO
{ pv!4, pv!5 := 0, 0
  cbounce(@batx, pv, 5, 1)
  batydot := 0 // Immediately damp out the bat’s vertical motion
}

// Left wall bouncing
IF cx <= xlim_lwall DO
{ pv!0 := xlim_lwall
  IF vx<0 DO pv!2 := rebound(vx)
}

// Right wall bouncing
IF cx >= xlim_rwall DO
{ pv!0 := xlim_rwall
  IF vx>0 DO pv!2 := rebound(vx)
}
// Ceiling bouncing
IF cy >= ylim_ceiling DO
    { pv!1 := ylim_ceiling
      IF vy>0 DO pv!3 := rebound(vy)
      // No other bounces are possible
      RETURN
    }

// Floor bouncing
IF cy <= ylim_floor DO
    { pv!1 := ylim_floor
      IF vy<0 DO pv!3 := rebound(vy)
    }

    // No other bounces are possible
    RETURN

}

Notice that the above code allowed for bounces to occur simultaneously between the ball and, say, a wall and the floor.

The function step is called repeatedly to update the positions of the balls and the bat. It definition starts as follows.

LET step() BE
{ IF started UNLESS finished DO
    displaytime := sdlmsecs() - starttime

    The timer starts as soon as the bucket base is reinstated after all three balls have fallen out of the bucket. It continues measuring the time until the three balls have again settled into the bucket. The variable displaytime holds the time measured in milli-seconds since the start. It is only updated after started becomes TRUE and before finished becomes TRUE.

    The next fragment of code updates started to TRUE at the appropriate moment.

    // Check whether to close the base
    WHILE starting DO
        { IF ylim_baseb < cgy1 & bucket_lxc < cgx1 < bucket_rxc BREAK
          IF ylim_baseb < cgy2 & bucket_lxc < cgx2 < bucket_rxc BREAK
          IF ylim_baseb < cgy3 & bucket_lxc < cgx3 < bucket_rxc BREAK
          starting := FALSE
          started := TRUE
        }
finished := FALSE
starttime := sdlmsecs()
displaytime := 0
BREAK
}

This code is not really a WHILE loop since its body is not repeatedly executed. It is a trick to allow the use of BREAK to exit from this fragment of code. The first IF statement executes BREAK if the first ball is above or possibly in contact with the bucket base and has an x value between the bucket walls. The second and third IF statements perform the same test for the other two balls. If none of these tests call BREAK, the game has just started causing starting to be set to FALSE and started to TRUE. The other three variables finished, starttime and displaytime are also initialised appropriately.

The next fragment tests whether the balls have returned to the bucket.

IF started UNLESS finished DO
  IF bucket_byt < cgy1 < bucket_tyb &
    bucket_lxc < cgx1 < bucket_rxc &
    bucket_byt < cgy2 < bucket_tyb &
    bucket_lxc < cgx2 < bucket_rxc &
    bucket_byt < cgy3 < bucket_tyb &
    bucket_lxc < cgx3 < bucket_rxc &
    ABS cgy1dot < 2.00000 &
    ABS cgy2dot < 2.00000 &
    ABS cgy3dot < 2.00000 DO finished := TRUE

It checks that the centre of each ball is within the bucket and that none of them are travelling fast enough in the y direction to escape. If all these tests succeed, finished is set to TRUE.

Variables, such ax1 and ay1, hold the horizontal and vertical accelerations of the balls. They are initialised by the following code.

// Calculate the accelerations of the balls
// Initialise and apply gravity
ax1, ay1 := 0, -ag
ax2, ay2 := 0, -ag
ax3, ay3 := 0, -ag

// Add a little random horizontal motion
ax1 := ax1 + randno(2001) - 1001
ax2 := ax2 + randno(2001) - 1001
ax3 := ax3 + randno(2001) - 1001
They are each given a vertical acceleration of \(-ag\) simulating gravity and small random horizontal accelerations to stop balls being able to stand unrealistically in a vertical column.

The next fragments are concerned with the bouncing of the balls on any surface they come in contact with. The following code deals with the balls bouncing of the left and right hand walls.

```cpp
ballbounces(@cgx1)
ballbounces(@cgx2)
ballbounces(@cgx3)
```

The ball on ball bounces are dealt with by the follow code. The only subtlety is that during a bounce the force of gravity are ignored by setting, for instance, \(ay1\) and \(ay2\) to zero. Since all ball have the same mass \(m1\) and \(m2\) are both given value 1.

```cpp
// Ball on ball bounce
IF incontact(@cgx1, @cgx2, ballradius+ballradius) DO
{ ay1, ay2 := 0, 0
  cbounce(@cgx1, @cgx2, 1, 1)
}

IF incontact(@cgx1, @cgx3, ballradius+ballradius) DO
{ ay1, ay3 := 0, 0
  cbounce(@cgx1, @cgx3, 1, 1)
}

IF incontact(@cgx2, @cgx3, ballradius+ballradius) DO
{ ay2, ay3 := 0, 0
  cbounce(@cgx2, @cgx3, 1, 1)
}
```

Then follows code to updates the velocities of the three balls and their positions.

```cpp
// Apply forces to the balls
cgx1dot := cgx1dot + ax1/Sps
cgy1dot := cgy1dot + ay1/Sps
cgx2dot := cgx2dot + ax2/Sps
cgy2dot := cgy2dot + ay2/Sps
cgx3dot := cgx3dot + ax3/Sps
cgy3dot := cgy3dot + ay3/Sps

cgx1, cgy1 := cgx1 + cgx1dot/Sps, cgy1 + cgy1dot/Sps
cgx2, cgy2 := cgx2 + cgx2dot/Sps, cgy2 + cgy2dot/Sps
cgx3, cgy3 := cgx3 + cgx3dot/Sps, cgy3 + cgy3dot/Sps
```
If B is pressed the bat moves randomly. This is implemented by setting `randombat` to `TRUE`, and then selecting a new target x position for the bat every half second. The bat always accelerates to this target. The selected target is either related to the position of the lowest ball, or is randomly chosen. The speed of the bat is limited to no more than 400 pixels per second. If the bat hits a wall it bounces without loss of energy. The y position of the bat is also given a slight correction.

```c
IF randombat DO
{ LET t = sdlmsecs()
  IF t > randbattime + 0.500 DO
    { // Choose a new random target x position every 1/10 second
      LET xmax = screenxsize*One
      randbatx := randno(xmax)
      IF randno(1000)<500 DO
        { // About 50% of the time choose as target the x position
          // depending on the position of the lowest ball to the bat.
          LET miny = cgy1
          randbatx := cgx1
          IF cgy2<miny D0 randbatx, miny := cgx2, cgy2
          IF cgy3<miny D0 randbatx, miny := cgx3, cgy3
        }
      randbattime := t
    }
    TEST batx > randbatx THEN abatx := -500_00000
      ELSE abatx := 500_00000
  }
// Apply forces to the bat
batxdot := batxdot + abatx/sps
IF batxdot> 600_00000 D0 batxdot := 600_00000
IF batxdot<-600_00000 D0 batxdot := -600_00000
batx := batx + batxdot/sps

IF batx+batradius > wall_rx D0
{ batx := wall_rx - batradius
  batxdot := -batxdot
}
IF batx-batradius < 0 D0
{ batx := batradius
  batxdot := -batxdot
}

// Slowly correct baty
```
5.11. BALL AND BUCKET GAME

\[
\text{baty} := \text{baty} - (\text{baty} - \text{batradius})/10
\]

In the first iteration of this program the bucket with or without its base, the balls and the bat were all drawn from scratch each time a new frame was displayed. This turned out to be too inefficient for the Raspberry Pi and so a more efficient implementation was chosen. This involved creating small SDL surfaces containing fragments of the scene which could be copied to the screen efficiently when needed. The fragments chosen were a wall of the bucket with its rounded ends, the three coloured balls and the bat. The corresponding surfaces are held in \text{bucketwallsurf}, \text{bucketbasesurf}, \text{ball1surf}, \text{ball2surf}, \text{ball3surf} and \text{batsurf}. They are created when needed by functions such as \text{initbucketwallsurf} defined below.

\[
\text{AND initbucketwallsurf()} = \text{VALOF}
\]
\[
\{ // Allocate a surface for the bucket walls
\LET \text{width} = 2\cdot\text{endradius}/\text{One} + 1
\LET \text{height} = (\text{bucket\_tyt} - \text{bucket\_byb})/\text{One} + 2
\LET \text{surf} = \text{mksurface}(\text{width}, \text{height})
\}
\]

It first calculates the width and height of the fragment, and creates a surface of the size. It fills the surface with the background colour and then draws the rounded ends of the bucket wall by suitable calls of \text{drawfillcircle}. The wall itself is then drawn by a call of \text{drawfillrect}. Notice that when \text{debugging} is \text{TRUE} the circular bucket ends are given a different colour.

The coding of the other initialisation functions follow the same pattern. They are defined as follows.
AND initbucketbasesurf(col) = VALOF
{ // Allocate the bucket base surface
  LET height = 2*endradius/One + 1
  LET width = (bucket_rxc - bucket_lxc)/One + 1
  LET surf = mksurface(width, height)

  selectsurface(surf, width, height)
  fillsurf(backcolour)
  setcolour(bucketcolour)
  drawfillrect(0, 0, width, height)
  RESULTIS surf
}

AND initballsurf(col) = VALOF
{ // Allocate a ball surface
  LET height = 2*ballradius/One + 2
  LET width = height
  LET colkey = maprgb(64,64,64)
  LET surf = mksurface(width, height)

  selectsurface(surf, width, height)
  fillsurf(colkey)
  setcolourkey(surf, colkey)

  setcolour(col)
  drawfillcircle(ballradius/One, ballradius/One+1, ballradius/One)

  RESULTIS surf
}

AND initbatsurf(col) = VALOF
{ // Allocate a bat surface
  LET height = 2*batradius/One + 2
  LET width = height
  LET surf = mksurface(width, height)
  selectsurface(surf, width, height)
  fillsurf(backcolour)

  setcolour(batcolour)
  drawfillcircle(batradius/One, batradius/One+1, batradius/One)

  RESULTIS surf
}

The only subtlety is in the function initballsurf which uses a feature called
5.11. BALL AND BUCKET GAME  

colour keying to cause only the circular ball to be written to the screen. The 
pixels outside the circle are given a special colour held in colkey and the call 
setcolourkey(surf,colkey) tells the surface not to copy any pixels of this 
colour to the screen. If you comment out the call of setcolourkey you will see 
why this call is necessary.

The next function plotscreen draws the entire scene. It first fills in the 
background colour and then checks all the required surface fragments have been 
created. It then copies the to the screen by calls of blitsurf. This function 
takes four arguments src, dst, x and y, where src and dst are the source and 
destination surfaces, and (x,y) is the position of where the 

AND plotscreen() BE 
{ selectsurface(screen, screenxsize, screenysize) 
  fillsurf(backcolour)

  // Allocate the surfaces if necessary
  UNLESS bucketwallsurf DO bucketwallsurf := initbucketwallsurf()
  UNLESS starting | 
    bucketbasesurf DO bucketbasesurf := initbucketbasesurf()
  UNLESS ball1surf DO ball1surf := initballsurf(ball1colour)
  UNLESS ball2surf DO ball2surf := initballsurf(ball2colour)
  UNLESS ball3surf DO ball3surf := initballsurf(ball3colour)
  UNLESS batsurf DO batsurf := initbatsurf(batcolour)

  // Left bucket wall
  blitsurf(bucketwallsurf, screen, bucket_lxl/One, bucket_tyt/One)
  // Right bucket wall
  blitsurf(bucketwallsurf, screen, bucket_rxl/One, bucket_tyt/One)

  IF bucketbasesurf DO 
    blitsurf(bucketbasesurf, screen, bucket_lxc/One, bucket_byt/One-1)

  // The bat
  blitsurf(batsurf, screen, (batx-batradius)/One, (baty+batradius)/One)

  IF debugging & randombat DO 
  { setcolour(bucketcolour)
    drawfillcircle(randbatx/One, baty/One, 7)
  }

  // Finally, the three balls
  blitsurf(ball1surf, screen, (cgx1-ballradius)/One, (cgy1+ballradius)/One)
  blitsurf(ball2surf, screen, (cgx2-ballradius)/One, (cgy2+ballradius)/One)
This draws the bucket (with or without its base) the three coloured balls and
the bat. All that remains is to write some text on the screen. This is done by
the following code.

```text
setcolour(maprgb(255,255,255))

IF finished DO
    plotf(30, 300, "Finished -- Well Done!")
ENDIF

IF started | finished DO
    plotf(30, 280, "Time %9.2d", displaytime/10)
ENDIF

IF help DO
    \{ plotf(30, 150, "R -- Reset")
    plotf(30, 135, "S -- Start the game")
    plotf(30, 120, "P -- Pause/Continue")
    plotf(30, 105, "H -- Toggle help information")
    plotf(30, 90, "B -- Toggle bat random motion")
    plotf(30, 75, "D -- Toggle debugging")
    plotf(30, 60, "U -- Toggle usage")
    \}
    plotf(30, 45, "Left/Right arrow -- Control the bat")
ENDIF

IF displayusage DO
    plotf(30, 245, "CPU usage = %i3%% sps = %n", usage, sps)
ENDIF

IF debugging DO
    \{ plotf(30, 220, "Ball1 x=%10.5d y=%10.5d xdot=%10.5d ydot=%10.5d", cgx1, cgy1, cgx1dot, cgy1dot)
    plotf(30, 205, "Ball2 x=%10.5d y=%10.5d xdot=%10.5d ydot=%10.5d", cgx2, cgy2, cgx2dot, cgy2dot)
    plotf(30, 190, "Ball3 x=%10.5d y=%10.5d xdot=%10.5d ydot=%10.5d", cgx3, cgy3, cgx3dot, cgy3dot)
    plotf(30, 175, "Bat  x=%10.5d y=%10.5d xdot=%10.5d", batx, baty, batxdot)
    \}
ENDIF
```

This code uses `plotf` to write text to specified positions on the screen but
otherwise should be self explanatory.

The next function initialises the position and velocity of the balls and a few
other variables. It definition is as follows.
AND resetballs() BE
{ cgy1 := bucket_byt+ballradius + 10_00000
  cgy2 := bucket_byt+3*ballradius + 20_00000
  cgy3 := bucket_byt+5*ballradius + 30_00000
  cgx1, cgx2, cgx3 := screen_xc, screen_xc, screen_xc
  cgx1dot, cgx2dot, cgx3dot := 0, 0, 0
  cgy1dot, cgy2dot, cgy3dot := 0, 0, 0

  starting := FALSE
  started := FALSE
  finished := FALSE
  displaytime := -1
}

The function proccesevents deals with input from the mouse and keyboard. Most keyboard events are simple letters detected when the key is pressed. These are all easily understood. The only subtlety is the treatment of the left and right arrow keys. An acceleration of 750_00000 is added to abatx while the right arrow key is held down. When it is eventually raised 750_00000 is decremented from abatx. Thus while the right arrow key is pressed the bat accelerates at a constant rate to the right. Similarly, the left arrow key accelerates the bat to the left.

AND proccesevents() BE WHILE getevent() SWITCHON eventtype INTO
{ DEFAULT:
  LOOP

  CASE sdle_keydown:
    SWITCHON capitalch(eventa2) INTO
    { DEFAULT: LOOP

      CASE 'Q': done := TRUE
        LOOP

      CASE '?':
      CASE 'H': help := ~help
        LOOP

      CASE 'D': debugging := ~debugging
        IF bucketwallsurf DO
        { freesurface(bucketwallsurf)
          bucketwallsurf := 0
        }
        LOOP

      CASE 'U': displayusage := ~displayusage

    }
  }
}

CASE sdle_keydown:
  SWITCHON capitalch(eventa2) INTO
  { DEFAULT: LOOP

    CASE 'Q': done := TRUE
      LOOP

    CASE '?':
    CASE 'H': help := ~help
      LOOP

    CASE 'D': debugging := ~debugging
      IF bucketwallsurf DO
      { freesurface(bucketwallsurf)
        bucketwallsurf := 0
      }
      LOOP

    CASE 'U': displayusage := ~displayusage
  }

  LOOP
}

CASE sdle_keydown:
  SWITCHON capitalch(eventa2) INTO
  { DEFAULT: LOOP

    CASE 'Q': done := TRUE
      LOOP

    CASE '?':
    CASE 'H': help := ~help
      LOOP

    CASE 'D': debugging := ~debugging
      IF bucketwallsurf DO
      { freesurface(bucketwallsurf)
        bucketwallsurf := 0
      }
      LOOP

    CASE 'U': displayusage := ~displayusage
  }

  LOOP
}

CASE sdle_keydown:
  SWITCHON capitalch(eventa2) INTO
  { DEFAULT: LOOP

    CASE 'Q': done := TRUE
      LOOP

    CASE '?':
    CASE 'H': help := ~help
      LOOP

    CASE 'D': debugging := ~debugging
      IF bucketwallsurf DO
      { freesurface(bucketwallsurf)
        bucketwallsurf := 0
      }
      LOOP

    CASE 'U': displayusage := ~displayusage
  }

  LOOP
}

CASE sdle_keydown:
  SWITCHON capitalch(eventa2) INTO
  { DEFAULT: LOOP

    CASE 'Q': done := TRUE
      LOOP

    CASE '?':
    CASE 'H': help := ~help
      LOOP

    CASE 'D': debugging := ~debugging
      IF bucketwallsurf DO
      { freesurface(bucketwallsurf)
        bucketwallsurf := 0
      }
      LOOP

    CASE 'U': displayusage := ~displayusage
  }

  LOOP
}
CHAPTER 5. INTERACTIVE GRAPHICS IN BCPL USING SDL

LOOP

CASE 'B': randombat := ~randombat
    abatx := 0
    randbatx := screen_xc
    randbattime := 0
    LOOP

CASE 'S': // Start again
    UNLESS ylim_baseb < cgy1 & bucket_lxc < cgx1 < bucket_rxc &
    ylim_baseb < cgy2 & bucket_lxc < cgx2 < bucket_rxc &
    ylim_baseb < cgy3 & bucket_lxc < cgx3 < bucket_rxc DO
    resetballs()
    starting := TRUE
    started := FALSE
    finished := FALSE
    starttime := -1
    displaytime := -1
    IF bucketbasesurf DO
        freesurface(bucketbasesurf)
        bucketbasesurf := 0
    END
    LOOP

CASE 'P': // Toggle stepping
    stepping := ~stepping
    LOOP

CASE 'R': // Reset the balls
    resetballs()
    finished := FALSE
    starting := FALSE
    displaytime := -1
    LOOP

CASE sdle_arrowright:
    abatx := abatx + 750_00000; LOOP
CASE sdle_arrowleft:
    abatx := abatx - 750_00000; LOOP
}

CASE sdle_keyup:
    SWITCHON capitalch(eventa2) INTO
    { DEFAULT: LOOP
Notice that the surface fragment `bucketballsurf` must be cleared when D is pressed since toggling the debugging flag causes the colour of the bucket ends to change. Similarly, `bucketbasesurf` must be cleared when S is pressed.

The final function `start` is the main program. It initialises all the required variables and then enters the event loop to repeatedly read events, update the state of the balls and bat and display the result. If you comment out the IF FALSE DO line near the top, code will run to test the `cosines` function. This was a debugging aid used to ensure the `cosines` behaved correctly.

```plaintext
LET start() = VALOF
{ LET stepmsecs = ?
  LET comptime = 0 // Amount of cpu time per frame

  UNLESS sys(Sys_sdl, sdl_avail) DO
  { writef("*nThe SDL features are not available*n")
    RESULTIS 0
  }

  bucketwallsurf := 0
  bucketbasesurf := 0
  ball1surf := 0
  ball2surf := 0
  ball3surf := 0
  batsurf := 0

  IF FALSE DO
  { // Code to test the cosines function
    LET e1, e2 = One, One
    FOR dy = 0 TO One BY One/100 DO
    { LET c, s, rsq = ?, ?, ?
      c := cosines(One, dy)
    }
s := result2
rsq := muldiv(c,c,One) + muldiv(s,s,One)
writef("dx=%9.5d  dy=%9.5d  cos=%9.5d  sin=%9.5d  rsq=%9.5d*n", One, dy, c, s, rsq)
   IF e1 < rsq D0 e1 := rsq
   IF e2 > rsq D0 e2 := rsq
}
writef("Errors +%6.5d -%7.5d*n", e1-One, One-e2)
RESULTIS 0
}

initsdl()
mkscreen("Ball and Bucket", 800, 500)

help := TRUE
randombat := FALSE
randbatx := screen_xc
randbattime := 0

stepping := TRUE    // =FALSE if not stepping
starting := TRUE    // Trap door open
started := FALSE
finished := FALSE
starttime := -1
displaytime := -1
usage := 0
debugging := FALSE
displayusage := FALSE
sps := 40    // Initial setting
stepmsecs := 1000/sps

backcolour := maprgb(120,120,120)
bucketcolour := maprgb(170, 60, 30)
bucketendcolour := maprgb(140, 30, 30)
ball1colour := maprgb(255, 0, 0)
ball2colour := maprgb(0, 255, 0)
ball3colour := maprgb(0, 0, 255)
batcolour := maprgb(40, 40, 40)

wall_lx := 0
wall_rx := (screenxsize-1)*One   // Right wall

floor_yt := 0       // Floor
ceiling_yb := (screenysize-1)*One  // Ceiling
screen_xc := screen_xsize / 2
bucket_tyt := ceiling_yb - 6 * ballradius
bucket_tyc := bucket_tyt - endradius
bucket_tyb := bucket_tyt - bucket_thickness
bucket_lxr := screen_xc - ballradius * 5 / 2
bucket_lxc := bucket_lxr - endradius
bucket_lxl := bucket_lxr - bucket_thickness
bucket_rxr := bucket_rxl + bucket_thickness
bucket_byt := bucket_tyt - 6 * ballradius
bucket_byc := bucket_byt - endradius
bucket_byb := bucket_byt - bucket_thickness

xlim_lwall := wall_lx + ballradius
xlim_rwall := wall_rx - ballradius
ylim_floor := floor_yt + ballradius
ylim_ceiling := ceiling_yb - ballradius
xlim_bucket_ll := bucket_lxl - ballradius
xlim_bucket_lc := bucket_lxc - ballradius
xlim_bucket_lr := bucket_lxr + ballradius
xlim_bucket_rl := bucket_rxl - ballradius
xlim_bucket_rc := bucket_rxc - ballradius
xlim_bucket_rr := bucket_rrx + ballradius
ylim_topt := bucket_tyt + ballradius
ylim_baseb := bucket_byb - ballradius
ylim_baset := bucket_byt + ballradius

resetballs()

ax1, ay1 := 0, 0  // Acceleration of ball 1
ax2, ay2 := 0, 0  // Acceleration of ball 2
ax3, ay3 := 0, 0  // Acceleration of ball 3

batx := screen_xc  // Position of bat
baty := floor_yt + ballradius  // Position of bat
ylim_bat := floor_yt + bat_radius + ballradius

batxdot, batyd := 150_00000, 0  // Velocity of bat
abatx := 0  // Acceleration of bat
done := FALSE

UNTIL done DO
{ LET t0 = sdlmsecs()
  LET t1 = ?

  processesets()

  IF stepping DO step()

  usage := 100*comptime/stepmsecs
  plotscreen()
  updatescreen()
  UNLESS 80<usage<95 DO
  { TEST usage>90
    THEN sps := sps-1
    ELSE sps := sps+1
    stepmsecs := 1000/sps
  }

  t1 := sdlmsecs()

  comptime := t1 - t0
  IF t0+stepmsecs > t1 DO sdldelay(t0+stepmsecs-t1)
}

writef("*nQuitting*n")
sdldelay(1_000)

IF bucketwallsurf DO freesurface(bucketwallsurf)
IF bucketbasesurf DO freesurface(bucketbasesurf)
IF ball1surf DO freesurface(ball1surf)
IF ball2surf DO freesurface(ball2surf)
IF ball3surf DO freesurface(ball3surf)
IF batsurf DO freesurface(batsurf)

closesdl()
RESULTIS 0
}

Although the Cintcode interpretive system runs this program reasonably well, you can improve its efficiency by compiling the BCPL into native machine code for the ARM processor. On the Raspberry Pi, try getting into the directory BCPL/natbcpl then typing the following.

make -f MakefileRaspiSDL clean
5.12. ROBOTS

make -f MakefileRaspiSDL bucket
./bucket

With luck this should run the bucket program with a frame rate of about 25 frames per second.

5.12 Robots

This section describes a program that displays some robots that are designed to work cooperatively collecting randomly placed bottles with their grabbers and depositing them in a pit. The dark green robot can be controlled by the user using the arrow keys, G for grab and R for release. The robots and bottles move and bounce off each other and the walls. Bottles over the pit disappear. The bottles slide over the ground without friction, but the pit is at the top of a gentle conical hill that repels bottles that get too close. As a debugging aid properties of the dark green robot and the black coloured bottle can be displayed on the screen by pressing D. At the moment, the dark green robot can be controlled by hand to grab bottles and deposit them into the pit, but the other robots do not yet have a strategy to do their job and so just move randomly bumping into each other and the walls. A typical image is the following.

This program is currently in the early stages of development and is only included here so I can conveniently read the code on my iPad. The program is called raspi/robots.b and is currently as follows.
/* This is a program that displays a number of robots designed to
pick up bottles with their grabbers and deposit them in a pit.

Implemented by Martin Richards (c) February 2015

History:

02/02/2015
Initial implementation started based on bucket.b.

*/

SECTION "sdllib"
GET "libhdr"
GET "sdl.h"
GET "sdl.b" // Insert the library source code
.
SECTION "robots"
GET "libhdr"
GET "sdl.h"

MANIFEST {
    One = 1_00000 // The constant 1.000 scaled with 5 decimal
digits after the decimal point.
    OneK = 1000_00000
    spacevupb = 100000

    pitradius = 50_00000
    bottleradius = 5_00000
    robotradius = 18_00000
    shoulderradius = 3_00000
    tipradius = 2_00000
    grablen = 12_00000
    //grabthickness = 3_00000

    // bottle selectors
    b_cgxx = 0; b_cgy
    b_cgxdot; b_cgydot
    b_costheta; b_sintheta
    b_grabbed
    b_robot // 0 or the grabbing robot
    b_dropped
    b_id // The bottle number
b_size
b_upb=b_size

// robot selectors
r_cgx=0;      r_cgy      // The first six must be in positions 0 to 5
r_cgxdot;    r_cgydot
r_costheta;  r_sintheta  // Changed every time cgxdot or cgydot changes
r_costheta;  r_sintheta  // Rotation per step
r_theta;     r_dtheta
r_reversing
r_cosphi;    r_sinphi    // Orientation of the robot
r_cosdphi;   r_sindphi   // Change in orientation per step
r_grabpos;   r_grabposdot
r_colour;    r_tipcolour
r_bottle     // =0 or the grabbed bottle
// Coords of rotated robot shoulders
r_lex;       r_ley;      r_rex;  r_rey // le  re
r_lcx;       r_lcy;      r_rcx;  r_rcy // lc  rc
// Coords of rotated robot arms
r_ltax;      r_ltay;     r_rtax;  r_rtay // ltd  ltp  ltc  rtp  rtd
r_ltx;       r_ltcy;     r_rtcx;  r_rtcy //
// r_ltdx;  r_ltdy;  r_rtdx;  r_rtdy //
// r_ltpx;  r_ltpy;  r_rtpx;  r_rtpy // lta  ltb  rtb  rta

r_bcx;       r_bcy // Centre of gripped bottle

r_id         // The robot number

r_size
r_upb=r_size
}

GLOBAL {
  done:ug

  help       // Display help information
  stepping   // =FALSE if not stepping
  starting   // Trap door open
  started
  finished

  starttime  // Set when starting becomes FALSE
  displaytime // Time to display
  usage
CHAPTER 5. INTERACTIVE GRAPHICS IN BCPL USING SDL

displayusage
displayusage

displayusage debuggin

dsps // Steps per second, adjusted automatically
bottles
bottlev
robots
robotv
// coords of the pit centre
pit_x; pit_y; pit_xdot; pit_ydot; pit_costheta; pit_sintheta
thepit // -> [pitx, pity, pit_xdot, pit_ydot, pit_costheta, pit_sintheta]
xsize // Window size in pixels
ysize
seed

spacev; spacelp; spacet
mkvec

bottlecount // Decreases as bottle fall into the pit
robotcount // Set by the -r command argument

bottlesurfR // Surface for a red bottle
bottlesurfK // Surface for a black bottle (number 1)
bottlesurfB // Surface for a brown bottle (grabbed)
pitsurf // Surface for the pit

backcolour // Background colour
col_red; col_black; col_brown
pitcolour
robotcolour
roboticolour
grabcolour

wall_lx // Left wall
wall_rx // Right wall
floor_y // Floor
ceiling_y // Ceiling

}
abort(999)
RESULTIS 0
}
//writef("mkvec (%n) => %n*n", upb, p)
RESULTIS p
}

AND mk2(a, b) = VALOF
{ LET p = mkvec(1)
  p!0, p!1 := a, b
  RESULTIS p
}

LET incontact(p1, p2, d) = VALOF
{ // This return TRUE if points p1 and p2 are less than d apart.
  LET x1, y1 = p1!0, p1!1
  LET x2, y2 = p2!0, p2!1
  // (x1,y1) and (x2,y2) are the centres of two circles
  // The result is TRUE if these centres are less than d apart.
  LET dx, dy = x1-x2, y1-y2
  IF d=pitradius & ABS dx <= d+100000 & ABS dy <= d+100000 DO
  { writef("p1=(%n,%n) p2=(%n,%n) dx=%n dy=%n d=%n*n", x1,y1, x2,y2, dx,dy, d)
  }
  IF ABS dx > d | ABS dy > d RESULTIS FALSE
  IF muldiv(dx,dx,One) + muldiv(dy,dy,One) > muldiv(d,d,One) RESULTIS FALSE
  RESULTIS TRUE
}

AND cbounce(p1, p2, m1, m2) BE
{ // p1!0 and p1!1 are the x and y coordinates of two circular object.
  // p1!2 and p1!3 are the corresponding velocities
  // p1!4 and p1!5 are the corresponding direction cosines
  // p2!0 and p2!1 are the x and y coordinates of another object.
  // p2!2 and p2!3 are the corresponding velocities
  // p2!4 and p3!5 are the corresponding direction cosines
  // m1 and m2 are the masses of the two objects in arbitrary units
  // m1=m2 if the collision is between two bottles or two robots.
  // m1=5 and m2=1 then p1 is a robot and p2 is a bottle.

  LET c = cosines(p2!0-p1!0, p2!1-p1!1) // Direction p1 to p2
  LET s = result2

  IF m2=0 DO
{ // Object 1 is a robot and object 2 is a bottle.
 // Robots are treated as infinitely heavy.
 LET xdot = p2!2 - p1!r_cgxdot
 LET ydot = p2!3 - p1!r_cgydot
 // Transform to (t,w) coordinates
 // where t is in the direction of the two centres
 LET tdot = inprod(xdot, ydot, c, s)
 LET wdot = inprod(xdot, ydot, -s, c)

 //writef("robot-bottle bounce tdot=%n wdot=%n\n", tdot, wdot)
 IF tdot>0 RETURN

 // Object 2 is getting closer so reverse tdot (but not wdot)
 // and transform back to world (x,y) coordinates.
 tdot := rebound(tdot) // Reverse tdot with some loss of energy
 // Transform back to real world (x,y) coordinates
 p2!2 := inprod(tdot, wdot, c, -s) + p1!r_cgxdot
 p2!3 := inprod(tdot, wdot, s, c) + p1!r_cgydot
 p2!4 := cosines(p2!2, p2!3)
 p2!5 := result2
 RETURN
}

IF m1=m2 DO
{ // This deals with bottle-bottle and robot-robot bounces.
 // Find the velocity of the centre of gravity
 LET cgxdot = (p1!2+p2!2)/2
 LET cgydot = (p1!3+p2!3)/2
 // Calculate the velocity of object 1
 // relative to the centre of gravity
 LET rx1dot = p1!2 - cgxdot
 LET ry1dot = p1!3 - cgydot
 // Transform to (t,w) coordinates
 LET t1dot = inprod(rx1dot, ry1dot, c, s)
 LET w1dot = inprod(rx1dot, ry1dot, -s, c)

 IF t1dot<=0 RETURN

 // Reverse t1dot with some loss of energy
 t1dot := rebound(t1dot)

 // Transform back to (x,y) coordinates relative to cg
 rx1dot := inprod(t1dot, w1dot, c, -s)
 ry1dot := inprod(t1dot, w1dot, s, c)
5.12. ROBOTS

// Convert to world (x,y) coordinates
p1!2 := rx1dot + cgxdot
p1!3 := ry1dot + cgydot
p1!4 := cosines(p1!2, p1!3)
p1!5 := result2

p2!2 := -rx1dot + cgxdot
p2!3 := -ry1dot + cgydot
p2!4 := cosines(p2!2, p2!3)
p2!5 := result2

// Apply a small repulsive force between the objects.
// This may not be necessary since there is no gravity.
p1!0 := p1!0 - muldiv(0.40000, c, One)
p1!1 := p1!1 - muldiv(0.40000, s, One)
p2!0 := p2!0 + muldiv(0.40000, c, One)
p2!1 := p2!1 + muldiv(0.40000, s, One)

RETURN
}

{ // Object 1 is a robot and object 2 is a bottle
// Find the velocity of the centre of gravity
LET cgxdot = (p1!2*m1+p2!2*m2)/(m1+m2)
LET cgydot = (p1!3*m1+p2!3*m2)/(m1+m2)
// Calculate the velocities of the two objects
// relative to the centre of gravity
LET rx1dot = p1!2 - cgxdot
LET ry1dot = p1!3 - cgydot
LET rx2dot = p2!2 - cgxdot
LET ry2dot = p2!3 - cgydot
// Transform to (t,w) coordinates
LET t1dot = inprod(rx1dot,ry1dot, c,s)
LET w1dot = inprod(rx1dot,ry1dot, -s,c)
LET t2dot = inprod(rx2dot,ry2dot, c,s)
LET w2dot = inprod(rx2dot,ry2dot, -s,c)

// IF t1dot<=0 DO
IF FALSE DO
{
  writef("dir =(%10.5d,%10.5d)*n", c, s)
  writef("p1 =(%10.5d,%10.5d)*n", p1!0, p1!1)
  writef("p2 =(%10.5d,%10.5d)*n", p2!0, p2!1)
  writef("p1dot=(%10.5d,%10.5d) m1=%n*n", p1!2, p1!3, m1)
  writef("p2dot=(%10.5d,%10.5d) m2=%n*n", p2!2, p2!3, m2)
CHAPTER 5. INTERACTIVE GRAPHICS IN BCPL USING SDL

writef("cgdot=(%10.5d,%10.5d)*n", cgxdot, cgydot)
writef("r1dot=(%10.5d,%10.5d)*n", rx1dot, ry1dot)
writef("r2dot=(%10.5d,%10.5d)*n", rx2dot, ry2dot)
writef("t1dot=(%10.5d,%10.5d)*n", t1dot, w1dot)
writef("t2dot=(%10.5d,%10.5d)*n", t2dot, w2dot)
writef("t1dot=%10.5d is the speed of the robot towards the centre of gravity*n", t1dot)
abort(1000)
}

IF t1dot<=0 RETURN

// Reverse t1dot and t2dot with some loss of energy
t1dot := rebound(t1dot)
t2dot := rebound(t2dot)

// Transform back to (x,y) coordinates relative to cg
rx1dot := inprod(t1dot,w1dot, c,-s)
ry1dot := inprod(t1dot,w1dot, s, c)
rx2dot := inprod(t2dot,w2dot, c,-s)
ry2dot := inprod(t2dot,w2dot, s, c)

// Convert to world (x,y) coordinates
p1!2 := rx1dot + cgxdot
p1!3 := ry1dot + cgydot
// Calculate cosine and sine of new direction of motion
p1!4 := cosines(p1!2, p1!3)
p1!5 := result2

p2!2 := rx2dot + cgxdot
p2!3 := ry2dot + cgydot
// Calculate cosine and sine of new direction of motion
p2!4 := cosines(p2!2, p2!3)
p2!5 := result2

AND rebound(vel) = vel/10 - vel // Returns the rebound speed of a bounce

AND cosines(dx, dy) = VALOF
{
// This function returns the cosine and sine of the angle between
// the line from (0,0) to (dx, dy) and the x axis.
// The result is the cosine and
// result2 is the sine.
LET d = ABS dx + ABS dy
LET c = muldiv(dx, One, d) // Approximate cos and sin
LET s = muldiv(dy, One, d) // Direction good, length not.
5.12. ROBOTS

LET a = muldiv(c, c, One) + muldiv(s, s, One) // 0.5 <= a <= 1.0
d := 1_00000 // With this initial guess only 3 iterations
// of Newton-Raphson are required.
// writef("a=%8.5d d=%8.5d d^2=%8.5d*n", a, d, muldiv(d, d, One))
d := (d + muldiv(a, One, d))/2
// writef("a=%8.5d d=%8.5d d^2=%8.5d*n", a, d, muldiv(d, d, One))
d := (d + muldiv(a, One, d))/2
// writef("a=%8.5d d=%8.5d d^2=%8.5d*n", a, d, muldiv(d, d, One))
d := (d + muldiv(a, One, d))/2
// writef("a=%8.5d d=%8.5d d^2=%8.5d*n", a, d, muldiv(d, d, One))
s := muldiv(s, One, d) // Corrected cos and sin
c := muldiv(c, One, d)
// writef("dx=%10.5d dy=%10.5d => cos=%8.5d sin=%8.5d*n", dx, dy, c, s)
result2 := s
RESULTIS c

AND inprod(dx, dy, c, s) = muldiv(dx, c, One) + muldiv(dy, s, One)

LET step() BE
{ IF started UNLESS finished DO
   displaytime := sdlmsecs() - starttime
   // writef("step: entered*n")
   IF bottlecount=0 DO finished := TRUE

   // Calculate the robot coordinates
   // robotcoords()

   // Bottle bounces
   FOR i = 1 TO bottlev!0 DO
   { LET bi = bottlev!i // bi -> [cgx, cgy, cgxdot, cgydot]

   UNLESS bi!b_dropped DO
   { LET xi = bi!b_cgx
     LET yi = bi!b_cgy
     // Test for bottle left wall bounces
     IF xi < wall_lx + bottleradius DO
     { xi := wall_lx + bottleradius
       bi!b_cgx := xi
       bi!b_cgxdot := - bi!b_cgxdot
     }
     // Test for bottle right wall bounces
IF xi > wall_rx - bottleradius DO
{ xi := wall_rx - bottleradius
  bi!b_cgx := xi
  bi!b_cgxdot := - bi!b_cgxdot
}
// Test for bottle floor bounces
IF yi < floor_y + bottleradius DO
{ yi := floor_y + bottleradius
  bi!b_cgy := yi
  bi!b_cgydot := - bi!b_cgydot
}
// Test for bottle ceiling bounces
IF yi > ceiling_y - bottleradius DO
{ yi := ceiling_y - bottleradius
  bi!b_cgy := yi
  bi!b_cgydot := - bi!b_cgydot
}
// Test for bottle-bottle bounces
FOR j = i+1 TO bottlev!0 DO
{ LET bj = bottlev!j
  IF bj!b_dropped LOOP
    IF incontact(bi, bj, bottleradius+bottleradius) DO
      cbounce(bi, bj, 1, 1)
  }
}

// Test for robot bounces
FOR i = 1 TO robotv!0 DO
{ LET r = robotv!i
  LET x = r!r_cgx
  LET y = r!r_cgy

  // Test for robot left wall bounces
  IF x < wall_lx + robotradius DO
  { x := wall_lx + robotradius
    r!r_cgx := x
    r!r_cgxdot := - r!r_cgxdot
  }

  // Test for robot right wall bounces
  IF x > wall_rx - robotradius DO
  { x := wall_rx - robotradius
    r!r_cgx := x
  }
5.12. **ROBOTS**

```plaintext
r!r_cgxdot := - r!r_cgxdot

// Test for robot floor bounces
IF y < floor_y + robotradius DO
{ LET cgxdot, cgydot = r!r_cgxdot, - r!r_cgydot
  y := floor_y + robotradius
  r!r_cgy := y
  r!r_cgydot := cgydot
  r!r_costheta := cosines(cgxdot, cgydot)
  r!r_sintheta := result2
}

// Test for robot ceiling bounces
IF y > ceiling_y - robotradius DO
{ LET cgxdot, cgydot = r!r_cgxdot, - r!r_cgydot
  y := ceiling_y - robotradius
  r!r_cgy := y
  r!r_cgydot := cgydot
  r!r_costheta := cosines(cgxdot, cgydot)
  r!r_sintheta := result2
}

// Test for robot pit bounces
IF incontact(r, thepit, robotradius+pitradius) DO
{ IF i=-11 DO
  { writef("Robot %n in contact with the pit*n", i)
    writef("Robot %n cg (%10.5d,%10.5d)*n", i, r!r_cgx, r!r_cgy)
    writef("Pit centre (%10.5d,%10.5d)*n", pit_x, pit_y)
  }
  cbounce(r, thepit, 0, 1)
}

// Test for robot bottle bounces
FOR j = 1 TO bottlev!0 DO
{ LET b = bottlev!j
  UNLESS b!b_dropped DO
  { UNLESS incontact(r, b, 8*robotradius) LOOP
    // This robot is near this bottle
    IF i=0 DO writef("robot %n near bottle %n*n", i, j)
  }
  // Test for robot body bounce
  IF incontact(r, b, robotradius+bottleradius) DO
    { // They are in contact so make the bottle bounce off
      //IF i=1 DO writef("Robot %n in contact with bottle %n*n", i, j)
  }
```
CHAPTER 5. INTERACTIVE GRAPHICS IN BCPL USING SDL

//cbounce(r, b, 10, 1) // Robot is 10 times heavier than a bottle
//cbounce(r, b, 1, 0) // Robot is heavy
}

// Test for robot left shoulder bounce
{ LET sx, sy, sxdot, sydot, sct, sst =
  r!r_cgx+r!r_lcx, r!r_cgy+r!r_lcy,
  r!r_cgxdot, r!r_cgydot, 0, 0
LET s = @sx
IF incontact(s, b, shoulderradius+bottleradius) DO
  { // They are in contact so make the bottle bounce off
    //IF i=1 DO writef("Robot %n in contact with bottle %n*n", i, j)
    //cbounce(s, b, 10, 1) // Shoulder is 10 times heavier than a bottle
    cbounce(s, b, 1, 0) // Robot is heavy
  }
}

// Test for robot right shoulder bounce
{ LET sx, sy, sxdot, sydot, sct, sst =
  r!r_cgx+r!r_rcx, r!r_cgy+r!r_rcy,
  r!r_cgxdot, r!r_cgydot, 0, 0
LET s = @sx
IF incontact(s, b, shoulderradius+bottleradius) DO
  { // They are in contact so make the bottle bounce off
    //IF i=1 DO writef("Robot %n in contact with bottle %n*n", i, j)
    //cbounce(s, b, 10, 1) // Shoulder is 10 times heavier than a bottle
    cbounce(s, b, 1, 0) // Robot is heavy
  }
}

// Test for robot left tip bounce
{ LET sx, sy, sxdot, sydot, sct, sst =
  r!r_cgx+r!r_ltcx, r!r_cgy+r!r_ltcy,
  r!r_cgxdot, r!r_cgydot, 0, 0
LET s = @sx
IF incontact(s, b, tipradius+bottleradius) DO
  { // They are in contact so make the bottle bounce off
    //IF i=1 DO writef("Robot %n in contact with bottle %n*n", i, j)
    //cbounce(s, b, 10, 1) // Shoulder is 10 times heavier than a bottle
    cbounce(s, b, 1, 0) // Robot is heavy
  }
}

// Test for robot right tip bounce
{ LET sx, sy, sxdot, sydot, sct, sst =
$r!r_{cgx} + r!r_{rtcx}, r!r_{cgy} + r!r_{rtcy},$
$r!r_{cgxdot}, r!r_{cgydot}, 0, 0$

LET $s = @sx$

IF incontact($s$, $b$, tipradius+bottleradius) DO
{
  // They are in contact so make the bottle bounce off
  //IF $i = 1$ DO writef("Robot $i$ in contact with bottle $j$", $i$, $j$)
  //cbounce($s$, $b$, 10, 1) // Shoulder is 10 times heavier than a bottle
  cbounce($s$, $b$, 1, 0) // Robot is heavy
}

// Test for robot grabber bounces
{
  // Make the robot's centre to origin
  LET $bx = b!b_{cgx} - r!r_{cgx}$
  LET $by = b!b_{cgy} - r!r_{cgy}$
  LET $c = r!r_{cosphi}$ // Direction cosines of the robot
  LET $s = r!r_{sinphi}$
  // Rotate clockwise the bottle position about the new origin
  LET $tx = \text{inprod}(bx, by, c, s)$
  LET $ty = \text{inprod}(bx, by, -s, c)$
  // Deal with bounces of the arm edges
  LET thickness = 2*tipradius // Arm thickness
  // Calculate the y position of the right edge of the left arm
  LET $y3 = \text{muldiv}(\text{robotradius-shoulderradius-thickness},$
          $r!r_{grabpos}, \text{One})$
  LET $y4 = y3 + \text{thickness}$ // Left edge of left arm
  LET $y2 = -y3$ // Left edge of right arm
  LET $y1 = y2 - \text{thickness}$ // Right edge of right arm
  IF $i = 0$ DO // Debugging aid
  {
    writef("robot $i2$ cg=(%10.5d %10.5d)*n", $i$, $r!r_{cgx}, r!r_{cgy}$)
    writef("bottle $i2$ cg=(%10.5d %10.5d)*n", $j$, $b!b_{cgx}, b!b_{cgy}$)
    writef("cosphi=%10.5d sinphi=%10.5d*n", $r!r_{cosphi}, r!r_{sinphi}$)
    writef("bx=%10.5d by=%10.5d*n", $bx$, $by$)
    writef("tx=%9.5d grablen=%9.5d*n", $tx$, $\text{grablen}$)
    writef("ty=%9.5d*n", $ty$)
    writef("y1=%9.5d y1=%9.5d y1=%9.5d y1=%9.5d*n", $y1$, $y2$, $y3$, $y4$)
  }
  IF robotradius <= $tx$ <= robotradius+grablen DO
  {
    // Bounces and grabbing are both possible
    IF $y1 - \text{bottleradius}$ <= $ty$ <= $y1$ DO
    {
      // Bottle collision with right edge of right arm
      LET rtdot = \text{inprod}(r!r_{cgxdot}, r!r_{cgydot}, c, s)
      LET rwdot = \text{inprod}(r!r_{cgxdot}, r!r_{cgydot}, -s, c)
      LET btdot = \text{inprod}(b!b_{cgxdot}, b!b_{cgydot}, c, s)
      LET bwdot = \text{inprod}(b!b_{cgxdot}, b!b_{cgydot}, -s, c)
    }
  }
LET v = bwdot-rwdot
IF v>0 DO
{ bwdot := rebound(v) + rwdot
  // Transform bottle velocity to world coords
  b!b_cgxdot := inprod(btdot,bwdot, c, -s)
  b!b_cgydot := inprod(btdot,bwdot, s, c)
}

IF i=0 DO
{ writef("robot %n collision bottle %n right edge of right arm*n", i, j)
  abort(1000)
}

IF y2 <= ty <= y2 + bottleradius DO
{ // Bottle collision with left edge of right arm
  //LET rtdot = inprod(r!r_cgxdot, r!r_cgydot, c, s)
  LET rwdot = inprod(r!r_cgxdot, r!r_cgydot,-s, c)
  LET btdot = inprod(b!b_cgxdot, b!b_cgydot, c, s)
  LET bwdot = inprod(b!b_cgxdot, b!b_cgydot,-s, c)
  LET v = bwdot-rwdot
  IF v<0 DO
  { bwdot := rebound(v) + rwdot
    // Transform bottle velocity to world coords
    b!b_cgxdot := inprod(btdot,bwdot, c, -s)
    b!b_cgydot := inprod(btdot,bwdot, s, c)
  }
  //IF tydot>0 DO tydot := rebound(tydot)
}

IF i=0 DO
{ writef("robot %n collision bottle %n left edge of right arm*n", i, j)
  abort(1000)
}

IF y3 - bottleradius <= ty <= y3 DO
{ // Bottle collision with right edge of left arm
  //LET rtdot = inprod(r!r_cgxdot, r!r_cgydot, c, s)
  LET rwdot = inprod(r!r_cgxdot, r!r_cgydot,-s, c)
  LET btdot = inprod(b!b_cgxdot, b!b_cgydot, c, s)
  LET bwdot = inprod(b!b_cgxdot, b!b_cgydot,-s, c)
  LET v = bwdot-rwdot
  IF v>0 DO
  { bwdot := rebound(v) + rwdot
    // Transform bottle velocity to world coords
    b!b_cgxdot := inprod(btdot,bwdot, c, -s)
    b!b_cgydot := inprod(btdot,bwdot, s, c)
  }
  IF i=0 DO
{ writef("robot %n collision bottle %n right edge of left arm\n", i, j)
 abort(1000)
}

IF y4 <= ty <= y4 + bottleradius DO
{ // Bottle collision with left edge of left arm
  LET rtdot = inprod(r!r_cgxdot, r!r_cgydot, c, s)
  LET rwdot = inprod(r!r_cgxdot, r!r_cgydot, -s, c)
  LET btdot = inprod(b!b_cgxdot, b!b_cgydot, c, s)
  LET bwdot = inprod(b!b_cgxdot, b!b_cgydot, -s, c)
  LET v = bwdot-rwdot
  IF v<0 DO
  { bwdot := rebound(v) + rwdot
   // Transform bottle velocity to world coords
   b!b_cgxdot := inprod(btdot,bwdot, c, -s)
   b!b_cgydot := inprod(btdot,bwdot, s, c)
  }

  IF i=0 DO
  { writef("robot %n collision bottle %n left edge of left arm\n", i, j)
    abort(1000)
  }

// Test for robot-robot bounces
FOR j = i+1 TO robotv!0 DO
  LET s = robotv!j
  IF incontact(r, s, robotradius+robotradius) DO
    IF i=-1 DO
      writef("Robot \%n in contact with robot \%n*n", i, j)
      writef("Robot \%n cg (%10.5d,%10.5d)*n", i, r!r_cgx, r!r_cgy)
      writef("Robot \%n cg (%10.5d,%10.5d)*n", j, s!r_cgx, s!r_cgy)
    } cbounce(r, s, 1, 1)
  }

// Robot motion
FOR i = 1 TO robotv!0 DO
  LET r = robotv!i // r -> [cgx, cgy, cgxdot, cgydot, costheta, sintheta]
  LET grabposdot = r!r_grabposdot
  LET grabpos = r!r_grabpos + grabposdot/sps
  LET cgxdot = r!r_cgxdot
  LET cgydot = r!r_cgydot
  //writef("robot %i2: cgx=%10.5d cgy=%10.5d cgxdot=%10.5d cgydot=%10.5d*n", i, r!r_cgx, r!r_cgy, cgxdot, cgydot)
  //abort(1000)
  // Rotate the direction of motion
  LET c = r!r_cosdtheta
  LET s = r!r_sindtheta
  LET xdot = inprod(cgxdot,cgydot, c, -s)
  LET ydot = inprod(cgxdot,cgydot, s,  c)
  r!r_cgxdot := xdot
  r!r_cgydot := ydot
  r!r_costheta := cosines(xdot, ydot)
  r!r_sintheta := result2
  r!r_cgx := r!r_cgx + xdot/sps
5.12. ROBOTS

r!r_cgy := r!r_cgy + ydot/sps
//r!r_cgdot := cgxdot * 95 / 100
//r!r_cgydot := cgydot * 95 / 100

IF grabpos < 0_10000 DO grabpos, grabposdot := 0_10000, 0
IF grabpos > 1_00000 DO grabpos, grabposdot := 1_00000, 0
r!r_grabpos, r!r_grabposdot := grabpos, grabposdot

// Robot orientation #FFFFFFFFFFFFFF
// Rotate slowly into the direction of motion
{ LET ct, st = r!r_costheta, r!r_sintheta
  LET cp, sp = r!r_cosphi, r!r_sinphi
  LET ca = inprod(ct, st, cp, sp)
  LET sa = inprod(st, ct, cp, -sp)
  LET corr = sa>0 -> One*5, // Correction per second
            -One*5
  IF ca>0 & -0_05000 < sa < 0_05000 DO corr := sa
  // Rotate
  corr := corr/sps
  ca := inprod( cp, sp, One, -corr)
  sa := inprod( sp, cp, One, corr)
  r!r_cosphi := cosines(ca, sa)
  r!r_sinphi := result2
}

// Bottle motion
FOR i = 1 TO bottlev!0 DO
{ LET b = bottlev!i // b -> [cgx, cgy, cgxdot, cgydot]
  UNLESS b!b_dropped DO
{ LET cgxdot = b!b_cgxdot
  LET cgydot = b!b_cgydot
  b!b_cgx := b!b_cgx + cgxdot/sps
  b!b_cgy := b!b_cgy + cgydot/sps
  //b!b_cgxdot := cgxdot * 95 / 100
  //b!b_cgydot := cgydot * 95 / 100
  IF incontact(b, thepit, 2*pitradius-bottleradius) DO
{ LET dir_x = cosines(b!b_cgx-pit_x, b!b_cgy-pit_y)
  LET dir_y = result2
  //writef("bottle=%n dx=%10.5d dy=%10.5d dir_x=%10.5d dir_y=%10.5d*n",
            i, b!b_cgx-pit_x, b!b_cgy-pit_y, dir_x, dir_y)
  //abort(1000)
  b!b_cgxdot := cgxdot + dir_x
  b!b_cgydot := cgydot + dir_y
}
CHAPTER 5. INTERACTIVE GRAPHICS IN BCPL USING SDL

IF incontact(b, thepit, pitradius-bottleradius) DO
  { b!b_dropped := TRUE
    bottlecount := bottlecount-1
  }
}

AND grabbottle(r, b) BE
{ LET mx = robotradius = 3*bottleradius/2
  LET my = 0
  LET c, s = r!r_cosphi, r!r_sinphi

  // Set the bottle's position and velocity
  b!b_cgx := inprod(mx,my, c, s) + r!r_cgx
  b!b_cgy := inprod(mx,my, -s, c) + r!r_cgy
  b!b_cgxdot := r!r_cgxdot
  b!b_cgydot := r!r_cgydot

  b!b_grabbed := TRUE
  b!b_robot := r // The grabbing robot
  r!r_bottle := b // The grabbed bottle
  r!r_grabposdot := 0 // Stop the grabber
  writef("Robot %n grabbed Bottle %n*n", r!r_id, b!b_id)
  //abort(3000)
}

AND releasebottle(r) BE IF r!r_bottle DO
{ LET b = r!r_bottle
  LET mx = robotradius + 3*bottleradius/2
  LET my = 0
  LET c, s = r!r_cosphi, r!r_sinphi

  // Set the bottle's position and velocity
  b!b_cgx := inprod(mx,my, c,-s) + r!r_cgx
  b!b_cgy := inprod(mx,my, s, c) + r!r_cgy
  b!b_cgxdot := r!r_cgxdot
  b!b_cgydot := r!r_cgydot

  b!b_grabbed := FALSE
  b!b_robot := 0 // No grabbing robot
  r!r_bottle := 0 // No grabbed bottle
  writef("Robot %n released Bottle %n*n", r!r_id, b!b_id)
  //abort(4000)
}
AND initpitsurf(col) = VALOF
{ // Allocate the pit surface
  LET height = 2*pitradius/One + 2
  LET width = height
  LET colkey = maprgb(64,64,64)
  LET surf = mksurface(width, height)

  selectsurface(surf, width, height)
  fillsurf(colkey)
  setcolourkey(surf, colkey)

  setcolour(col)
  drawfillcircle(pitradius/One, pitradius/One+1, pitradius/One)

  RESULTIS surf }

AND initbottlesurf(col) = VALOF
{ // Allocate a bottle surface
  LET height = 2*bottleradius/One + 2
  LET width = height
  LET colkey = maprgb(64,64,64)
  LET surf = mksurface(width, height)

  selectsurface(surf, width, height)
  fillsurf(colkey)
  setcolourkey(surf, colkey)

  setcolour(col)
  drawfillcircle(bottleradius/One, bottleradius/One+1, bottleradius/One)

  RESULTIS surf }

AND sine(theta) = VALOF
// theta = 0 for 0 degrees
// = 64000 for 90 degrees
// Returns a value in range -1000 to 1000
{ LET a = theta / 1000
  LET r = theta REM 1000
  LET s = rawsine(a)
  RESULTIS s + (rawsine(a+1)-s)*r/1000
}
AND cosine(x) = sine(x+64_000)

AND rawsine(x) = VALOF

{ // x is scaled d.ddd with 64.000 representing 90 degrees
  // The result is scaled d.ddd, ie 1.00000 represents 1.00000
  LET t = TABLE 0, 25, 49, 74, 98, 122, 147, 171,
       195, 219, 243, 267, 290, 314, 337, 360,
       383, 405, 428, 450, 471, 493, 514, 535,
       556, 576, 596, 615, 634, 653, 672, 690,
       707, 724, 741, 757, 773, 788, 803, 818,
       831, 845, 858, 870, 882, 893, 904, 914,
       924, 933, 942, 950, 957, 964, 970, 976,
       981, 985, 989, 992, 995, 997, 999, 1000,
       1000
  LET a = x&63
  UNLESS (x&64)=0 DO a := 64-a
  a := t!a
  UNLESS (x&128)=0 DO a := -a
  RESULTIS a * 100
}

AND robotcoords(r) BE

{ LET c = r!r_cosphi
  LET s = r!r_sinphi
  LET r1 = robotradius
  LET r2 = shoulderradius
  LET r3 = tipradius
  LET d1 = 2*r3
  LET d2 = muldiv(r!r_grabpos, r1-r2-d1, One)
  LET d3 = grablen
  LET ns = -s
  //IF r!r_reversing DO c, s, ns := -c, -s, s

  r!r_lcx := inprod( c,ns, r1-r2, r1-r2) // Left side
  r!r_lcy := inprod( s, c, r1-r2, r1-r2)
  r!r_lex := inprod( c,ns, r1, r1-r2)
  r!r_ley := inprod( s, c, r1, r1-r2)

  r!r_rcx := inprod( c,ns, r1-r2, r2-r1) // Right side
  r!r_rcy := inprod( s, c, r1-r2, r2-r1)
  r!r_rex := inprod( c,ns, r1, r2-r1)
  r!r_rey := inprod( s, c, r1, r2-r1)

  r!r_ltax := inprod( c,ns, r1, d1+d2) // Left arm
5.12. ROBOTS

\[
\begin{align*}
\text{r!r_ltay} &= \text{inprod}(\text{s, c, r1, d1+d2}) \\
\text{r!r_ltbx} &= \text{inprod}(\text{c,ns, r1, d2}) \\
\text{r!r_ltbx} &= \text{inprod}(\text{s, c, r1, d2}) \\
\text{r!r_ltcx} &= \text{inprod}(\text{c,ns, r1+d3, d2}) \\
\text{r!r_ltcx} &= \text{inprod}(\text{s, c, r1+d3, d2}) \\
\text{r!r_ltcx} &= \text{inprod}(\text{c,ns, r1+d3, d1+d2}) \\
\text{r!r_ltcx} &= \text{inprod}(\text{s, c, r1+d3, d1+d2}) \\
\text{r!r_ltpx} &= \text{inprod}(\text{c,ns, r1+d3, d2+r3}) \\
\text{r!r_ltpy} &= \text{inprod}(\text{s, c, r1+d3, d2+r3}) \\
\text{r!r_rtax} &= \text{inprod}(\text{c,ns, r1,-d1-d2}) \quad \text{// Right arm} \\
\text{r!r_rtay} &= \text{inprod}(\text{s, c, r1,-d1-d2}) \\
\text{r!r_rtbx} &= \text{inprod}(\text{c,ns, r1, -d2}) \\
\text{r!r_rtby} &= \text{inprod}(\text{s, c, r1, -d2}) \\
\text{r!r_rtcx} &= \text{inprod}(\text{c,ns, r1+d3, -d2}) \\
\text{r!r_rtcx} &= \text{inprod}(\text{s, c, r1+d3, -d2}) \\
\text{r!r_rtdx} &= \text{inprod}(\text{c,ns, r1+d3,-d1-d2}) \\
\text{r!r_rtdx} &= \text{inprod}(\text{s, c, r1+d3,-d1-d2}) \\
\text{r!r_rtpx} &= \text{inprod}(\text{c,ns, r1+d3,-d2-r3}) \\
\text{r!r_rtpy} &= \text{inprod}(\text{s, c, r1+d3,-d2-r3}) \\
\text{r!r_bcx} &= \text{inprod}(\text{c,ns, robotradius+3*bottleradius/2, 0}) \\
\text{r!r_bcy} &= \text{inprod}(\text{s, c, robotradius+3*bottleradius/2, 0})
\end{align*}
\]

AND drawrobot(i) BE

\{ LET r = robotv\!i
  LET x, y = r!r_cgx/One, r!r_cgy/One
  robotcoords(r)

  setcolour(r!r_colour)
  // Body
drawfillcircle(x, y, robotradius/One)
  // Left shoulder
drawfillcircle(x + r!r_lcx/One, y + r!r_lcy/One, shouderradius/One)
  // Right shoulder
drawfillcircle(x + r!r_rcx/One, y + r!r_rcy/One, shouderradius/One)

  setcolour(grabcolour)
  // Grabber base
drawquad(x + r!r_lcx/One, y + r!r_lcy/One, 
          x + r!r_lex/One, y + r!r_ley/One, 
          x + r!r_rex/One, y + r!r_rey/One, 
          x + r!r_rcx/One, y + r!r_rcy/One)
// Left arm
drawquad(x + r!r_ltax/One, y + r!r_ltax/One,
   x + r!r_ltbx/One, y + r!r_ltbx/One,
   x + r!r_ltcx/One, y + r!r_ltcx/One,
   x + r!r_ltdx/One, y + r!r_ltdx/One)
drawfillcircle(x + r!r_ltpx/One, y + r!r_ltpx/One, tipradius/One)
// Right arm
drawquad(x + r!r_rtax/One, y + r!r_rtax/One,
   x + r!r_rtbx/One, y + r!r_rtbx/One,
   x + r!r_rtcx/One, y + r!r_rtcx/One,
   x + r!r_rtdx/One, y + r!r_rtdx/One)
drawfillcircle(x + r!r_rtpx/One, y + r!r_rtpx/One, tipradius/One)
}

AND drawbottle(i) BE
{ LET b = bottlev!i
  LET bottlesurf = bottlesurfR // Normally red
  IF b!b_dropped RETURN
  IF i=1 DO bottlesurf := bottlesurfK

  IF b!b_grabbed DO
  { LET r = b!b_robot
    bottlesurf := bottlesurfB
    // Set the bottle coords
    b!b_cgx := r!r_cgx + r!r_bcx
    b!b_cgy := r!r_cgy + r!r_bcy
  }

  blitsurf(bottlesurf, screen, (b!b_cgx-bottleradius)/One,
           (b!b_cgy+bottleradius)/One)
  }

AND plotscreen() BE
{ selectsurface(screen, screenxsize, screenysize)
  fillsurf(backcolour)

  // Allocate the surfaces if necessary
  UNLESS bottlesurfR DO bottlesurfR := initbottlesurf(col_red)
  UNLESS bottlesurfK DO bottlesurfK := initbottlesurf(col_black)
  UNLESS bottlesurfB DO bottlesurfB := initbottlesurf(col_brown)
  UNLESS pitsurf DO pitsurf := initpitsurf(pitcolour)

  selectsurface(screen, xsize, ysize)
  //drawquad(100,-20, 102,-10, 102,-5, 150,-1)
//updatescreen()
//abort(1000)
// The pit
//blitsurf(pitsurf, screen, pit_xl/One, pit_yt/One)
blitsurf(pitsurf, screen,
    (pit_x-pitradius)/One, (pit_y+pitradius)/One)

selectsurface(screen, xsize, ysize)

// Must draw the robots first in case there are grabbed bottles
FOR i = 1 TO robotv!0 DO drawrobot(i)

FOR i = 1 TO bottlev!0 DO drawbottle(i)

setcolour(maprgb(255,255,255))

IF finished DO
    plotf(30, 300, "Finished -- Well Done!")

IF started | finished DO
    plotf(30, 280, "Time %9.2d", displaytime/10)

IF help DO
    { plotf(30, 150, "Q -- Quit")
      plotf(30, 135, "P -- Pause/Continue")
      plotf(30, 120, "H -- Toggle help information")
      plotf(30, 105, "G -- Grab")
      plotf(30, 90, "R -- Release")
      plotf(30, 75, "D -- Toggle debugging")
      plotf(30, 60, "U -- Toggle usage")
      plotf(30, 45, "Arrow keys -- Control the blue robot")
    }

setcolour(maprgb(255,255,255))

IF displayusage DO
    plotf(30, 245, "CPU usage = %i3%% sps = %n", usage, sps)

IF debugging DO
    { LET r = robotv!1
      LET b = bottlev!1
      plotf(30, 220, "Robot1 x=%10.5d y=%10.5d dot=%10.5d ydot=%10.5d",
          r!r_cgx, r!r_cgy, r!r_cgxdot, r!r_cgydot)
plotf(30, 205, " costheta=%10.5d  sinheta=%10.5d",
          r!r_costheta, r!r_sinheta)
plotf(30, 190, "cosdtheta=%10.5d  sindtheta=%10.5d",
          r!r_cosdtheta, r!r_sindtheta)
plotf(30, 175, " grabpos=%10.5d  grabposdot=%10.5d",
          r!r_grabpos, r!r_grabposdot)
plotf(30, 160, "Bottle1 x=%10.5d  y=%10.5d  xdot=%10.5d  ydot=%10.5d",
          b!b_cgx, b!b_cgy, b!b_cgxdot, b!b_cgydot)
}

AND process_events() BE WHILE getevent() SWITCH ON eventtype INTO
{ DEFAULT:
  LOOP
  CASE sdle_keydown:
    SWITCH ON capitalch(eventa2) INTO
    { DEFAULT:  LOOP
      CASE 'Q': done := TRUE
      LOOP
      CASE '?':
      CASE 'H': help := ~help
      LOOP
      CASE 'D': debugging := ~debugging
      LOOP
      CASE 'U': displayusage := ~displayusage
      LOOP
      CASE 'G': // Grab
        { LET r = robotv!1
        // Close grabber unless a bottle is already grabbed
        // UNLESS r!r_bottle DO
          r!r_grabposdot := -1_00000
        LOOP
        }
      CASE 'R': // Release
        { LET r = robotv!1
          r!r_grabposdot := +1_00000
          writef("Releasing grabber of robot 1 bottle node %n*n", r!r_bottle)
        // abort(5000)
IF r!r_bottle DO releasebottle(r)
LOOP
}

CASE 'S': // Start again
LOOP

CASE 'P': // Toggle stepping
stepping := ~stepping
LOOP

CASE sdle_arrowup:
{
LET r = robotv!1
LET cgxdot = r!r_cgxdot +
    muldiv(9_00000, r!r_cosphi, One)
LET cgydot = r!r_cgydot +
    muldiv(9_00000, r!r_sinphi, One)
// If cgxdot and cgydot have both changed signs
// the complement reversing
IF (cgxdot>0 & r!r_cgxdot<0 |
    cgxdot<0 & r!r_cgxdot>0) &
  (cgydot>0 & r!r_cgydot<0 |
    cgydot<0 & r!r_cgydot>0) DO
  r!r_reversing := ~ r!r_reversing
  r!r_cgxdot := cgxdot
  r!r_cgydot := cgydot
  r!r_costheta := cosines(cgxdot, cgydot)
  r!r_sintheta := result2
//r!r_cosphi := r!r_costheta
//r!r_sinphi := r!r_sintheta
//writef("costheta=%10.5d sinheta=%10.5d*n", r!r_costheta, r!r_sintheta)
//abort(5000)
  LOOP
}

CASE sdle_arrowdown:
{
LET r = robotv!1
LET cgxdot = r!r_cgxdot -
    muldiv(9_00000, r!r_cosphi, One)
LET cgydot = r!r_cgydot -
    muldiv(9_00000, r!r_sinphi, One)
// If cgxdot and cgydot have both changed signs
// the complement reversing
IF (cgxdot>0 & r!r_cgxdot<0 |
    cgxdot<0 & r!r_cgxdot>0) &
(cgydot>0 & r!r_cgydot<0 | cgydot<0 & r!r_cgydot>0) DO
  r!r_reversing := ~ r!r_reversing
  r!r_cgxdot := cgxdot
  r!r_cgydot := cgydot
  r!r_costheta := cosines(cgxdot, cgydot)
  r!r_sintheta := result2
  // r!r_cosphi := r!r_costheta
  // r!r_sinphi := r!r_sintheta
  // writef("costheta=%10.5d sintheta=%10.5d\n", r!r_costheta, r!r_sintheta)
  // abort(5000)
  LOOP
CASE sdle_arrowright:
  { LET r = robotv!1
  LET cdt = r!r_cosdtheta
  LET sdt = r!r_sindtheta
  LET dc = cosine(0_100)
  LET ds = sine(0_100)
  r!r_cosdtheta := inprod(cdt, sdt, dc, ds)
  r!r_sindtheta := inprod(cdt, sdt, -ds, dc)
  r!r_cosphi := r!r_costheta
  r!r_sinphi := r!r_sintheta
  LOOP
  }
CASE sdle_arrowsleft:
  { LET r = robotv!1
  LET cdt = r!r_cosdtheta
  LET sdt = r!r_sindtheta
  LET dc = cosine(0_100)
  LET ds = -sine(0_100)
  // writef("cdt=%10.5d sdt=%10.5d\n", cdt, sdt)
  // writef(" dc=%10.5d ds=%10.5d\n", dc, ds)
  r!r_cosdtheta := inprod(cdt, sdt, dc, ds)
  r!r_sindtheta := inprod(cdt, sdt, -ds, dc)
  r!r_cosphi := r!r_costheta
  r!r_sinphi := r!r_sintheta
  // writef("cdt=%10.5d sdt=%10.5d\n", r!r_cosdtheta, r!r_sindtheta)
  // abort(5000)
  LOOP
  }
CASE sdle_quit:
    writef("QUIT\n");
    done := TRUE
    LOOP
} 

AND nearedge(x, y, size) = VALOF
{ size := 8*size 

// writef("nearedge: x=%n y=%n size=%n xsize*One=%n ysize*One=%n*n", 
// x, y, size, xsize*One, ysize*One) 
//abort(1000) 
   UNLESS size < x < xsize*One - size RESULTIS TRUE 
   UNLESS size < y < ysize*One - size RESULTIS TRUE 
//writef("=> TRUE\n") 
RESULTIS FALSE
} 

AND nearpit(x, y, size) = VALOF
{ LET cx = pit_x 
   LET cy = pit_y //ysize*One/2 
   LET dx = ABS(x - cx) 
   LET dy = ABS(y - cy) 
   size := size + pitradius 
   IF dx < size | dy < size RESULTIS TRUE 
RESULTIS FALSE 
} 

AND nearbottle(x, y, size) = VALOF
{ size := 2*(size + bottleradius) 
FOR i = 1 TO bottlev!0 DO 
{ LET b = bottlev!i 
   LET bx = b!b_cg_x 
   LET by = b!b_cg_y 
   LET dx = ABS(x - bx) 
   LET dy = ABS(y - by) 
//writef("nearbottle: i=%i2 x=%n y=%n bx=%n by=%n size=%n*n", i, x, y, bx, by, size) 
   IF dx+dy < size RESULTIS TRUE 
} 
//writef("=>FALSE\n") 
//abort(1000) 
RESULTIS FALSE 
} 

AND nearrobot(x, y, size) = VALOF
{ size := 2*(size + robotradius)
    FOR i = 1 TO robotv!0 DO
        LET r = robotv!i
            LET rx = r!r_cgx
            LET ry = r!r_cgy
            LET dx = ABS(x - rx)
            LET dy = ABS(y - ry)
            //writef("nearrobot: i=%i2 x=%n y=%n bx=%n by=%n size=%n*n", i, x, y, bx, by, size)
            IF dx+dy < size RESULTIS TRUE
    }
    //writef("=>FALSE*n")
    //abort(1000)
    RESULTIS FALSE
}
setseed(seed)

UNLESS sys(Sys_sdl, sdl_avail) DO
{\texttt{writef("*nThe SDL features are not available*n")
  RESULTIS 0}
}

\texttt{spacev := getvec(spacevupb)}

UNLESS spacev DO
{\texttt{writef("Insufficient space available*n")
  RESULTIS 0}
}

\texttt{spacep, spacet := spacev, spacev+spacevupb}

IF FALSE DO
{\texttt{// Code to test the cosines function}
  LET e1, e2 = One, One
  FOR dy = 0 TO One BY One/100 DO
  \texttt{LET c, s, rsq = ?, ?, ?}
  \texttt{c := cosines(One, dy)}
  \texttt{s := result2}
  \texttt{rsq := inprod(c,c, s,s)}
  \texttt{writef("dx=%9.5d  dy=%9.5d  cos=%9.5d  sin=%9.5d  rsq=%9.5d*n",
                   One, dy, c, s, rsq)}
  \texttt{IF e1 < rsq D0 e1 := rsq}
  \texttt{IF e2 > rsq D0 e2 := rsq}
  \texttt{}}
\texttt{writef("Errors +%6.5d  -%7.5d*n", e1-One, One-e2)}
\texttt{RESULTIS 0}
}

\texttt{initsdl()}
\texttt{mkscreen("Robots", xsize, ysize)}

\texttt{backcolour := maprgb(120,120,120)}
\texttt{col_red := maprgb(255, 0, 0)}
\texttt{col_black := maprgb( 0, 0, 0)}
\texttt{col_brown := maprgb(100, 50, 20)}
\texttt{pitcolour := maprgb( 20, 20,100)}
\texttt{robotcolour := maprgb( 0,255, 0)}
\texttt{robot1colour := maprgb( 0,120, 40)}
\texttt{grabcolour := maprgb(200,200, 40)}
pit_x, pit_y := xsize*One/2, ysize*One/2
pit_xdot, pit_ydot := 0, 0
thepit := @pit_x

// Initialise robotv
robotv := mkvec(robots)
robotv!0 := 0
FOR i = 1 TO robots DO
  LET r = mkvec(r_upb)
  LET x = ?
  LET y = ?

  { x := randno(xsize*One)
    y := randno(ysize*One)
    UNLESS nearedge (x, y, robotradius) | nearpit (x, y, robotradius) | nearrobot(x, y, robotradius) BREAK
  } REPEAT

  robotv!0 := i
  robotv!i := r
  // Position
  r!r_cgx := x
  r!r_cgy := y
  // Motion
  r!r_cgxdot := randno(40_00000) - 20_00000
  r!r_cgydot := randno(40_00000) - 20_00000
  r!r_costheta := cosines(r!r_cgxdot, r!r_cgydot)
  r!r_sintheta := result2
  r!r_cosdtheta := 1_00000 // No rotation
  r!r_sindtheta := 0_00000
  r!r_reversing := FALSE
  // orientation
  r!r_cosphi := r!r_costheta
  r!r_sinphi := r!r_sintheta
  r!r_cosdphi := r!r_cosdtheta
  r!r_sindphi := r!r_sindtheta
  // grabber
  r!r_grabpos := 1_00000 // grabber open
  r!r_grabposdot := 0_00000
  r!r_bottle := 0 // No grabbed bottle
  r!r_colour := i=1 -> robot1colour, robotcolour
  r!r_id := i
robotcoords(r)
5.12. ROBOTS

}\n
// Initialise bottlev
bottlev := mkvec(bottles)
bottlev!0 := 0
FOR i = 1 TO bottles DO
{ LET b = mkvec(b_upb)
LET x = ?
LET y = ?

{ // Choose a random position for the next bottle
x := randno(xsize*One)
y := randno(ysize*One)
UNLESS nearedge (x, y, bottleradius) |
    nearpit (x, y, bottleradius) |
    nearrobot (x, y, robotradius) |
    nearbottle(x, y, bottleradius) BREAK
} REPEAT

bottlev!0 := i
bottlev!i := b
b!b_cgxx := x
b!b_cgy := y
b!b_cgx := randno(50_00000) - 25_00000
b!b_cgy := randno(50_00000) - 25_00000
b!b_grabbed := FALSE
b!b_robot := 0 // No grabbing robot
b!b_dropped := FALSE
b!b_id := i
}

help := FALSE //TRUE

stepping := TRUE // =FALSE if not stepping
starting := TRUE
started := FALSE
finished := FALSE
starttime := -1
displaytime := -1
usage := 0
debugging := FALSE
displayusage := FALSE
sps := 40 // Initial setting
stepmsecs := 1000/sps
wall_lx := 0
wall_rx := (screenxsize-1)*One // Right wall

floor_y := 0 // Floor
ceiling_y := (screenysize-1)*One // Ceiling

// Lots of initialisation ###########################
bottlesurfR := 0
bottlesurfK := 0
bottlesurfB := 0
pitsurf := 0
done := FALSE

UNTIL done DO
{ LET t0 = sdlmsecs()
  LET t1 = ?

  //writef("calling processevents\n")
  processevents()

  IF stepping DO step()

  usage := 100*comptime/stepmsecs
  //writef("calling plotscreen\n")
  plotscreen()
  //writef("calling updatescreen\n")
  updatescreen()

  //abort(2000)
  UNLESS 80<usage<95 DO
  { TEST usage>90
    THEN sps := sps-1
    ELSE sps := sps+1
      stepmsecs := 1000/sps
  }

  t1 := sdlmsecs()

  comptime := t1 - t0
  IF t0+stepmsecs > t1 DO sdldelay(t0+stepmsecs-t1)
  }

writef("\nQuitting\n")
sdldelay(0_200)
5.13. **MOON LANDER**

This is a re-implementation of a moon lander program originally written in September 1973 for the PDP-7 and the Vector General display. It now uses the SDL graphics library and runs under Linux, the Raspberry Pi and Windows. If you run the program without touching any of the controls the lander makes a perfect landing.

```
GET "libhdr"
GET "sdl.h"
GET "sdl.b" // Insert the library source code.
GET "libhdr"
GET "sdl.h"

MANIFEST {
  fuelmax=4000000
}

STATIC {
  shape=9111
  rotforce=0//50

  //** Perfect landing
  cgx= 322_855_260 // in millimetres
  cgy= 129_712_464 -16000 +3000
  theta= 3232
  cgxdot=-526_837 // in millimetres per second
  cgydot= -0_357
  thetadot= 32

  //**
```
/* Take off
  cgx=-37000000
  cgy=28001
  theta=64*1000
  cgxdot=0
  cgydot=1
  thetadot=-32
 */

mins = 400

fuel=fuelmax
thrust=450
dthrust=50

target=-37000000
halftargetsize=30_000 // in millimetres

scale=4
weight=300
mass=1

moonradius = 8000*#x1000 * 7 / 22 // circumference/pi

costheta=0

sintheta=0
flamelength=0

x0=0
y0=0
thrustmax=2000
thrustmin=100

single=FALSE
novice=FALSE
delay=1
offscreen=TRUE
ch=0
tracing=FALSE
}

GLOBAL {
  done:ug
  rotleft
  rotright

  landed       // Quality of the landing
toofast       // Quality of the landing
badsite
badorientation

goodlanding

5.13. MOON LANDER

stepping

col_black
col_blue
col_green
col_yellow
col_red
col_majenta
col_cyan
col_white
col_darkgray
col_darkblue
col_darkgreen
col_darkyellow
col_darkred
col_darkmajenta
col_darkcyan
col_gray
col_lightgray
col_lightblue
col_lightgreen
col_lightyellow
col_lightred
col_lightmajenta
col_lightcyan

Let start() = VALOF
{ Let mes = VEC 256/bytesperword

writes("*nMoon Lander*n")

initsdl()

mkscreen("Moon Lander", 640, 480)

rotleft, rotright := FALSE, TRUE

startlander(format)

// Update screen
updatescreen()

// Pause for 10 secs
sdldelay(10_000);
// Quit SDL
closesdl()

writef("Done!*n")

RESULT IS 0

AND startlander(fmt) = VALOF
{ LET count = 0

// Declare a few colours in the pixel format of the screen
col_black := maprgb(0, 0, 0)
col_blue := maprgb(0, 0, 255)
col_green := maprgb(0, 255, 0)
col_yellow := maprgb(0, 255, 255)
col_red := maprgb(255, 0, 0)
col_majenta := maprgb(255, 0, 255)
col_cyan := maprgb(255, 255, 0)
col_white := maprgb(255, 255, 255)
col_darkgray := maprgb(64, 64, 64)
col_darkblue := maprgb(0, 0, 64)
col_darkgreen := maprgb(0, 64, 0)
col_darkyellow := maprgb(0, 64, 64)
col_darkred := maprgb(64, 0, 0)
col_darkmajenta := maprgb(64, 0, 64)
col_darkcyan := maprgb(64, 64, 0)
col_gray := maprgb(128, 128, 128)
col_lightblue := maprgb(128, 128, 255)
col_lightgreen := maprgb(128, 255, 128)
col_lightyellow := maprgb(128, 255, 255)
col_lightred := maprgb(255, 128, 128)
col_lightmajenta := maprgb(255, 128, 255)
col_lightcyan := maprgb(255, 255, 128)

fillscreen(col_gray)

IF FALSE DO
{ LET days, msecs, flag = ?, ?, ?
  datstamp(@days)
  // Draw some random coloured lines rapidly
  setcolour(col_blue)
  drawpoint(screenxsize/2, screenysize/2)
FOR i = 1 TO 100_000 DO
{ LET col = maprgb(randno(255),randno(255),randno(255))
  LET x, y = randno(screenxsize)-1, randno(screenysize)-1
  IF i=10 DO setcaption("Hello World Again")
  setcolour(col)
  drawto(x, y)
  updatescreen()
  //sdldelay(100)
  IF i MOD 100 = 99 DO
  { LET d, m, f = ?, ?, ?
    datstamp(@d)
    printf("%8.3d frames per second*n", 100000_000/(m-msecs))
    days, msecs, flag := d, m, f
  }
  }
RESULTIS 0
}
lander()
RESULTIS 0

AND lander() BE
{ single := TRUE
  delay := 0
  landed := FALSE
  stepping := TRUE

  done := FALSE
  UNTIL done DO
  { readcontrols()
    IF stepping DO step()
    sdldelay(100)
  }

  WHILE sys(Sys_pollsardch)=pollingch LOOP
  writes("*nPress any key*n")
  sys(Sys_sardch)
  newline()
  }

AND setwindow() BE
{ // Set the position and scale of the window to display
  // ie set x0, y0 and scale.
  LET x, y = x0, y0
LET h = height(cgx)
LET relheight = ABS(cgy-h)

// Choose scale so that relheight appears no larger than half screenysize
LET s = relheight*2/screenysize
scale := minscale
UNTIL scale > s DO scale := scale*2

// Adjust y so that the moon's surface is suitably placed
UNLESS screenysize*2/10 < (h-y)/scale < screenysize*4/10 DO
  y := h - (screenysize*3/10)*scale
UNLESS screenysize/8 < (h-y0)/scale < screenysize/3 &
  screenysize/10 < (cgy-y0)/scale < screenysize*9/10 DO y0 := y

IF screenysize/4 > (cgx-x0)/scale DO x0 := cgx - (screenysize*3/5)*scale
IF screenysize*3/4 < (cgx-x0)/scale DO x0 := cgx - (screenysize*2/5)*scale

IF tracing DO
  { writef("cgx=\%n cgy=\%n h=\%n scale=\%n x=\%n y=\%n+x*x", cgx, cgy, h, scale, (cgx-x0)/scale, (cgy-y0)/scale)
    writef("screenxsize=\%n screenysize=\%n+x", screenxsize, screenysize)
  }
}

AND readcontrols() BE
{ WHILE getevent(@eventtype) SWITCHON eventtype INTO
  { DEFAULT:
    writef("Unknown event type = \%n+x", eventtype)
    LOOP
  CASE sdle_active: // => 1
    // writef("active \%d \%d+x", eventa1, eventa2)
    LOOP
  CASE sdle_keydown: // => 2 mod ch
    SWITCHON capitalch(eventa2) INTO
    { DEFAULT:
      LOOP
      CASE ".": rotforce := rotforce - 1
        IF rotforce<-1 DO rotforce := -1
        LOOP
      CASE ",": rotforce := rotforce + 1
        IF rotforce>1 DO rotforce := 1
        LOOP
      CASE 'Z': thrust := thrust - dthrust; LOOP
    }
CASE 'X': thrust := thrust + dthrust; LOOP
CASE 'T': tracing := ~tracing; LOOP
CASE 'P': stepping := ~stepping LOOP
CASE 'Q': done := TRUE; LOOP
}
LOOP

CASE sdle_keyup: // => 3 mod ch
  //writef("keyup %d %d*n", eventa1, eventa2)
  LOOP

CASE sdle_mousemotion: // 4
  //writef("mousemotion %n %n %n*n", eventa1, eventa2, eventa3)
  LOOP

CASE sdle_mousebuttondown: // 5
  //writef("mousebuttondown*n", eventa1, eventa2, eventa3)
  LOOP

CASE sdle_mousebuttonup: // 6
  //writef("mousebuttonup*n", eventa1, eventa2, eventa3)
  LOOP

CASE sdle_joyaxismotion: // 7
{ LET which = eventa1
  LET axis = eventa2
  LET value = eventa3
  //writef("joyaxismotion %n %n %n*n", eventa1, eventa2, eventa3)

  SWITCHON axis INTO
  { DEFAULT:
    LOOP

    CASE 0: // Aileron
      rotforce := 0
      IF value > 0 DO rotforce := -1
      IF value < 0 DO rotforce := +1
      LOOP

    CASE 1: // Elevator
      LOOP

    CASE 2: // Throttle
      thrust := thrustmax - muldiv(thrustmax-thrustmin, value+32769, 32768+32767)
      LOOP
CASE sdle_joyballmotion: // 8
   //writef("joyballmotion*n", eventa1, eventa2, eventa3)
   LOOP

CASE sdle_joyhatmotion: // 9
   //writef("joyhatmotion*n", eventa1, eventa2, eventa3)
   LOOP

CASE sdle_joybuttondown: // 10
   //writef("joybuttondown*n", eventa1, eventa2, eventa3)
   LOOP

CASE sdle_joybuttonup: // 11
   //writef("joybuttonup*n", eventa1, eventa2, eventa3)
   LOOP

CASE sdle_quit: // 12
   writef("QUIT*n");
   LOOP

CASE sdle_syswmevent: // 13
   //writef("syswmevent*n", eventa1, eventa2, eventa3)
   LOOP

CASE sdle_videosize: // 14
   //writef("videosize*n", eventa1, eventa2, eventa3)
   LOOP

CASE sdle_userevent: // 15
   //writef("userevent*n", eventa1, eventa2, eventa3)
   LOOP

AND step() BE
{ thetadot := thetadot + 20*rotforce

theta := theta + thetadot
IF novice DO theta, thetadot := theta+15*thetadot, 0

costheta := cosine(theta) // scaled d.ddd
sintheta := sine(theta)
IF thrust > thrustmax DO thrust := thrustmax
IF thrust < thrustmin DO thrust := thrustmin
IF fuel>0 DO { fuel := fuel - thrust
      IF fuel<0 DO fuel := 0
    }
IF fuel<=0 DO thrust := 0
flamelength := thrust*30000/thrustmax
cgxdot := cgxdot + (thrust*costheta/1000 )/mass
cgydot := cgydot + (thrust*sintheta/1000 - weight)/mass
// Add the effect of centrifugal force.
// This should allow the lander to remain in orbit, if cgxdot large enough.
//cgydot := cgydot + muldiv(cgxdot, cgxdot, cgy+moonradius)
cgx := cgx + cgxdot
cgy := cgy + cgydot
//writef("x=%n, y=%n*n", cgx, cgy)

IF tracing DO
{ writef("*nxydot= %n, %n*n", cgxdot, cgydot)
  writef("t,tdot = %n, %n*n", theta, thetadot)
  writef("x=%n, y=%n*n", cgx, cgy)
  writef("h = %n*n", height(cgx))
  // writef("x0y0= %n, %n*n", x0, y0)
  // writef("scale = %n*n", scale)
}

// The CG of the lander is 3 metre above the feet.
IF cgy <= height(cgx)+3_000 DO
{ toofast := FALSE
  badsite := FALSE
  badorientation := FALSE
  goodlanding := TRUE
  landed, thrust := TRUE, 0
  stepping := FALSE
  writes("*nLanded*n")
  writef("xdot = %7.3d ydot = %7.3d*n", cgxdot, cgydot)
UNLESS 0 < cgxdot*cgxdot+cgydot*cgydot < 1_500*1_500 DO
{ goodlanding := FALSE // Speed greater than 1.5 metre per second
  toofast := TRUE
  writef("Too fast*n")
}
// The craft width is 12 metres
UNLESS ABS(height(cgx-6_000) - height(cgx)) +
ABS(height(cgx+6_000) - height(cgx)) < 1000 DO
{ // Not level enough
  goodlanding := FALSE
  badsite := TRUE
  writef("Bad landing site*n")
}
UNLESS sintheta>950 DO
{ // Bad orientation
  goodlanding := FALSE
  badorientation := TRUE
  writes("Bad orientation*n")
  IF goodlanding DO writes("Perfect, Well done!!*n")
}

displayall()
}

AND height(x) = VALOF
{ IF -halftargetsize < x-target < halftargetsize DO x := target
  x := x/8000
  LET ra, rb, rc = x&777, x&77, x&7
  LET a, b, c = x-ra, x-rb, x-rc
  LET h = (hf(a)*(#777-ra) + hf(a+#1000)*ra +
           hf(b)*(#77 -rb) + hf(b+#100) *rb +
           hf(c)*(#7 -rc) + hf(c+#10) *rc)/512
  h := h*h/100
  IF (hf(x&2)&71)=0 DO h := h+4
  RESULTIS h*6*1000
}

AND hf(n) = VALOF
{ LET a = n XOR shape
  LET b = a*(a XOR #4132)/100 + a
  RESULTIS (b*b/313*a) & 255
}

AND cdrawto(x, y) BE
{ LET tx = x / minscale
  AND ty = y / minscale
  //writef("cdrawto: %n,%n ", x, y)
  x := (+tx*sintheta + ty*costheta)/1000 + (cgx-x0)/scale
}
5.13. MOON LANDER

\[ y := (-tx \cdot \cos \theta + ty \cdot \sin \theta) / 1000 + (cgy - y0) / \text{scale} \]

// writef(" %n,%n*n", x, y)
drawto(x, y)

AND cpoint(x, y) BE
{ LET tx = x / minscale
  AND ty = y / minscale
  x := (+tx \cdot \sin \theta + ty \cdot \cos \theta) / 1000 + (cgx - x0) / \text{scale}
y := (-tx \cdot \cos \theta + ty \cdot \sin \theta) / 1000 + (cgy - y0) / \text{scale}
drawpoint(x, y)
}

AND plotcraft() BE
{ setcolour(col_white)

  // The units are millimetres
  // The craft width is 12 metres (-6 to +6)
cpoint(-3000, -2000) // The base
cdrawto ( 3000, -2000)
cdrawto ( 3000, 0)
cdrawto ( -3000, 0)
cdrawto ( -3000, -2000)
cpoint( 1000, 0) // The return module
cdrawto ( 2000, 1000)
cdrawto ( 2000, 3000)
cdrawto ( 1000, 4000)
cdrawto ( -1000, 4000)
cdrawto ( -2000, 3000)
cdrawto ( -2000, 1000)
cdrawto ( -1000, 0)
cpoint( -3000, -1000) // The legs
cdrawto ( -5000, -3000)
cpoint( -6000, -3000)
cdrawto ( -4000, -3000)
cpoint( 3000, -1000)
cdrawto ( 5000, -3000)
cpoint( 4000, -3000)
cdrawto ( 6000, -3000)

setcolour(col_cyan)
IF thrust DO
{ cpoint( 0, -3000) // The flame
CHAPTER 5. INTERACTIVE GRAPHICS IN BCPL USING SDL

```plaintext
cdrawto ( -2000, -flamelength-3000)
cdrawto ( 0, -flamelength/2-3000)
cdrawto ( 2000, -flamelength-3000)
cdrawto ( 0, -3000)
}

IF thrust DO
    { IF rotforce>0 DO
        { setcolour(col_yellow)
            cpoint(-3000, 0) // Rotate left jets
cdrawto( -3500, 2000)
cdrawto( -2500, 2000)
cdrawto( -3000, 0)
cpoint( 3000,-2000)
cdrawto( 2500,-4000)
cdrawto( 3500,-4000)
cdrawto( 3000,-2000)
        }
        IF rotforce<0 DO
            { setcolour(col_yellow)
                cpoint( 3000, 0) // Rotate right jets
cdrawto( 3500, 2000)
cdrawto( 2500, 2000)
cdrawto( 3000, 0)
cpoint(-3000,-2000)
cdrawto( -2500,-4000)
cdrawto( -3500,-4000)
cdrawto( -3000,-2000)
            }
        }
    }

AND plotmoon() BE
    { LET x, dx = 0, 4//screenxsize/128

        setcolour(col_lightblue)

drawpoint(x, (height(x0)-y0)/scale)

        WHILE x<screenxsize DO
            { x := x+dx
```
5.13. MOON LANDER

```plaintext
    drawto(x, (height(x0+scale*x)-y0)/scale)
}

setcolour(col_lightmagenta)
drawpoint((target-halftargetsize-x0)/scale, (height(target)-y0)/scale)
drawto ((target+halftargetsize-x0)/scale, (height(target)-y0)/scale)
}

AND displayall() BE
{ LET xm = screenxsize/2
  LET tary = screensize - 60
  LET fuely = screensize - 30
  LET fuelxl = xm - 100
  LET fuelxh = xm + 100
  LET fuelx = fuelxl + muldiv(200, fuel, fuelmax)
  LET targx = xm + (target-cgx)/100000
  LET targxl = xm + (target-cgx)/1000000
  LET tdotx = xm - thetadot/8
  LET tdoty = fuely-15
  LET flx0, fly0 = xm, fuely-100
  LET flxs, flys = flamelength*costheta/1000, flamelength*sintheta/1000

  sys(Sys_sdl, sdl_fillsurf, screen, col_darkgray)
  setwindow()

  setcolour(col_cyan)             // Fuel
drawpoint(fuelxl, fuely)
drawby(200, 0)
setcolour(col_red)
drawpoint(fuelx, fuely)
drawby(0, 20)

  setcolour(col_lightmagenta)    // Target
drawpoint(targx-10, tary)
drawby(20, 0)
drawpoint(targx1-5, tary-2)
drawby(5, 0)

  setcolour(col_cyan)            // Thetadot
drawpoint(xm, fuely)
drawby(0, -15)
setcolour(col_red)
drawpoint(tdotx, tdoty)
drawby(0, -15)
```

setcolour(col_lightgreen) // Acceleration
drawpoint(flx0, fly0)
drawby(flxs/200, flys/200)

setcolour(col_red) // Velocity
drawpoint(flx0, fly0)
drawby(cgxdot/10_000, cgydot/10_000)

{ LET x = flx0+cgxdot/200-1
LET y = fly0+cgydot/200-1
   drawfillrect(x, y, x+3, y+3) // Velocity/200
}

setcolour(col_white)
plotf(10, 75, "target \%11.3d", target-cgx)
plotf(10, 60, "cgx= \%11.3d xdot=\%9.3d", cgx, cgxdot)
plotf(10, 45, "cgy= \%11.3d ydot=\%9.3d", cgy, cgydot)
plotf(10, 30, "fuel= \%11.3d", fuel)
//plotf(10, 15, "scale= \%11.3d", scale)

IF landed DO
{ LET x = screenxsize/2
LET y = screenysize/2
   plotf(x, y, "Landed")
   IF toofast DO { y := y-15; plotf(x, y, "Too fast") } 
   IF badsite DO { y := y-15; plotf(x, y, "Bad site") } 
   IF badorientation DO { y := y-15; plotf(x, y, "Bad orientation") } 
   IF goodlanding DO { y := y-15; plotf(x, y, "Perfect landing -- well done!") } 
}

plotmoon()
plotcraft()

ret1: 
   updatescreen()
}

AND rdjoystick() = 0

AND rdn() = VALOF
{ LET res = 0
   ch := sys(10)
   WHILE '0'<=ch<='9' DO { res := 10*res + ch - '0'
      ch := sys(10)
   }
RESULTIS res

AND sine(theta) = VALOF
// theta = 0 for 0 degrees
// = 64000 for 90 degrees
// Returns a value in range -1000 to 1000
{ LET a = theta / 1000
  LET r = theta REM 1000
  LET s = rawsine(a)
  RESULTIS s + (rawsine(a+1)-s)*r/1000
}

AND cosine(x) = sine(x+64_000)

AND rawsine(x) = VALOF
{ // x is scaled d.ddd with 64.000 representing 90 degrees
  // The result is scaled d.ddd, ie 1000 represents 1.000
  LET t = TABLE 0, 25, 49, 74, 98, 122, 147, 171,
   195, 219, 243, 267, 290, 314, 337, 360,
   383, 405, 428, 450, 471, 493, 514, 535,
   556, 576, 596, 615, 634, 653, 672, 690,
   707, 724, 741, 757, 773, 788, 803, 818,
   831, 845, 858, 870, 882, 893, 904, 914,
   924, 933, 942, 950, 957, 964, 970, 976,
   981, 985, 989, 992, 995, 997, 999, 1000,
   1000

  LET a = x&63
  UNLESS (x&64)=0 DO a := 64-a
  a := t!a
  UNLESS (x&128)=0 DO a := -a
  RESULTIS a
}

As the lander approaches the landing site, the screen should look something like the following.
5.14 A Library for High Precision Arithmetic

You may well wonder why there is a section here covering a library for high precision arithmetic when we have already seen simple examples of such arithmetic already. The reason is that the next two sections concern the tracing of rays of light through a catadioptric telescope and optics is renowned for needing high precision arithmetic including functions for division and square root. Efficient implementations of these two functions make use of Newton-Raphson iterations which have only recently been covered in page 303. The programs also use of SDL graphics which have only just been covered.

For convenience, the library is located in BCPL/cintcode/g and consists of a arith.b and a header file arith.h. Programs using this library should typically start as follows.

GET "libhdr"
MANIFEST {
    ArithGlobs=350 // The first global used by the arith library
    numupb=2+25    // Room for the sign, the exponent and 25 radix
digits, equivalent to 100 decimal digits.
    numupb        // Each radix digit is in the range 0 to 9999
}
GET "arith.h"
GET "arith.b"
The constant `ArithGlobs` allows the user to avoid global number clashes with other libraries. The constant `numupb` specifies the precision of numbers used internally by the library particularly in the implementation of divide and square root. Numbers in the user’s program will normally use a slightly lower precision.

The header file `arith.h` contains the following declarations.

```
GLOBAL {
  str2num: ArithGlobs
  setzero
  settok
  copy
  copyu
  addcarry
  roundnum
  standardize
  addu
  add
  subu
  sub
  neg
  mul
  mulbyk
  div
  divbyk
  exptok
  sqrt
  inprod
  radius
  normalize
  iszero
  numcmpu
  numcmp
  integerpart
  roundtoint
  prnum
  checknum
}
```

The file `arith.b` contains the definitions of all the `arith` library functions. Numbers processed by this library are represented by variable length vectors containing the sign, the exponent and fractional part. Typically, a number is passed to the functions in this library as a pair `(N, upb)` where `N` is the vector and `upb` is its upperbound. For such a number the following holds:
CHAPTER 5. INTERACTIVE GRAPHICS IN BCPL USING SDL

\[ N!0 = \text{TRUE if the number is negative.} \]
\[ = \text{FALSE if the number is greater than or equal to 0.} \]

\[ N!1 \]
is the exponent \( e \), ie the value of the number is \( \text{fractional}_\text{part} \times 10000^e \).

The exponent may be positive or negative.

\[ N!2 \text{ to } N!\text{upb} \]
hold the fractional part with an assumed decimal point to the left of the first digit in \( N!2 \). The digits are in the range 0 to 9999.

If \( N \) is in standard form then either \( N!2 \) is non zero or all its elements are zero. The fractional part contains digits of radix 10000 so the approximate precision of the number is slightly less than \( 4 \times (\text{upb} - 2) \) decimal digits. Rounding errors cause some loss of precision and also \( N!2 \) may contain fewer the 4 significant decimal digits.

After some comments, \texttt{arith.b} starts with the definition of \texttt{str2num} as follows.

\begin{verbatim}
LET str2num(s, n1, upb1) = VALOF
{ LET p = 0 // Count of decimal digits not including
  // leading zeroes
  LET fp = -1 // Count of decimal digits after the decimal point, if any.
  LET pos = 2 // Position of next radix digit
  LET dig = 0 // To hold the next radix digit
  LET dexp = ?
  LET e = 0 // For the exponent specified by En
  n1!0 := -2 // No sign yet
  FOR i = 1 TO upb1 DO n1!i := 0
  FOR i = 1 TO s%0 DO
    { LET ch = s%i
      SWITCHON ch INTO
      { DEFAULT: RESULTIS FALSE
      CASE ' ': LOOP // Ignore spaces
      CASE '-': UNLESS n1!0=-2 RESULTIS FALSE
        n1!0 := TRUE
        LOOP
      CASE '+': UNLESS n1!0=-2 RESULTIS FALSE
        n1!0 := FALSE
        LOOP
      CASE '.': IF fp>=0 RESULTIS FALSE // Invalid decimal point
        fp := 0 // Count of digits after the decimal point.
  }
}
\end{verbatim}
LOOP
CASE 'E':
CASE 'e': { // Read a possibly signed exponent leaving
   // its value in e.
   LET nege = -2
   FOR j = i+1 TO s%0 DO
   { ch := s%j
      SWITCHON ch INTO
      { DEFAULT: RESULTIS FALSE
        CASE ' ': LOOP // Ignore spaces
        CASE '-': UNLESS nege=-2 RESULTIS FALSE
           nege := TRUE
           LOOP
        CASE '+': UNLESS nege=-2 RESULTIS FALSE
           nege := FALSE
           LOOP
        CASE '0':CASE '1':CASE '2':CASE '3':CASE '4':
        CASE '5':CASE '6':CASE '7':CASE '8':CASE '9':
           IF nege=-2 DO nege := FALSE
           e := 10*e + ch - '0'
           LOOP
      }
   }
   IF nege DO e := -e
   BREAK
}
CASE '0': IF p=0 DO
   { // No significant decimal digits yet
     // If sign unset make it positive
     IF n1!0=-2 DO n1!0 := FALSE
     IF fp>=0 DO fp := fp+1
     LOOP
   }
CASE '1':CASE '2':CASE '3':CASE '4':
CASE '5':CASE '6':CASE '7':CASE '8':CASE '9':
   { // A significant digit
     // If sign unset make it positive
     IF n1!0=-2 DO n1!0 := FALSE
     p := p+1 // Increment count of significant digits
IF fp>=0 DO fp := fp+1 // Count of fractional digit
dig := 10*dig + ch - '0'
IF p MOD 4 = 0 DO
{ // Just completed a radix digit
  // Store it digit, if possible
  IF pos<=upb1 DO n1!pos := dig
  dig := 0
  pos := pos+1
}
LOOP

IF p=0 DO
{ // No significant digits, so the result is zero.
  setzero(n1,upb1)
  RESULT IS TRUE
}

// Place a decimal point here if not already present.
IF fp<0 DO fp := 0

// Pad last radix digit by adding fractional decimal zeroes.
UNTIL p MOD 4 = 0 DO
{ dig := dig * 10
  p := p+1
  IF fp >= 0 DO fp := fp+1
}

// Store the last digit, if room.
IF pos<= upb1 DO n1!pos := dig

// n1!2 contains 4 decimal digits including the padding zeroes.
dexp := p-fp // Decimal exponent

// p is the number of decimal digits including padding.
// p is a multiple of 4.
// fp is the number of fractional decimal digits including padding.

// We require dexp to be a multiple of 4, so
// until dexp is a multiple of 4, increment dexp and divide
// the fractional value by 10.
UNTIL dexp MOD 4 = 0 DO
{ divbyk(10, n1, upb1)
    dexp := dexp + 1
}

// The decimal exponent dexp is now a multiple of 4.

n1!1 := dexp/4 // Set the radix exponent.

// Now add in the En exponent
n1!1 := n1!1 + e

// checknum(n1, upb1)
RESULTIS TRUE
}

This function converts a string representing a high precision number into its vector form. The number starts with an optional sign followed by decimal digits and at most one decimal point. An explicit exponent can then be given consisting of E or e followed by a possibly signed integer. The exponent represents a power of 10000. Spaces are ignored, so for instance: str2num("-12.3456 789 E3", N, 5) will set the elements of N to:

[TRUE, 4, 0012, 3456 7890, 0000]

A vector representation of a number in the range \(-9999\) to \(+9999\) may be set using the function settok defined as follows.

AND settok(k, n1, upb1) = VALOF
{ // k must be in range \(-9999\) to \(+9999\)
    // Return TRUE is k is in range
    setzero(n1, upb1)
    IF k=0 RESULTIS TRUE
    IF k<0 DO n1!0, k := TRUE, -k
    IF k>=10000 RESULTIS FALSE
    n1!1, n1!2 := 1, k
    RESULTIS TRUE
}

The integer part of a number can be found using integerpart defined below. This function requires the number to be in standard form and in the range \(-9999.9999\) to \(+9999.9999\). If the integer part is out of range the result is either \(+1_0000_0000\) or \(-1_0000_0000\).

AND integerpart(n1, upb1) = VALOF
{ LET e = n1!1
  LET x = n1!2 * 10000
  IF n1!0 DO x := -x
  IF upb1 >= 3 DO x := x + n1!3
  IF e > 2 DO x := 1_0000_0000
  IF e <= 0 RESULTIS 0
  IF e = 1 RESULTIS x / 10000
  RESULTIS x
}

The following function rounds its argument to the nearest integer, returning a value representing the first 8 decimal digits after the decimal point before rounding takes place.

AND roundtoint(n1,upb1) = VALOF
{ LET e = n1!1 // The exponent
  LET frac = 0

  IF e > 0 DO
    { // The integer part is non zero

      // e > 0
      LET p = e+2 // Position of the first fractional radix digit
      LET carry = p<upb1 & n1!p>=5000 -> 1, 0
      IF p <= upb1 DO frac := n1!p * 10000 // Fractional digits 1 to 4
      IF p < upb1 DO frac := frac + n1!(p+1) // Fractional digits 5 to 8

      FOR i = p TO upb1 DO n1!i := 0
      IF carry DO addcarry(n1,p-1)

      //checknum(n1,upb1)
      RESULTIS frac
    }

    // The unrounded integer part is zero, so the rounded
    // integer part is zero or 1.
    // e <= 0
    IF e = 0 DO frac := n1!2*10000 + n1!3 // 8 fractional digits
    IF e = -1 DO frac := n1!2 // 4 fractional digits

    TEST e=0 & frac >= 50000000
    THEN settok(1, n1,upb1) // n1 was in range 0.5 to 0.99999999
    ELSE setzero(n1,upb1) // n1 was in range 0.0 to 0.49999999

    //checknum(n1,upb1)
The next function outputs a character representation of the high precision number it is given.

\[ \text{AND prnum(n, upb) BE} \]
\[ \{ \text{// Output a number n of size upb, followed by a newline().} \]
\[ \text{writef("\%c0."', n!0->'-', '+'\})} \]
\[ \text{FOR i = 2 TO upb DO} \]
\[ \{ \text{writef("\%z4 ", n!i)} \]
\[ \text{IF (i-2) MOD 10 = 9 DO writef("*n \})} \]
\[ \}\]
\[ \text{IF n!1 DO writef("E\%n", n!1)} \]
\[ \text{newline()} \}\]

For example, if \( N \) points to: [TRUE, 4, 0012, 3456 7890, 0000], \prnum(N,5) outputs: -0.0112 3456 7890 0000 E4.

The function defined below compares the magnitude of two high precision numbers returning \(-1, 0\) or \(+1\) depending on whether the absolute value of the first number is less, equal, or greater than the absolute value of the second.

\[ \text{AND numcmpu(n1,upb1, n2,upb2) = VALOF} \]
\[ \{ \text{// Return 1 if abs n1 > abs n2} \]
\[ \text{// Return 0 if abs n1 = abs n2} \]
\[ \text{// Return -1 if abs n1 < abs n2} \]
\[ \text{// n1 and n2 are assumed to be in standard form.} \]
\[ \text{LET upb = upb1<=>upb2 \rightarrow upb1, upb2} \]
\[ \text{// upb is the smaller upper bound} \]
\[ \text{// Deal with the cases when n1 or n2 is zero.} \]
\[ \text{IF n1!2=0 DO} \]
\[ \{ \text{IF n2!2=0 RESULTIS 0 // n1= 0 n2= 0} \]
\[ \text{RESULTIS -1 // n1= 0 n2^=0} \]
\[ \}\]
\[ \text{IF n2!2=0 RESULTIS 1 // n1^=0 n2= 0} \]
\[ \text{// Neither n1 nor n2 is zero} \]
\[ \text{FOR i = 1 TO upb DO} \]
\[ \{ \text{// Compare the exponents and digits of n1 and n2.} \]
\[ \text{LET a, b = n1!i, n2!i} \]
\[ \text{IF a > b RESULTIS 1 // n1 > n2} \]
\[ \text{IF a < b RESULTIS -1 // n1 < n2} \]
CHAPTER 5. INTERACTIVE GRAPHICS IN BCPL USING SDL

The function defined below compares two standardized signed numbers, returning -1, 0 or +1.

```
AND numcmp(n1,upb1, n2,upb2) = VALOF
{ // Return 1 if n1 > n2
  // Return 0 if n1 = n2
  // Return -1 if n1 < n2
  // n1 and n2 are assumed to be in standard form.
  IF n1!0 DO
    { IF n2!0 RESULTIS - numcmpu(n1,upb1, n2,upb2) // n1< 0, n2< 0
      RESULTIS -1 // n1< 0, n2>=0
    }
    IF n2!0 RESULTIS 1 // n1>=0, n2< 0
    RESULTIS numcmpu(n1,upb1, n2,upb2) // n1>=0 n2>=0
  }

  IF n2!0 RESULTIS 1 // n1>=0, n2< 0
  RESULTIS numcmpu(n1,upb1, n2,upb2) // n1>=0 n2>=0
}
```

The next function standardizes n1 then sets n2 = -n1.

```
AND neg(n1,upb1) = VALOF
{ // Standardize n1 and the set n1 = -n1
  standardize(n1,upb1)
  IF n1!2 DO n1!0 := ~ n1!0
  RESULTIS TRUE
}
```

The function iszero defined below returns TRUE if n1 is zero. If n1 is known to be in standard form, it is more efficient just to test n1!2.

```
AND iszero(n1,upb1) = VALOF
{ // n1 is zero if all the fraction digits are zero
  FOR i = 2 TO upb1 UNLESS n1!i=0 RESULTIS FALSE
  RESULTIS TRUE
}
```
The next function sets \( n_3 \) to \( \text{ABS } n_1 + \text{ABS } n_2 \), assuming \( n_1 \) and \( n_2 \) are in standard form. The result is rounded.

\[
\text{AND addu}(n_1, \text{upb}_1, n_2, \text{upb}_2, n_3, \text{upb}_3) = \text{VALOF}
\]

\[
\{
\begin{align*}
\text{// Set } n_3 \text{ to abs } n_1 + \text{abs } n_2 \text{ rounded } \\
\text{// } n_1 \text{ and } n_2 \text{ are assumed to be in standard form.}
\end{align*}
\]

\[
\text{LET carry } = 0
\]

\[
\text{LET } t_1, u_1 = n_1, \text{upb}_1 \quad // \text{To hold the number with the larger exponent}
\]

\[
\text{LET } t_2, u_2 = n_2, \text{upb}_2 \quad // \text{To hold the number with the smaller exponent}
\]

\[
\text{LET offset } = ?
\]

\[
\text{LET } p, q = ?, ?
\]

\[
\text{LET } \text{tmp} = \text{VEC numupb}
\]

\[
\text{//checknum}(n_1, \text{upb}_1)
\]

\[
\text{//checknum}(n_2, \text{upb}_2)
\]

\[
\text{IF } n_1 > 0 \text{ RESULTIS copyu}(n_2, \text{upb}_2, n_3, \text{upb}_3)
\]

\[
\text{IF } n_2 > 0 \text{ RESULTIS copyu}(n_1, \text{upb}_1, n_3, \text{upb}_3)
\]

\[
\text{// Neither } n_1 \text{ nor } n_2 \text{ are zero.}
\]

\[
\text{IF } n_1 < n_2 \text{ D0 } // \text{Compare their exponents}
\]

\[
\{
\begin{align*}
& t_1, u_1 := n_2, \text{upb}_2 \\
& t_2, u_2 := n_1, \text{upb}_1
\end{align*}
\}
\]

\[
\text{// } t_1 \geq t_2 \quad \text{So } t_1 \text{ has the larger exponent}
\]

\[
\text{offset } := t_1 - t_2
\]

\[
\text{// offset is } \geq 0
\]

\[
\text{// It is the amount } t_2 \text{ must be shifted before adding to } t_1
\]

\[
\text{p } := u_2 \quad // \text{Position of the last digit of } t_2
\]

\[
\text{q } := u_2 + \text{offset} \quad // \text{Position in } \text{tmp} \text{ of where to add it.}
\]

\[
\text{IF } q > \text{numupb} \text{ D0}
\]

\[
\{
\begin{align*}
& \text{// Reduce both } p \text{ and } q \text{ so the } q=\text{numupb}
& p := p - (u_2 + \text{offset} - \text{numupb})
& q := \text{numupb}
\end{align*}
\}
\]

\[
\text{// Form the sum in } \text{tmp}
\]

\[
\text{copyu}(t_1, u_1, \text{tmp}, \text{numupb})
\]

\[
\text{// Add } t_2 \text{ suitably shifted into } \text{tmp}
\]

\[
\text{WHILE } p \geq 2 \text{ D0}
\]

\[
\text{WHILE } p \geq 2 \text{ D0}
\]

\[
\text{WHILE } p \geq 2 \text{ D0}
\]
\{ \text{LET } x = \text{tmp!q }+ t2!p + \text{carry} \\
\quad \text{tmp!q} := x \mod 10000 \\
\quad \text{carry} := x / 10000 \\
\quad p, q := p-1, q-1 \}

// There are no more digits of t2, but the may still be a carry 
// to deal with 
WHILE carry & q >= 2 DO 
\{ \text{LET } x = \text{tmp!q }+ \text{carry} \\
\quad \text{tmp!q} := x \mod 10000 \\
\quad \text{carry} := x / 10000 \\
\quad q := q-1 \}

// If there is a carry out of the senior digit, tmp must be 
// shifted right and the exponent corrected. 
IF carry DO 
\{ \text{// Shift the radix digits to the right by one position.} \\
\quad \text{FOR } i = \text{numupb}-1 \text{ TO } 2 \text{ BY } -1 \text{ DO } \text{tmp!(i+1)} := \text{tmp!i} \\
\quad \text{tmp!2} := \text{carry} \\
\quad \text{tmp!1} := \text{tmp!1} + 1 // Adjust the exponent \}

copy(tmp,numupb, n3,upb3) // Set n3 = tmp rounded

//checknum(n3,upb3) 
RESULTIS TRUE \}

The next function sets n3 to ABS n1 - ABS n2 assuming the result is greater 
than or equal to zero. Both n1 and n2 are assumed to be in standard form. The 
result is rounded.

AND subu(n1,upb1, n2,upb2, n3,upb3) = VALOF 
\{ \text{LET } borrow = 0 \\
\quad \text{LET } t1,u1 = n1,upb1 // To hold the number with the larger exponent \\
\quad \text{LET } t2,u2 = n2,upb2 // To hold the number with the smaller exponent \\
\quad \text{LET } offset = ? \\
\quad \text{LET } p, q = ?, ? \\
\quad \text{LET } \text{tmp} = \text{VEC numupb} \\

\quad //checknum(n1,upb1) \\
\quad //checknum(n2,upb2) \}
IF \( n_2 \neq 0 \) RESULTIS \( \text{copyu}(n_1, \text{upb}_1, n_3, \text{upb}_3) \)

IF \( n_1 \neq 0 \) DO
{ // Since \( \text{abs } n_1 \geq \text{abs } n_2 \) and \( n_1 = 0 \) the so does \( n_2 \).
  \text{setzero}(n_3, \text{upb}_3)
  \text{RESULTIS } \text{TRUE}
}

// Neither \( n_1 \) nor \( n_2 \) are zero.

// Since \( \text{abs } n_1 \geq \text{abs } n_2 \) and they are both non zero,
// the exponent of \( n_1 \) must be \( \geq \) exponent of \( n_2 \)

// is \( n_1 \) >= \( n_2 \)  So \( t_1 \) has the larger exponent

offset := \( n_1 \) - \( n_2 \)
// offset is >= 0
// It is the amount \( n_2 \) must be shifted before adding to \( n_1 \)

p := \text{upb}_2 // Position of the last digit of \( n_2 \)
q := \text{upb}_2 + \text{offset} // Position in \text{tmp} of where to subtract it.

IF \( q > \text{numupb} \) DO
{ // Reduce both \( p \) and \( q \) so the \( q = \text{numupb} \)
  p := p - (\text{upb}_2 + \text{offset} - \text{numupb})
  q := \text{numupb}
}

// Form the difference in \text{tmp}
\text{copyu}(t_1, u_1, \text{tmp}, \text{numupb})

// Subtract \( n_2 \) suitably shifted from \text{tmp}
WHILE \( p \geq 2 \) DO
{ LET \( x = \text{tmp![q] } - \text{borrow} - \text{t2![p]}
  \text{borrow} := 0
  \text{IF } x < 0 \text{ DO } \text{borrow, } x := 1, x + 10000
  \text{tmp![q] } := x
  p, q := p-1, q-1
}

// There are no more digits of \( n_2 \), but the may still be a borrow
// to deal with
WHILE \( \text{borrow } \& q \geq 2 \) DO
{ LET \( x = \text{tmp![q] } - \text{borrow}
  \text{borrow} := 0

IF x < 0 DO borrow, x := 1, x + 10000
    tmp!q := x
    q := q-1

IF borrow DO
    { // There was a borrow out of the senior radix digit.
        // This is a system error since abs n1 is >= abs n2.
        writef("SYSTEM ERROR: in subu*n")
        abort(999)
        RESULTIS FALSE
    }

standardize(tmp,numupb)

copy(tmp,numupb, n3,upb3) // Set n3 = tmp rounded

    //checknum(n3,upb3)
    RESULTIS TRUE
}

The next function sets it argument to 0.0.

AND setzero(n1, upb1) BE FOR i = 0 TO upb1 DO n1!i := 0

The next function sets n3 to n1 + n2 using signed arithmetic. Both n1 and n2 are assumed to be in standard form. The result is rounded.

AND add(n1,upb1, n2,upb2, n3,upb3) = VALOF
    { // Add signed numbers n1 and n2 placing the rounded result in n3

        LET rc = ?
        LET t = VEC numupb

        //checknum(n1,upb1)
        //checknum(n2,upb2)

        IF n1!2=0 DO
            { copy(n2,upb2, n3,upb3) // n1 is zero
                RESULTIS TRUE
            }

        IF n2!2=0 DO
            { copy(n1,upb1, n3,upb3) // n2 is zero
                RESULTIS TRUE
            }

    }
The next function sets \( n_3 \) to \( n_1 - n_2 \) using signed arithmetic. Both \( n_1 \) and \( n_2 \) are assumed to be in standard form. The result is rounded.
AND sub(n1,upb1, n2,upb2, n3,upb3) = VALOF
{ // Subtract n2 from n1 using signed arithmetic placing
   // the rounded result in n3.
   LET rc = ?
   LET n2sign = n2!0

   //checknum(n1,upb1)
   //checknum(n2,upb2)

   IF n2!2=0 DO
   { copy(n1,upb1, n3,upb3) // n2 is zero
      RESULTIS TRUE
   }

   // n2 is non zero
   n2!0 := ~ n2!0 // Negate n2

   rc := add(n1,upb1, n2,upb2, n3,upb3)
   //checknum(n3,upb3)
   n2!0 := n2sign // Restore the sign of n2

   // No need to call checknum since add has already done so.
   standardize(n3,upb3)

   //checknum(n3,upb3)
   RESULTIS rc
}

The next function sets n3 to n1 * n2 using signed arithmetic. Both n1 and n2 are assumed to be in standard form. The result is rounded.

It does this by clearing an accumulator t1 and the successively adding the product of radix digits from n1 and n2 into appropriate positions in t1. These products are typically larger than 9999 and so a carry operation is performed occasionally to ensure all fraction digits in t1 are in the range 0 to 9999. Notice that if the exponents of n1 and n2 are both zero then the result will have exponent zero. Unless the result is zero the exponent of the result will be the sum of the exponents of the two operands. Having computed the product in t1, this result is copied to n3, rounding it if upb3 is smaller the numupb.

AND mul(n1,upb1, n2,upb2, n3,upb3) = VALOF
{ LET sign = n1!0 XOR n2!0 // Set the sign of the result
   LET e1 = n1!1
   LET e2 = n2!1
   LET exponent = e1 + e2 // Initial exponent
   LET carry = ?
LET t1 = VEC numupb

IF iszero(n1,upb1) | iszero(n2,upb2) DO
{ setzero(n3,upb3)
  RESULTIS TRUE
}

setzero(t1,numupb)

// Neither n1 nor n2 are zero.

// Set the exponents of n1 and n2 to zero
n1!1, n2!1 := 0, 0
// Form the product n1*n2 in t1.
// Both n1 and n2 are less than 1.0 so the product will be less than 1.0.
FOR i = 2 TO upb1 DO
{ // Take each digit of n1
  LET n1i = n1!i
  IF n1i DO
  { LET p, x = ?, ?
    LET jlim = numupb+1-i
    IF jlim > upb2 DO jlim := upb2
    // j is in the range 2 to upb2, but the destination
    // position i+j-1 must be <= numupb, so
    // i+j-1 <= numupb ie j <= numupb+1-i
    // j must also be <= upb2
    FOR j = jlim TO 2 BY -1 DO
      { p := i+j-1 // The destination position
        t1!p := t1!p + n1i * n2!j
      }
    carry := 0
    FOR j = numupb TO 1 BY -1 DO
      { LET x = t1!j + carry
        t1!j := x MOD 10000
        carry := x / 10000
      }
  }
  t1!0 := sign
  t1!1 := t1!1 + exponent
  standardize(t1,numupb)
copy(t1,numupb, n3,upb3)
// Restore the exponents of n1 and n2
n1!1, n2!1 := e1, e2

//checknum(n3,upb3)
RESULTIS TRUE
}

The next function set n1 = k * n1 where k is in the range -9999 to =9999.

AND mulbyk(k, n1,upb1) = VALOF
{ LET sign = n1!0
  LET carry = 0

  standardize(n1,upb1)

  IF n1!2=0 RESULTIS TRUE

  IF k=0 DO
  { setzero(n1,upb1)
    RESULTIS TRUE
  }

  IF k<0 DO n1!0, k := ~n1!0, -k

  // The result sign is correct and k is non zero.
  // Multiply digits from the least significant end
  // dealing with carry.
  FOR i = upb1 TO 2 BY -1 DO
  { LET x = n1!i * k + carry
    n1!i := x MOD 10000
    carry := x / 10000
  }

  IF carry DO
  { // Shift the fractional part to the right one place
    // and adjust the exponent.
    LET lsdig = n1!upb1
    FOR i = upb1-1 TO 2 BY -1 DO n1!(i+1) := n1!i
    n1!1 := n1!1 + 1
    n1!2 := carry
    IF lsdig >= 5000 DO addcarry(n1,upb1)
  }

  //checknum(n1,upb1)
  RESULTIS TRUE
}
The next function sets $n_2 = 1/n_1$. It uses a Newton-Raphson iteration based on finding the value of $x$ for which $f(x) = (1/x - a) = 0$. The slope (or differential) of this function at $x$ is $-1/x^2$ and this leads to the iteration: $x_{n+1} = x_n + x_n(1 - a \times x_n)$. This iteration requires a good initial guess since it is easy to choose a value that causes the iteration to diverge. Luckily we can use ordinary single length division to help provide a good initial guess using the first 8 significant decimal digits of $n_1$. The implementation is otherwise quite straightforward. If $upb_2 < numupb$ the result is rounded.

```plaintext
AND inv(n1,upb1, n2,upb2) = VALOF
{ // Standardize n1 if necessary, then if n1 is zero return FALSE,
    // otherwise standardize set n2 = 1/n1.
    // upb1 is assumed to be > 2.
    LET one = TABLE FALSE, 1, 0001 // The number +1.0
    LET sign = n1!0 // The sign of n1
    LET e = ? // To hold the exponent of n1
    LET elim = -(numupb - 2 - numupb/4)

    LET t1 = VEC numupb
    AND t2 = VEC numupb
    AND t3 = VEC numupb
    AND t4 = VEC numupb

    // checknum(n1,upb1)

    IF n1!2=0 RESULTIS FALSE // Cannot take the inverse of zero.

    IF numcmp(one,2, n1,upb1)=0 DO
    { settok(1, n2,upb2) // The inverse of 1.0 is 1.0.
        RESULTIS TRUE
    }

    e := n1!1
    n1!0, n1!1 := FALSE, 0 // Set n1 to be in the range +0.0001 to +1.0000

    // Select initial guess
    { LET w = n1!2 * 1_0000 + n1!3
        // 10000 <= w <= 99999999
        LET a = muldiv(9999_9999, 1_0000, w)
        setzero(t1,numupb)
        t1!0 := FALSE // Set positive
        t1!1 := 1 // Set the exponent
        t1!2 := a / 1_0000 // and two radix digits
        t1!3 := a MOD 1_0000 // of the initial guess.
    }
```
\{ // Start of Newton-Raphson loop

again:
    \texttt{mul(t1,numupb, n1,upb1, t2,numupb)} \quad \text{// Set } t2 = t1 \times n1
    \texttt{sub(one,2, t2,numupb, t3,numupb)} \quad \text{// Set } t3 := 1 - t1 \times n1
    \texttt{mul(t1,numupb, t3,numupb, t2,numupb)}
    \texttt{add(t1,numupb, t2,numupb, t3,numupb)}
    \texttt{IF t3!1>100 DO}
        \{ // The iteration for $1/n1$ has diverged
            \texttt{newline()}
            \texttt{writef("The iteration for $1/n1$ has diverged")}
            \texttt{writef("n1= "); prnum(n1,upb1)}
            \texttt{abort(999)}
        \}
    \texttt{sub(t3,numupb, t1,numupb, t2,numupb)}
    \texttt{UNLESS iszero(t2,numupb) DO}
        \{ IF t2!1 > elim \texttt{DO}
            \{ \texttt{copy(t3,numupb, t1,numupb)}
                \texttt{GOTO again}
            \}
        \}
    \}

    t3!0, t3!1 := sign, 1-e \quad \text{// Set the sign and exponent of the result}
    \texttt{copy(t3,numupb, n2,upb2)}

    n1!0, n1!1 := sign, e \quad \text{// Restore the sign and exponent of n1}

    \texttt{//checknum(n2,upb2)}
    \texttt{RESULTIS TRUE}\}

The next function divides $n1$ by a single radix digit $k$ leaving the result in $n1$. It does this using, so called, short division which is quite straightforward to implement. The divisor can be negative.

AND divbyk(k, n1,upb1) = VALOF
\{ \texttt{LET sign, carry = ?, 0}
    \texttt{LET e = n1!1}
    \texttt{standardize(n1,upb1)}

    \texttt{IF k=0 \quad RESULTIS FALSE}
    \texttt{IF n1!2=0 \quad RESULTIS TRUE}

sign := n1!0

IF k<0 DO sign, k := ~sign, -k

FOR i = 1 TO upb1-1 DO
{ LET x = carry*10000 + n1!(i+1)
  n1!i := x / k
  carry := x MOD k
}
n1!upb1 := carry

TEST n1!1
THEN FOR i = upb1-1 TO 1 BY -1 DO n1!(i+1) := n1!i
ELSE e := e-1

n1!0 := sign
n1!1 := e

//checknum(n1,upb1)
RESULTIS TRUE
}

The next function sets \( n3 = n1 / n2 \) using signed arithmetic. After dealing with the special case of \( n1=0 \), it does this by first calculating the inverse of \( n2 \) and then multiplying it by \( n1 \). This method is used since there is a good Newton-Raphson iteration to calculate an inverse but not for a general division.

AND div(n1,upb1, n2,upb2, n3,upb3) = VALOF
{ LET t1 = VEC numupb
  LET t2 = VEC numupb

  //checknum(n1,upb1)
  //checknum(n2,upb2)

  IF n1!2=0 DO
  { setzero(n3,upb3) // If n1 is zero the result is zero
   RESULTIS TRUE
  }

  IF n2!2=0 RESULTIS FALSE // Cannot divide by zero

  inv(n2,upb2, t1, numupb) // t1 = 1/n2
  mul(n1,upb1, t1, numupb, t2, numupb) // t2 = n1 * 1/n2
  copy(t2, numupb, n3, upb3) // n3 = t2 rounded
}
The next function calculates the square root of \( n_1 \) leaving the result in \( n_2 \). It uses a Newton-Raphson iteration based on finding the value of \( x \) that causes \( f(x) = x^2 - a \) to be zero. The differential (the slope at \( x \)) of this function \( 2x \) and this leads to the iteration: 

\[
x_{n+1} = \frac{x_n + a/x_n}{2}
\]

After dealing with the simple special case of \( n_1=0 \), the function remembers the exponent of \( n_1 \) in \( e \) then sets \( n_1!1 \) to zero causing \( n_1 \) to be in range 0.0001 to 0.9999. A reasonable initial guess is then chosen based on the first 8 decimal digits of \( n_1 \) and the iteration started. Once the iteration is complete, the exponent of the result is set. As can be seen special care is needed if the original exponent was an odd number. The final call of copy rounds the result, if necessary.

```bcpl
AND sqrt(n1,upb1, n2,upb2) = VALOF
{ // Set n2 to the square root of n1.
  LET rc, prevrc = ?, -2
  LET e = ?
  LET elim = -(numupb - 2 - numupb/4)
  LET t1 = VEC numupb
  AND t2 = VEC numupb
  AND t3 = VEC numupb

  setzero(n2,upb2)

  IF iszero(n1) RESULTIS TRUE // sqrt(0) = 0

  standardize(n1,upb1)
  // n1!2 will certainly be non zero

  IF n1!0 RESULTIS FALSE // n1 must be positive

  e := n1!1 // Remember the exponent of n1 in e
  n1!1 := 0 // Cause n1 to be in range 0001 to 9999

  // n1 is greater than zero

  { // Choose a reasonable initial guess
    LET a = n1!2 * 10000 + n1!3 // 0001 <= a <= 9999_9999
    LET guess = 100_0000

    UNTIL muldiv(guess, guess, 1_0000_0000) >= a DO
      guess := guess + guess
  }
}
guess := (guess + muldiv(a, 1_0000_0000, guess)) >> 1

// Place the initial guess in t1
setzero(t1,numupb)
  t1!2, t1!3 := guess/10000, guess MOD 10000
}

setzero(t2,numupb)
setzero(t3,numupb)

// Start of Newton=Raphson sqrt loop
again:
  div(n1,upb1, t1,numupb, t2,numupb) // t2 = n1/t1
  add(t1,numupb, t2,numupb, t3,numupb) // t3 = t1 + n1/t1
  divbyk(2, t3,numupb) // Set t3 := (t1 + n1/t1)/2

  sub(t3,numupb, t1,numupb, t2,numupb)
  UNLESS iszero(t2,numupb) DO
  { IF t3!1 + elim < t2!1 DO
    { copy(t3,numupb, t1,numupb)
      GOTO again
    }
  }
}

  t3!1 := e>=0 -> (e+1)/2, (e-1)/2
  n1!1 := e // Restore the exponent of n1
  UNLESS (e&1)=0 TEST e>0 THEN divbyk(100, t3,numupb)
  ELSE mulbyk(100, t3,numupb)

  copy(t3,numupb, n2,upb2)

  //checknum(n2,upb2)
  RESULTIS TRUE
}

The next function set n2 = n1 rounding if necessary.

AND copy(n1,upb1, n2,upb2) = VALOF
{ LET p = upb1
  IF p > upb2 DO p := upb2
FOR i = 0 TO p DO n2!i := n1!i
FOR i = p+1 TO upb2 DO n2!i := 0 // Pad with zeroes

IF p>upb2 & n1!(upb2+1) > 5000 DO addcarry(n2,p)

IF n2!2=0 RESULTIS standardize(n2,upb2)
//checknum(n2,upb2)
RESULTIS TRUE
}

The next function sets \( n2 = \text{ABS } n1 \), rounding if necessary.

AND copyu(n1,upb1, n2,upb2) = VALOF
{ LET p = upb1
  IF p > upb2 DO p := upb2

  FOR i = 1 TO p DO n2!i := n1!i
  FOR i = p+1 TO upb2 DO n2!i := 0 // Pad with zeroes

  IF p>upb2 & n1!(upb2+1) > 5000 DO addcarry(n2,p)

  n2!0 := FALSE // Set the result to be positive
  IF n2!2=0 RESULTIS standardize(n2,upb2)
  //checknum(n2,upb2)
  RESULTIS TRUE
}

The next function is mainly used internally in the \texttt{arith} library to help implements rounding. It adds one at position \( p \) of the fraction digits of \( n1 \). Note that if \( p \) is less than the upperbound of \( n1 \), the radix digits from position \( p+1 \) to the end are not changed.

AND addcarry(n1,p) = VALOF
{ FOR i = p TO 2 BY -1 DO
  { LET x = n1!i
    UNLESS x = 9999 DO { n1!i := x+1; RESULTIS TRUE }
    n1!i := 0
  }

  // There is a carry out of the senior digit position.
  // This can only happen if 1 was added to 9999 9999 .. 9999
  // so n1!2 to n1!upb1 are all zero.
  n1!2 := 0001
  n1!1 := n1!1 + 1 // Correct the exponent
5.14. A LIBRARY FOR HIGH PRECISION ARITHMETIC

//checknum(n1,p)
RESULTIS TRUE
}

The next function standardizes the high precision number it is given. It either sets n1 to zero or ensures that n1!2 is non zero, shifting the fraction digits and adjusting the exponent, if necessary.

AND standardize(n1,upb1) = VALOF
{ LET p = 2
UNTIL p>upb1 | n1!p DO p := p+1

IF p>upb1 DO
{ // The number is zero if every radix digit is zero.
  n1!0, n1!1 := FALSE, 0 // Other elements are already zero.
  RESULTIS TRUE
}

UNLESS p=2 DO
{ // Shift the fractional part to the left
  // and adjust the exponent.
  FOR i = p TO upb1 DO n1!(2+i-p) := n1!i
  FOR i = upb1-p+1 TO upb1 DO n1!i := 0
  n1!1 := n1!1 - p + 2 // Correct the exponent
}

RESULTIS TRUE
}

The next function sets d to the radius of a sphere centred at the origin that has point p on its surface. If p represents the point (x,y,z) then d is given the value representing $\sqrt{x^2 + y^2 + z^2}$.

AND radius(p,upb1, d,upb2) = VALOF
{ LET t1 = VEC numupb
  LET t2 = VEC numupb
  LET t3 = VEC numupb
  LET t4 = VEC numupb
  LET t5 = VEC numupb

  UNLESS mul(p!0,upb1, p!0,upb1, t1,numupb) RESULTIS FALSE
  UNLESS mul(p!1,upb1, p!1,upb1, t2,numupb) RESULTIS FALSE
  UNLESS mul(p!2,upb1, p!2,upb1, t3,numupb) RESULTIS FALSE
  UNLESS add(t1,numupb, t2,numupb, t4,numupb) RESULTIS FALSE
The function `inprod`, defined below, computes the inner product of two 3D vectors `dir1` and `dir2` leaving the result in `n3`. The components of `dir1` and `dir2` have upperbounds `upb1` and `upb2`, respectively. If `dir1` and `dir2` represent \((a, b, c)\) and \((x, y, z)\), respectively, the result placed in `n3` represents \(ax + by + cz\). As will be shown on page 465, if \((a, b, c)\) and \((x, y, z)\) are direction cosines, the result is the cosine of the angle between them.

\[
\begin{align*}
\text{AND inprod}(dir1, upb1, dir2, upb2, n3, upb3) &= \text{VALOF} \\
\{ & // dir1 and dir2 are 3D vectors. \\
// ie dir1 -> [dx1, dy1, dz1] and dir2 -> [dx2, dy2, dz2] where \\
// upb1 is the upperbounds of dx1, dy1 and dz1 \\
// upb2 is the upperbounds of dx2, dy2 and dz2 \\
// n3 is set to the dx1*dx2+dy1*dy2*dz1*dz2 \\
// If dir1 and dir2 represent direction cosines, n3 will be the \\
// cosine of the angle between them. \\
\text{LET t1} &= \text{VEC numupb} \\
\text{AND t2} &= \text{VEC numupb} \\
\text{AND t3} &= \text{VEC numupb} \\
\text{AND t4} &= \text{VEC numupb} \\
\text{mul}(dir1!0, upb1, dir2!0, upb2, t1, numupb) \\
\text{mul}(dir1!1, upb1, dir2!1, upb2, t2, numupb) \\
\text{mul}(dir1!2, upb1, dir2!2, upb2, t3, numupb) \\
\text{add}(t1, numupb, t2, numupb, t4, numupb) \\
\text{add}(t3, numupb, t4, numupb, n3, upb3) \\
\text{RESULTIS TRUE}
\}
\end{align*}
\]

The function `crossprod`, defined below, calculates the cross product of two 3D vectors `dir1` and `dir2` leaving the result in `dir3`. The upperbounds of the components of these three vectors are `upb1`, `upb2` and `upb3`, respectively.

If `dir1` and `dir2` represent \((a, b, c)\) and \((x, y, z)\), respectively, then the components of `dir3` are set to to represent \((bz - cy, cx - az, ay - bx)\). The direction of `dir3` will be orthogonal to the plane specified by `dir1` and `dir2`, and its length will be the product of the lengths of `dir1` and `dir2` multiplied by the sine of the angle between them. As a special case, if `dir1 = (1, 0, 0)` and `dir2 = (0, 1, 0)`, then `dir3` will represent \((0, 0, 1)\).
5.14. A LIBRARY FOR HIGH PRECISION ARITHMETIC

AND crossprod(dir1,upb1, dir2,upb2, dir3,upb3) = VALOF
{ LET t1 = VEC numupb
  AND t2 = VEC numupb

  mul(dir1!1,upb1, dir2!2,upb2, t1,numupb) // t1 = bz
  mul(dir1!2,upb1, dir2!1,upb2, t2,numupb) // t2 = cy, cx-az and ay-bx,
  sub(t1,numupb, t2,numupb, dir3!0,upb3) // dir3!0 = bz-cy

  mul(dir1!2,upb1, dir2!0,upb2, t1,numupb) // t1 = cx
  mul(dir1!1,upb1, dir2!2,upb2, t2,numupb) // t2 = az
  sub(t1,numupb, t2,numupb, dir3!1,upb3) // dir3!1 = cx-az

  mul(dir1!0,upb1, dir2!1,upb2, t1,numupb) // t1 = ay
  mul(dir1!1,upb1, dir2!2,upb2, t2,numupb) // t2 = bx
  sub(t1,numupb, t2,numupb, dir3!2,upb3) // dir3!0 = ay-bx

  RESULTIS TRUE
}

The function normalize, defined below, converts an arbitrary 3D vector to
one of unit length pointing in the same direction. Its implementation is straight-

AND normalize(dir, upb) = VALOF
{ // This function causes a 3D vector dir to be scaled to make it
 // unit length. dir!0, dir1 and dir!2 are the three components
 // of the vector and each have upprbound upb. It is implemented
 // dividing these numbers by radius(dir!0,upb, dir!1,upb, dir!2,upb)
 // The resulting values are often call direction cosines.
 LET d = VEC numupb
 LET t = VEC numupb

 UNLESS radius(dir,upb, d,numupb) RESULTIS FALSE

 IF iszero(d,numupb) DO
 { settok(1, dir!0,upb)
   setzero(dir!1,upb)
   setzero(dir!2,upb)
   //writef("Set dir to (1,0,0) since dir was too small\n"
   RESULTIS TRUE
 
 }

 UNLESS div(dir!0,upb, d,numupb, t,numupb) RESULTIS FALSE
 copy(t,numupb, dir!0,upb)
The function \texttt{exptok}, defined below, computes \( n_1^k \) by the reasonably efficient method described on page 65.

\begin{verbatim}
AND exptok(k, n1,upb1, n2,upb2) BE
{ // Set n2 to n1^n rounded where n is an integer >= 0.
    LET P = VEC numupb
    AND R = VEC numupb
    AND T = VEC numupb

    copy(n1,upb1, P,numupb) // To hold the next power of n1
    settok(1, R,numupb)     // To hold the result

    WHILE k DO
        IF (k & 1)>0 DO
            // Set R = R * P ie multiply R by the current power of n1
            mul(R,numupb, P,numupb, T,numupb)
            copy(T,numupb, R,numupb)
        } // Set P to P * P
    mul(P,numupb, P,numupb, T,numupb)
    copy(T,numupb, P,numupb)
    k := k>>1
    }
    copy(R,numupb, n2,upb2)
}
\end{verbatim}

The function \texttt{checknum}, defined below, is primarily a debugging aid that checks that its argument is valid standardized number.

\begin{verbatim}
AND checknum(n1,upb1) BE
{ // The calls abort(999) if n1 is not in standard form.
    LET sign, e, d1 = n1!0, n1!1, n1!2
    UNLESS sign=TRUE | sign=FALSE DO
        { writef("n1 has a bad sign field\n")
            writef("n1= "); prnum(n1,upb1)
        }
}
\end{verbatim}
5.14.1 A Simple Example

The **arith** library was developed and tested using a variant of the program in `bcplprogs/tests/testarith.b`, but rather than describing this program, a more interesting program called `fastfib.b` will be presented. This program exercises most of the **arith** library including particularly the functions **div**, **sqrt** and **exptok**. It computes high precision Fibonacci numbers and can be used to find the position of the first Fibonacci number that has 1000 decimal digits which corresponds to problem 25 in the interesting collection of over 500 somewhat mathematical programming problems set in [www.ProjectEuler.net](http://www.ProjectEuler.net). These problems range from being quite simple to extremely challenging, and are well worth looking at. This program is as follows.

```
GET "libhdr"

MANIFEST {
    ArithGlobs=350
    numupb = 2+250+10 // Size of numbers used in the library,
                     // good for 1000 decimals 40 check digits.
    nupb = numupb-5 // Size of numbers used in this program,
                    // allowing 5 guard digits
}
GET "arith.h"
GET "arith.b"    // Get the high precision library

GLOBAL {
    tracing: ug

    root5
    invroot5
    P    // To hold (1+sqrt(5))/2
    Q    // To hold (1-sqrt(5))/2
    One  // To hold 1.0
    pos  // Position of the fibonacci number
    F    // To hold fib(pos)
    T1   // Temp value
}

LET start() = VALOF
{ LET argv = VEC 50

    UNLESS rdargs("n/N,-t/S", argv, 50) DO
    { writef("Bad arguments*n")
        RESULTIS 0
    } 

    pos := 5
    IF argv!0 DO pos := !(argv!0) // pos/N
    tracing := argv!1        // -t/S

    root5 := getvec(nupb)
    invroot5 := getvec(nupb)
    P := getvec(nupb)
    Q := getvec(nupb)
    F := getvec(nupb)
    One := getvec(nupb)
    T1 := getvec(nupb)

    settok(5, T1,nupb)
    sqrt(T1,nupb, root5,nupb)
    inv(root5,nupb, invroot5,nupb)

    IF tracing DO
    { writef("root5=*n")
        prnum(root5,nupb)
        // Check root5
        mul(root5,nupb, root5,nupb, T1,nupb)
writef("root5^2=n"); prnum(T1,nupb)
newline()

// Check invroot5
writef("invroot5=n")
prnum(invroot5,nupb)
mul(invroot5,nupb, invroot5,nupb, T1,nupb)
writef("invroot5^2=n")
prnum(T1,nupb)
}

settok(1, One,nupb)

// Set P to (1 + sqrt(5))/2
add(One,nupb, root5,nupb, P,nupb)
divbyk(2, P,nupb)
IF tracing DO
{ writef("P = (1 + sqrt(5))/2 =n")
  prnum(P,nupb)
  newline()
}

// Set Q to (1 - sqrt(5))/2
sub(One,nupb, root5,nupb, Q,nupb)
divbyk(2, Q,nupb)
IF tracing DO
{ writef("Q = (1 - sqrt(5))/2 =n")
  prnum(Q,nupb)
  newline()
}

//writef("Calling fib(%n, F,%n)*n", pos, nupb)
fib(pos, F,nupb) // Compute fibonacci of pos
writef("fib(%n) =n", pos)
prnum(F,nupb)

{ LET k, d1 = 4*F!1-4, F!2
  UNTIL d1 = 0 DO k, d1 := k+1, d1/10
  IF k<0 DO k := 1
  writef("Number of decimal digits: %n*n", k)
}

freevec(root5)
freevec(invroot5)
freevec(One)
As can be seen it computes \( \text{fib}(n) \) using the formula derived on page 64, namely:

\[
\text{fib}(n) = \frac{(1+\sqrt{5})^n - (1-\sqrt{5})^n}{2^n \sqrt{5}}.
\]
When this program is run with argument 4782 it generates the following output.

0.000> fastfib 4782
fib(4782) =
+0.1070 0662 6638 2758 9367 6498 0584 4573 9688 5083
   6838 9663 2151 6650 1323 5203 3753 1452 0604 6940
   4062 1889 1475 8248 9792 6578 0469 4888 1775 9195
   7484 3364 6667 2569 9595 1299 6030 4612 6274 8092
   4821 8614 4069 4330 5123 4774 4427 5027 3781 7530
   8757 9391 6661 9214 9259 1867 5955 3966 4228 3714
   8943 1130 7469 9503 4395 4700 1985 4326 0972 3067
   2901 9287 0526 4472 4372 6117 7158 2182 5548 4911
   2052 5013 2014 7861 2965 9313 8179 2235 5596 5745
   2039 5061 3755 1467 8375 4322 9119 6021 2993 4048
   2607 0617 5397 7068 4706 8202 8954 8690 2666 1854
   3512 4521 9003 6948 0641 3574 4747 0911 7076 1976
   6945 6910 7009 8024 3934 3961 7474 1037 3691 2503
   2313 6553 2164 7736 9702 3167 7550 5159 5173 5184
   6057 9954 9194 1096 7778 3732 2966 5796 5816 4651
   3903 4881 5425 6310 1842 2419 0259 8460 8800 0110
   1862 5555 0245 4939 3711 3651 6570 3944 7629 5847
   1454 8523 4259 5042 8582 4253 0608 3544 4354 2821
   2611 0089 9286 3795 0480 0689 4330 3097 7321 7834
   8645 4311 3205 7656 5986 8456 2886 1680 8718 6938
   3529 7350 6439 8629 7640 6600 0072 3562 9179 0520
   7051 1640 7761 4812 4918 8583 0945 9405 6668 8339
   1093 5094 4456 5763 5766 6151 6193 1775 3792 8916
   6158 1327 1596 1687 7487 9838 2182 0492 5203 4847
   3874 3847 3677 1934 5127 8702 9218 6362 5062 7816
   0000 0000 0000 0000 0000 0000 E250
Number of decimal digits: 1000
1.550>

Although this program may be good for really large fibonacci numbers with perhaps a million digits, the following naive program (fib1000.b) is much faster for a mere 1000 digits.

/*
This program finds the position of the first fibonacci number having 1000 decimal digits. The first fibonacci number has position zero.
Ie fib(0)=0, fib(1)=1, fib(2)=1, fib(3)=2, fib(4)=3, fib(5)=5, etc

This is a naive implementaion using vectors of digits of radix 100_00_000.
*/
GET "libhdr"

MANIFEST {
  radix = 100_000_000
  digs  = 1000      // Number of decimal digits
  upb  = digs / 8   // 8 decimal digits per word
}

LET start() = VALOF
{ LET a = getvec(upb)
  AND b = getvec(upb)
  LET upba, upbb = ?, ?
  LET t, upbt = ?, ?
  LET n = ?
  LET w = 1
  LET k = (digs-1) / 8
  FOR i = 1 TO (digs-1) MOD 8 DO w := 10*w

      // Set a=0 and b=1
      FOR i = 0 TO upb DO a!i, b!i := 0, 0
      b!0 := 1
      upba, upbb := 0, 0
      n := 1 // n is the position of the fibonacci number in b

      { IF b!k >= w BREAK

      // b is greater than a
      upba := add(b,upbb, a,upba) // Set a to b + a
      n := n+1 // n is now the position of the fibonacci number in a
      //pr(n, a, upba)

      // Swap a and b
      t, upbt := a, upba
      a, upba := b, upbb
      b, upbb := t, upbt
      } REPEAT

  writef("The first fibonacci number with %d digits is at position %d*n",
         digs, n)

  freevec(a)
  freevec(b)
  RESULTIS TRUE
}
5.15. **THE AIRY DISK**

This program uses the *arith* library described in the previous section to calculate the diffraction pattern caused by a point source of light at infinity observed by a telescope with an aperture of 100mm and focal length of 1000mm. It does this by considering many rays of light with wave length 550nm passing through an assumed perfect circular objective lens of diameter 100mm causing the wave front to become spherical with a radius of 1000mm centred at the focal point. The effect of rays reaching nearby points on the focal plane are summed taking account of their different phases. The resulting intensities are plotted to show the size of the central spot and the radius of some of the surrounding diffraction rings. The central spot is called the Airy disk named after George Airy who, in 1835, was the first to give a mathematical explanation of this pattern. The mathematical theory states that the radius of the innermost dark ring should be
\[ r = 1.22\lambda \times \frac{F}{A} \]

where \( \lambda \) is the wave length of the light (=550nm)

\( F \) is the focal length (=1000mm)

\( A \) is the aperture (=100mm)

(The ratio \( \frac{F}{A} \) is commonly called the F number of a camera or telescope)

So for the telescope under consideration

\[ r = 1.22 \times (550 \times 10^{-6}) \times 1000/100 = 0.00671 \text{mm} \]

The program confirms this result. The Rayleigh criterion of barely being able to resolve two close stars is when the centre of the airy disk of one of the stars is on the edge of the airy disk of the other. Increasing the magnification of the image will not help.

The program starts as follows.

MANIFEST {
  ArithGlobs=350
  numupb = 2+10 // Allow a maximum precision of about 40 decimal digits.
}

GET "libhdr"
GET "arith.h" // Insert the arith high precision library
GET "arith.b"

GET "sdl.h" // Insert the SDL library
GET "sdl.b"

GLOBAL {
  stdin:ug
  stdout

  tracing

  // Colours
  c_black
  c_white
  c_gray
  c_blue
  c_red

  screen // For the SDL graphics
  fmt // The graphics format

  // All numerical values use high precision numbers.
The $z$ axis is the axis of the telescope in direction from the objective lens towards the mirror. The $x$ axis is to the right when viewing the objective in the $z$ direction, and $y$ is upwards. The origin is on the $z$ axis in the plane of the thin perfect objective lens. The focal point has coordinates $(0, 0, F)$, where $F$ is 1000mm.

The vectors $pvx$, $pvy$ and $pvz$ hold the coordinates of thousands of points on the spherical wave front touching the objective lens. Each point is derived from a lattice point in the plane of the objective lens within 50mm of the $z$ axis.

The program continues as follows.

MANIFEST {
  pvupb = 101*100  // UPB of the vectors holding the coordinated
              // of points on the spherical wave front.
  nupb = 2+8      // The size of most high precision numbers used
              // in the calculation. This setting allows about
              // 32 decimal digits of precision.
  spacevupb = 1000000
  F = 1000        // The focal length
  A = 100         // The aperture
  Ar = A/2        // Objective lens radius
}

LET drawdot(x, y) BE
{ // Draw a 3x3 dot at (x,y) relative to (centrex,centrey).
// This function is used to plot points on the graph.
LET sx = centrex + x
LET sy = centrey + y
drawfillrect(sx-1, sy-1, sx+1, sy+1)
updatescreen()
}

LET initscreen() BE
{ initsdl()
mkscreen("Airy Diffraction Pattern", 800, 400)
// Define some colours
c_black := maprgb( 0, 0, 0)
c_white := maprgb(255, 255, 255)
c_gray := maprgb(200, 200, 200)
c_blue := maprgb( 0, 0, 255)
c_red := maprgb(255, 0, 0)

// Choose the screen position of (0,0)
centrex := screenxsize/2
centrey := 60
writef("screenxsize=%n screenysize=%n", screenxsize, screenysize)
writef("centrex=%n centrey=%n", centrex, centrey)
fillsurf(c_gray)
updatescreen()
}

LET start() = VALOF
{ LET argv = VEC 50

stdin := input()
stdout := output()
UNLESS rdargs("-t/s", argv, 50) DO
{ writef("Bad arguments for airy*n")
RESULTIS 0
}

tracing := argv!0 // -t/s
inizscreen()

spacev := getvec(spacevupb)
spacet := spacev+spacevupb
5.15. THE AIRY DISK

```plaintext
spacep := spacet

mkfront()
drawgraph()

freevec(spacev)

writef("Space used = %n out of %n\star %n", spacet-spacep, spacevupb)
RESULTIS 0
}

AND newvec(upb) = VALOF
{ LET p = spacep - upb - 1
  IF p<spacev DO
    { writef("\star More space needed\star")
      abort(999)
      RESULTIS 0
    }
  spacep := p
  FOR i = 0 TO upb DO p!i := 0
  RESULTIS p
}

AND newnum(upb) = newvec(upb)

The function initscreen creates a suitable SDL window that will be used
to draw a graph showing the intensity of points in the focal plane near the z
axis. The function drawdot draws a 3x3 square representing a point on the
intensity curve. The main function start reads the command arguments setting
the variable tracing, but currently this variable is not used. It also allocated
some space using getvec for use by newvec which is mainly used to allocate
the vectors holding high precision numbers. Before returning from start it calls
freevec to return the space obtained by getvec.

AND mkfront() BE
{ // Create the coordinates of all the points on the
  // spherical wave front.
  LET t1 = VEC nupb
  AND t2 = VEC nupb
  AND t3 = VEC nupb

  MANIFEST { step=4 }

  // step controls the number of points chosen in the objective lens.
  // The larger step is the faster the program runs but the resulting
// graph becomes less accurate.

pvcount := 0
pxy := newvec(pvupb) // These will hold the x, y and z coordinated
pvy := newvec(pvupb) // of points on the spherical wave front.
pvy := newvec(pvupb)

FOR x = 0 TO +Ar BY step DO
  FOR y = 0 TO +Ar BY step IF x*x+y*y <= Ar*Ar DO
    // (x,y) are the coordinates of a lattice point in the plane of
    // the objective lens within a distance Ar from the z axis.
    LET Fnum = TABLE FALSE, 1, F // F as a high precision number.
    (x,y,0) is a point in the first quadrant of the objective lens
    LET dx = VEC nupb
    AND dy = VEC nupb
    AND dz = VEC nupb

    LET nx, ny, nz = ?, ?, ? // Three will hold the x,y and z
      // coordinates of a point on the
      // spherical wave front.

    // writef("*nx=%i3 y=%i3*n", x, y)
    // abort(1000)
    settok(x, dx,nupb) // Direction of the line from the focal point
    settok(y, dy,nupb) // to the point (x,y,0) in the objective lens.
    settok(-1000, dz,nupb)

    // writef("dx= "); prnum(dx, nupb)
    // writef("dy= "); prnum(dy, nupb)
    // writef("dz= "); prnum(dz, nupb)
    normalize(@dx,nupb)

    // writef("*nAfter normalisation we have directions cosines*n")
    // writef("dx= "); prnum(dx, nupb)
    // writef("dy= "); prnum(dy, nupb)
    // writef("dz= "); prnum(dz, nupb)
    // newline()
    // (dx,dy,dz) is now a unit vector in direction focal point to (x,y,0)

    // Multiply dx by F
    mulbyk(F, dx,nupb)
    // writef("Multiply dx by F where F=%n*n", F)
// writef("dx= "); prnum(dx,nupb)
// newline()

// Multiply dy by F
mulbyk(F, dy,nupb)
// writef("Multiply dy by F where F=%n*n", F)
// writef("dy= "); prnum(dy,nupb)
// newline()

mulbyk(F, dz,nupb)
// writef("Multiply dz by F where F=%n*n", F)
// writef("dz= "); prnum(dz,nupb)
// newline()

// Add the coordinates (0,0,1000) of the focal point
add(Fnum,2, dz,nupb, t1,nupb)
copy(t1,nupb, dz,nupb)
// writef("Add F to the z coordinate where F=%n*n", F)
// writef("dz= "); prnum(dz,nupb)

// (dx,dy,dz) is now a point on the spherical wave front in
// the first quadrant.

nx := newnum(nupb)  // Allocate the numbers to hold the
ny := newnum(nupb)  // x, y and z coordinates of a point
nz := newnum(nupb)  // on the spherical wave front.

copy(dx,nupb, nx,nupb)
copy(dy,nupb, ny,nupb)
copy(dz,nupb, nz,nupb)

// Store these coordinates in pvx, pvy and pvz.
pvcount := pvcount+1
pvx!pvcount := nx  // A point in the first quadrant
pvy!pvcount := ny  // ie nx>=0 and ny>=0
pvz!pvcount := nz
// writef("Wave front point %i4 for (%i3,%i3)*n", pvcount, x,y)
// writef("x= "); prnum(nx, 4)
// writef("y= "); prnum(ny, 4)
// writef("z= "); prnum(nz, 4)

IF x=0 & y=0 LOOP
UNLESS x=0 DO
{ nx := newnum(nupb)  // Allocate the numbers to hold the
ny := newnum(nupb)  // x, y and z coordinates of a point
nz := newnum(nupb)  // on the spherical wave front.

copy(dx,nupb, nx,nupb)
copy(dy,nupb, ny,nupb)
copy(dz,nupb, nz,nupb)

nx!0 := TRUE        // Negate just x -- second quadrant
pvcount := pvcount+1
pvx!pvcount := nx    // A point in the second quadrant
pvy!pvcount := ny    // ie nx<0 and ny>=0
pvz!pvcount := nz
//writef("Wave front point %i4 for (%i3,%i3)*n", pvcount, -x,y)
//writef("x= "); prnum(nx, 4)
//writef("y= "); prnum(ny, 4)
//writef("z= "); prnum(nz, 4)
}

IF x>0 & y>0 DO
{ nx := newnum(nupb)  // Allocate the numbers to hold the
ny := newnum(nupb)  // x, y and z coordinates of a point
nz := newnum(nupb)  // on the spherical wave front.

copy(dx,nupb, nx,nupb)
copy(dy,nupb, ny,nupb)
copy(dz,nupb, nz,nupb)

nx!0, ny!0 := TRUE, TRUE // Negate x and y -- third quadrant
pvcount := pvcount+1
pvx!pvcount := nx    // A point in the third quadrant
pvy!pvcount := ny    // ie nx<0 and ny<0
pvz!pvcount := nz
//writef("Wave front point %i4 for (%i3,%i3)*n", pvcount, -x,-y)
//writef("x= "); prnum(nx, 4)
//writef("y= "); prnum(ny, 4)
//writef("z= "); prnum(nz, 4)
}

IF x>=0 & y>0 DO
{ nx := newnum(nupb)  // Allocate the numbers to hold the
ny := newnum(nupb)  // x, y and z coordinates of a point
nz := newnum(nupb)  // on the spherical wave front.

copy(dx,nupb, nx,nupb)
copy(dy,nupb, ny,nupb)
This function considers every grid point in the plane of the objective lens that is no more than $R$ (50mm) from the $z$ axis. For each point it constructs a line to the focal point and computes the coordinates of the point on this line that is 1000mm from the focal point. These coordinates are placed in the vectors $pvx$, $pvy$ and $pvz$. Since the telescope is symmetric about the $z$ axis, the computation is only done for points in the first quadrant (when $x \geq 0$ and $y \geq 0$). The coordinates of points on the wave front in the other quadrants just involve sign changes.

You can imagine the result is a circular disc with a shallow spherical depression uniformly covered with thousands of point sources of light, and since they are on the wave front they will all be in phase.

The next function `drawgraph` draws the graph showing the intensity of the resulting image at points in the focal plane close to the $z$ axis.

```plaintext
AND drawgraph() BE
{
    moveto(0, centrey)
drawto(screenxsize, centrey)
moveto(centrex, 0)
drawto(centrex, screenysize)

    setcolour(c_black)
    moveto(centrex- 671*3/10, centrey-20)
drawto(centrex- 671*3/10, centrey+20)
plotf (centrex- 671*3/10 - 40, centrey-40, "-0.00671mm")

    moveto(centrex+ 671*3/10, centrey-20)
}
}
CHAPTER 5. INTERACTIVE GRAPHICS IN BCPL USING SDL

drawto(centrex+ 671*3/10, centrey+20)
plotf (centrex+ 671*3/10 - 40, centrey-40, "+0.00671mm")

updatescreen()

// Plot the intensity points
FOR r = 0 TO 126 BY 1 DO // r is in units of 0.0001mm
 { LET fx = VEC nupb // To hold the coordinates of a point on the focal
   LET fy = VEC nupb // plane at a distance r from the z axis.
   LET fz = VEC nupb

   LET t1 = VEC nupb
   LET lambda = VEC nupb // To hold the wave length 550nm.
   LET angle = 0
   LET sum = 0

   str2num("0.000550", lambda,nupb) // Average wave length of visible light
   //writef("lambda= "); prnum(lambda, nupb)

   settok(r, fx,nupb)
   UNLESS r=0 DO fx!1 := fx!1 - 1 // Divide fx by 10000
   setzero(fy,nupb)
   settok(1000, fz,nupb)

   // Iterate through all the points on the wave front.
   FOR i = 1 TO pvcount DO
   { LET x = pvx!i // Coordinates of the next point
     LET y = pvy!i
     LET z = pvz!i

     LET dx = VEC nupb
     AND dy = VEC nupb
     AND dz = VEC nupb
     AND d = VEC nupb

     AND diff = VEC nupb // The difference between the length
     // the ray from the selected point on
     // the wave front to the selected point
     // the focal plane.

     //writef("*ni=%i4 r=%7.4dmm*n", i, r)
     //writef("fx= "); prnum(fx, nupb)
     //writef("fy= "); prnum(fy, nupb)
     //writef("fz= "); prnum(fz, nupb)
5.15. THE AIRY DISK

//writef("x= "); prnum(x, nupb)
//writef("y= "); prnum(y, nupb)
//writef("z= "); prnum(z, nupb)

//writef("Calling sub(x,nupb, fx,nupb, dx,nupb)*n")
//writef("x= "); prnum(x, nupb)
//writef("fx= "); prnum(fx, nupb)
sub(x,nupb, fx,nupb, dx,nupb)
sub(y,nupb, fy,nupb, dy,nupb)
sub(z,nupb, fz,nupb, dz,nupb)

//writef("dx= "); prnum(dx, nupb)
//IF i=54 DO abort(5544)
//writef("dy= "); prnum(dy, nupb)
//writef("dz= "); prnum(dz, nupb)
radius(dx,nupb, d,nupb)
//writef("d= "); prnum(d, nupb)

// d is the distance between the selected point on the wave front
// and the selected point in the focal plane.

sub(d,nupb, fz,nupb, diff,nupb)
//writef("diff="); prnum(diff, nupb)
div(diff,nupb, lambda,nupb, t1,nupb)
//writef("t1= "); prnum(t1, nupb)

// t1 is diff divided by the wavelength of light.
// The integer part of t1 is the number of complete wavelengths
// in diff, and the fractional part is the phase represents
// as a number in the range -0.9999 to +0.9999. For digits
// of precision is sufficient so we set angle to the
// first 4 decimal digits after the decimal point. -1.0000
// represents -180 degrees and +1.0000 represents +180 degrees.

angle := -1
IF t1!1= 1 DO angle := t1!3
IF t1!1= 0 DO angle := t1!2
IF t1!1< 0 DO angle := 0

IF angle<0 DO
{ writef("System error: angle too large*n")
 abort(999)
 }


CHAPTER 5. INTERACTIVE GRAPHICS IN BCPL USING SDL

IF t1!0 DO angle := -angle

// angle is in the range -9999 to +9999, representing -180 to +180 degrees.
// writef("fx= "); prnum(fx, nupb)
// writef("t1= "); prnum(t1, nupb)
// writef("angle=%8.4d\n", angle)
// abort(1000)

// We now convert this angle to radians and take the cosine
// which we then convert to a number in the range -1.0000 to +1.0000.
{ LET fangle = sys(Sys_flt, fl_float, angle)
  LET x = sys(Sys_flt, fl_cos, 2.0 ** 3.14159 ** fangle / 1.0000.0)
  LET cosangle = sys(Sys_flt, fl_fix, x ** 10000.0)
  // We add all the cosines
  sum := sum + cosangle
  // writef("%i4/%i4: %8.4d sum=%8.4d\n", i, pvcount, cosangle, sum)
}

// IF i>=53 DO
// IF r=126 DO abort(1001)
}

sum := sum / pvcount // Take the average cosine
sum := sum*sum/10000 // and square it to give the intensity.
writef("r=%7.4d mm intensity= %i6\n", r, sum)
// abort(1001)
setcolour(c_black)
drawdot(+r*3, sum/30) // Plot the resulting two points
drawdot(-r*3, sum/30)
updatescreen()

Strictly speaking, we should take the average of intensities over many different phases of the wave front. But even without doing this the resulting graph is reasonably accurate. Note that the point on the z axis will give the maximum intensity and points at a distance of 0.00671mm where the first dark ring occurs has intensity zero corresponding to an amplitude of zero for all phases of the wave front.

When this program runs it generates the following window showing a graph that confirms that, for a telescope with a 100mm objective lens and a focal length of 1000mm, the airy disk has a radius of 0.00671mm.
5.16 A Catadioptric Telescope

This program is a demonstration of ray tracing through lenses and mirrors. It concentrates on the design of a Hamiltonian catadioptric telescope consisting of a convex objective made of crown glass and a mirror made of flint glass silvered on its back surface. All the surfaces are spherical but the resulting spherical aberration can be minimised by careful placement of the mirror and the choice of the radii of four optical surface. At the same time chromatic aberration can also be minimized. The objective lens and mirror are arranged as follows.

Analysis of optical instruments is renowned for requiring high precision arithmetic, so the arith is used library to perform the calculations to sufficient precision. Currently numbers with about 60 significant decimal digits are used while allowing the library functions to use up to 72. This precision can be changed easily if required.

The program uses a BCPL system that includes the SDL graphics library since it draws an image on the focal plane resulting from point sources of blue and red light light at infinity from directions on or near the axis of the telescope.
For each direction, 17 rays are chosen using different entry points through the objective lens. Since spherical and chromatic aberration cannot be fully corrected the images contain scatterings of blue and red dots. The program attempts to improve the geometry of the telescope iteratively minimising this scattering. The iteration only changes the radii of the spherical front and rear surfaces of the objective lens and the front and rear surfaces of the mirror. From a well chosen initial setting the program can find a near optimum design for the telescope. If we can obtain a design that causes the scattering to be no larger than the size of the corresponding airy disk, further optimization will not improve the resolution of the telescope.

The program is called cataopt.b and starts as follows, declaring all the global variables and manifest constants needed.

```
MANIFEST {
    ArithGlobs=350
    numupb = 2+18  // Allow a maximum precision of about 72 decimal digits
}

GET "libhdr"
GET "arith.h"
GET "arith.b"

GET "sdl.h"
GET "sdl.b"

// Compile the arith library as a separate section.

MANIFEST {
    ArithGlobs=350
    numupb = 2+18  // Allow a maximum precision of about 72 decimal digits
}

GET "libhdr"
GET "arith.h"
GET "sdl.h"

GLOBAL {
    stdin:ug
    stdout

    spotmag  // This specified the magnification of spot
              // drawn by drawdot.
    pausing  // Pause when the geometry improves
    tracing
```
reduced  // =TRUE when the geometry improves

// Colours
c_black
c_white
c_gray
c_blue
c_red

screen // For the SDL graphics
fmt   // The graphics format

R1    // Objective lens front radius
prevR1
C1    // Centre of objective lens front
R2    // Objective lens rear radius
prevR2
C2    // Centre of objective lens rear surface
R3    // Concave mirror front radius
prevR3
C3    // Centre of concave mirror front surface
R4    // Concave mirror silvered surface radius
prevR4
C4    // Centre of concave mirror silvered surface
T1    // Objective thickness
T2    // Mirror thickness

MirrorRadius  // Actual mirror radius

D     // The distance between the objective and mirror.
     // Typically about 700mm
F     // The z coordinate of the focus plane. Typically F=0

deltaR1
deltaR2
deltaR3
deltaR4

factor  // A power of ten used in computing the next delta
initfactor  // This hold the value of the f argument

spotsize  // Current spot size or -1

dist  // Distance from y average to theoretical y centre.
spotvalue  // Set to spotsize+5*dist
bestspotvalue // The best spotvalue so far
bestdist // Set to dist of the best spotvalue.
bestspotsize // Set to spotsize of the best spotvalue.

// Directions at small angles in the y-z plane
dir0cx; dir0cy; dir0cz // Parallel to the telescope axis.
dir1cx; dir1cy; dir1cz // about 1/8 degrees of the axis.
dir2cx; dir2cy; dir2cz // about 1/4 degrees off the axis.

Inx; Iny; Inz // Coordinates of the entry point in plane z=0
Outx; Outy; Outz // Coordinates of the exit point on the front
    // surface of the mirror.
outdircx;outdircy;outdircz // Direction of the out ray

Arad // Radius of the A circle in the objective, typically 50mm
Brad // Radius of the B circle in the objective, typically 25mm

root2 // Two useful conatants
    one

spacev
spacet
spacep

currentline // A byte vector with upb=255, used
    // when reading catageometry.txt

iterations // Number of iterations to perform

spot0vx // Dot coordinates is the focal plane
spot0vy // resulting from point light sources
spot1vx // from three directions, 0, 1/8 and
spot1vy // 1/4 degree from the axis. Note the
spot2vx // moon has a angular radius of
spot2vy // about 1/4 degree.

geometrystream // Used when reading or writing catageometry.txt
    // This holds the latest setting of R1 to R4.

    // The refractive indices of crown and flint glass
    // for both air to glass and glass to air.
crownblueindex; crownblueinvindex
crownredindex; crownredinvindex
flintblueindex; flintblueinvindex
flintredindex; flintredinvindex
5.16. A CATADIOPTRIC TELESCOPE

centrex // Centre of the SDL screen
centrey
}

MANIFEST {

nupb = 2+15 // Size of most high precision numbers, allowing
    // about 60 decimal digits of precision.
    // If p is a number
    // p!0 TRUE is negative, = FALSE otherwise
    // p!1 The exponent, ie the power of 10000 to multiple or
    // divide the fractional part by.
    // p!2 .. p!upb is the fractional part representing a
    // value in the range 0 to 1.0

spacevupb = 10000

Blue=1 // Specifying colours used in raytrace.
Red=2
}

The $z$-axis is the axis of the telescope in direction from the centre of the objective lens towards the mirror. The $x$-axis is to the right when viewing the objective in the $z$ direction, and $y$ is upwards. The origin is on the $z$ axis at the centre of the objective lens. The separation $D$ is the $z$ coordinate of where the silvered surface of the mirror intersects the $z$-axis. The focal plane is at $z=0$ and the setting of $D=700$ allows the telescope to have a focal length of between 700mm and 1400mm. The optimisation process aims for a focal length of 1000mm.

The program continues as follows.

LET drawdot(dir, x, y) BE
{
    // Draw a 3x3 dot at (x,y) relative to (centrex,centrey)
    // x and y are in units of 1/10000mm.
    // dir = 0, 1 or 2 specifying the direction of the
    // incoming ray.
    // Direction 1 the image spot on the focal plane is centred
    // at $x=0$ and $y = -1000/(8*60) = -2.0833$mm
    // For direction 2 the y coordinate is $-4.1666$mm

    LET spotcentrex = centrex
    LET spotcentrey = centrey - muldiv(screenysize, 2_0833*dir, 8_0000)

    // Place the dot relative to the origin
    y := y + 2_0833*dir
}
// Magnify the spot dot
x, y := x * spotmag, y * spotmag

// These are in units of 1/10000mm relative to the spot centre.

// Convert to screen coordinates assuming screensize is
// equivalent to 8mm
x := spotcentrex + (screensize * x) / 8_0000
y := spotcentrey + (screensize * y) / 8_0000

drawfillrect(x-1, y-1, x+1, y+1) // Draw a 3x3 dot

setcolour(c_black)
moveto(spotcentrex-10, spotcentrey)
drawto(spotcentrex+10, spotcentrey)
updatescreen()
IF tracing DO
{ writef("drawdot: x=%n y=%n*n", x,y)
  //abort(1077)
}

The function drawdot, defined above, plots a the point on the focal plane corre-
sponding to a ray through the telescope originating from a blue or red point source
at infinity. The argument dir is 0, 1 or 2 specifies the direction of the incoming ray.
The function also draws a short horizontal line through the y-axis indicating the the-
oretical position of the centre of the image of a star from this direction assuming the
focal length of the telescope is 1000mm. To make the scattering of points more visible
their distance from the theoretical centre is magnified by the factor spotmag whose
default value is 20. It can be changed using the mag option when cataopt is called.
For a reasonable telescope design each image spot has a size of about two pixels when
spotmag=1.

Since cataopt draws and image on the screen using the SDL graphics library, it
must initialise SDL and create a window in which to draw it. This is done by the
function defined below.

LET initscreen() BE
{ initsdl()
mkscreen("Catadioptric", 500, 500)

  // Define some colours
  c_black := maprgb( 0, 0, 0)
c_white := maprgb(255, 255, 255)
The next function start is the main function of cataopt. It reads the command arguments using rdargs and after setting iterations and spotmag and initialising SDL by the call initsdl(). It enters the function telescope to optimise the geometry of the telescope. The f argument overrides the setting t factor that controls the maximum size for the delta values used in choosing another setting of R1 to R4. The -p argument causes the program to pause every time a new setting of R1 to R4 has been processed. The -t argument causes some debugging output to be generated while the program runs. More debugging output can be generated by uncommenting various writef and abort. Since the program allocates space for many high precision numbers, it is convenient to allocate one area for the using getvec. This space is used by newvec and newnum that are defined later.

LET start() = VALOF
{ LET argv = VEC 50
  LET str = VEC 255/bytesperword
  currentline := str

  stdin := input()
  stdout := output()

  UNLESS rdargs("mag/n,n/n,f/n,-p/s,-t/s", argv, 50) DO
  { writef("Bad arguments for cataopt*n")
    RESULTIS 0
  }
}

spotmag := 20
IF argv!0 DO spotmag := !argv!0 // mag/n

iterations := 1000
IF argv!1 DO iterations := !argv!1 // n/n

initfactor := 0
IF argv!2 DO initfactor := !argv!2 // f/n
pausing := argv!3 // -p/s
tracing := argv!4 // -t/s

spacev := getvec(spacevupb)
spacet := spacev + spacevupb
spacep := spacet

UNLESS spacev DO
{ writef("*nERROR: More memory is needed*n")
  RESULTIS 0
}

initscreen()

// Analyse the catadioptric telescope, hopefully optimising its
// geometry.
telescope()

IF spacev DO
{ writef("Space used is %n out of %n*n", spacet-spacep, spacevupb)
  freevec(spacev)
}

RESULTIS 0
}

The program spends most of its time tracing rays of blue and red light through the
telescope. The effect of refraction must be calculated as the ray enters or leaves the
objective lens. It must also deal with the refraction and reflection as the ray passes
through the mirror. Luckily the underlying mathematics is quite simple making the
program easy to write and understand, even though it is quite long.

We must first choose a way to represent a ray. Our selected method is to choose a
point, \( P \), on the ray specified by coordinate \((x, y, z)\) and the direction of the ray using
direction cosines \((u, v, w)\). You will remember that direction cosines represent a 3D
vector of unit length, so \((x, y, z) + (u, v, w)t = (x + ut, y + vt, z + wt)\) represent the
coordinates of a point on the ray at a distance \( t \) from \( P \). In this program, \( P \) is always
either a point in the focal plane \((z = 0)\) or a point on the spherical surface of the lens
or mirror.

Having chosen our representation of a ray, we need a mathematical representation of
a spherical surface. This is simple since every point \((x, y, z)\) on it must be at a constant
distance \( R \) from the centre of the sphere. Assuming the centre is at coordinate \((0, 0, c)\),
the resulting equation is: \( x^2 + y^2 + (z - c)^2 = R^2 \). The centre of each spherical surface
is always on the \( z \) axis, so both its \( x \) and \( y \) coordinates are zero.

We will frequently need to calculate the coordinates of the point where a ray inter-
sects the spherical surface of a lens or mirror. This is easily done by substituting the
coordinates of the point on the ray at distance \( t \) from \( P \) in the equation of the sphere. This gives us the following equation:

\[
(x + vt)^2 + (y + wt)^2 + (z + wt - c)^2 = R^2
\]

which simplifies to the following quadratic in \( t \):

\[
At^2 + Bt + C = 0
\]

where

\[
A = u^2 + v^2 + w^2 = 1 \\
B = 2(xu + yv + (z - c)w) \\
C = x^2 + y^2 + (z - c)^2 - R^2
\]

Since \( A = 1 \), the quadratic equation yields two solutions for \( t \), namely:

\[
t = \frac{-B + \sqrt{B^2 - 4C}}{2} \\
t = \frac{-B - \sqrt{B^2 - 4C}}{2}
\]

The function places the two solutions in the given high precision numbers \( t_1 \) and \( t_2 \), returning TRUE if successful. If the ray does not intersect the sphere the result is FALSE, but in this program this never happens. The definition of intersect is as follows.

```
AND intersect(dir, P, c, r, t1, t2) = VALOF
{
    // This calculates the intersection points of a line and a sphere.
    // dir!0,dir!1,dir!2 are the direction cosines of the line
    // P!0,P!1,P!2 is a point on it.
    // c is the z coordinate of the centre of the lens surface
    // r is the radius of the lens surface.
    // P+t1*dir and P+t2*dir are the intersection points, if any.
    // The result is TRUE if t1 and t2 exist.
    // All the numbers above have upperbound nupb.

    // A point on the line has coordinates
    // x = P!0 + dir!0*t
    // y = P!1 + dir!1*t
    // z = P!2 + dir!2*t

    // These must be on the surface of the sphere, so
    // x^2 + y^2 + (z-c)^2 = r^2

    // This gives a quadratic of the form At^2 + Bt + C = 0

    // where
```
CHAPTER 5. INTERACTIVE GRAPHICS IN BCPL USING SDL

// A = dx^2 + dy%2 + dz^2 = 1
// B = 2(xdx + ydy + (z-c)dz)
// C = x^-2 + y^-2 + (z-c)^-2 - r^-2

// giving the solutions: t = (-B +/- sqrt(B^2 - 4AC))/2A
// and since A=1 t = (-B +/- sqrt(B^2 - 4C))/2

LET x  = P!0
LET y  = P!1
LET z  = P!2
LET dx = dir!0
LET dy = dir!1
LET dz = dir!2
LET tmp1 = VEC numupb
LET tmp2 = VEC numupb
LET tmp3 = VEC numupb
LET tmp4 = VEC numupb
LET tmp5 = VEC numupb
LET B  = VEC nupb
LET C  = VEC nupb

mul(x,nupb, dx,nupb, tmp1,numupb) // tmp1 = x dx
mul(y,nupb, dy,nupb, tmp2,numupb) // tmp2 = y dy
sub(z,nupb, c,nupb, tmp3,numupb) // tmp3 = z - c
mul(tmp3,numupb, dz,nupb, tmp4,numupb) // tmp4 = (z - c)dz

add(tmp1,numupb, tmp2,numupb, tmp3,numupb) // tmp3 = xdx + ydy
add(tmp3,numupb, tmp4,numupb, B,nupb) // B = xdx + ydy + (z - c)dz
mulbyk(2, B,nupb) // B = 2(xdx + ydy + (z - c)dz)

mul(x,nupb, x,nupb, tmp1,numupb) // tmp1 = x^2
mul(y,nupb, y,nupb, tmp2,numupb) // tmp2 = y^2
sub(z,nupb, c,nupb, tmp3,numupb) // tmp3 = z-c
mul(tmp3,nupb, tmp3,numupb, tmp4,numupb) // tmp4 = (z-c)^2
mul(r,nupb, r,nupb, tmp5,numupb) // tmp5 = r^-2
add(tmp1,numupb, tmp2,numupb, tmp5,numupb) // C = x^-2+y^-2
add(C,nupb, tmp4,numupb, tmp1,numupb) // tmp1 = x^-2+y^-2+(z-c)^-2
sub(tmp1,numupb, tmp5,numupb, C,nupb) // C = x^-2+y^-2+(z-c)^-2-r^-2

mul(B,nupb, B,nupb, tmp1,numupb) // tmp1 = B^-2
mulbyk(4, C,nupb) // C = 4C
sub(tmp1,numupb, C,nupb, tmp2,numupb) // tmp2 = B^-2 - 4C
sqrt(tmp2,numupb, tmp3,numupb) //tmp3 = sqrt(B^-2 - 4C)
neg(B,nupb) // B = -B
The next function, \texttt{refract}, takes arguments specifying the inward direction \texttt{(indir)} of a ray, the coordinates of the intersection point \texttt{(point)} on a spherical surface, the \texttt{z} coordinate of the centre of the sphere and the inverse of the refractive index \texttt{(index)} of boundary. The inverse is used since this allows \texttt{mul} to be used instead of \texttt{div} which is more efficient. These quantities allow the function to calculate the direction cosines of the outgoing ray using Snell’s law. This law states that, when a ray of light passes through the boundary between two media such as air to glass, the ratio of the sines of the angles of incidence and refraction is the refractive index of the boundary. The boundary from air to crown glass has a refractive index is about 1.5 but this varies slightly on the wavelength of the light. For the the boundary glass to air the inverse 1/1.5 is used.

The angle on incidence is the angle between the ray and the normal at the point \texttt{P} where the ray intersects the boundary. Normal is a mathematical term for the direction perpendicular to a surface. For all our surfaces the normal is easy to calculate since its direction is from the centre of the spherical surface to \texttt{P}. If \texttt{P} = \((x, y, z)\) and the centre is at \((0, 0, c)\) then a 3D vector in the direction of the normal is \(\langle x, y, z - c \rangle\) and this can be converted to direction cosines by calling \texttt{standardize}. We can calculate the cosine of the angle of incidence \(\theta\) by evaluating the inner product of \texttt{indir} and the normal, but we may have to negate the normal first so that they are both advancing more or less the same direction. Why the inner product of two direction cosines yields the cosine of the angle between them is explained on page 465. Having calculated \(\cos \theta\) we can easily compute \(\sin \theta\) using the formula: \(\cos^2 \theta + \sin^2 \theta = 1\) that was derived on page 270.

Assuming \texttt{P} is the intersection point on the lens surface we can calculate a point \texttt{P1} on the incoming ray one unit from \texttt{P} by subtracting the direction cosines \texttt{indir} from \texttt{P}. We have already calculated \(\cos \theta\) so we can easily calculate the coordinates of a point \texttt{P2} on the normal a distance of \(\cos \theta\) from \texttt{P} and on the same side of the surface as \texttt{P1}. We now have a right angled triangle with vertices \texttt{P}, \texttt{P1} and \texttt{P2}, and the length of the edge from \texttt{P1} to \texttt{P2} is \(\sin \theta\). Snell’s law tells us that \(\sin \phi\), where \(\phi\) is the angle of refraction, equals \(\sin \theta\) divided by the refractive index. We can easily construct a triangle with vertices \texttt{P}, \texttt{Q1} and \texttt{Q2} in the same plane as the first triangle \texttt{P−P1−P2} with \texttt{Q2} on the normal at a distance \(\cos \phi\) from \texttt{P}, and \texttt{Q1} chosen to be on the line through \texttt{Q2} parallel to \texttt{P1−P2} with the length of \(\texttt{Q1−Q2}\) equal to \(\sin \phi\). The triangle \texttt{P−Q1−Q2} is thus a right angled triangle with the outgoing ray lying along \texttt{P−Q1}, and since
$P - Q1$ is already of unit length the components are the direction cosines of the refracted ray. The definition of \texttt{refract} is as follows.

\begin{verbatim}
AND refract(indir, P, c, invindex, outdir) = VALOF
{
  // indir!0, indir!1, indir!2 are the direction cosines
  // of the in ray.
  // P!0, P!1, P!2 are the coordinates of the entry
  // point P on the lens surface.
  // c is the z coordinate of the lens surface centre.
  // invindex is the inverse of the refractive index air to glass.
  // outdir!0, outdir!1, outdir!2 will be the direction cosines
  // of the out ray.
  // All numbers have upper bound nupb.
  // The result is TRUE if refract is successful.

  LET indx = indir!0 // The direction cosines of the in ray.
  AND indy = indir!1
  AND indz = indir!2

  LET Px = P!0 // The coordinates of the entry point.
  AND Py = P!1
  AND Pz = P!2

  LET outdx = outdir!0 // To hold the direction cosines of the out ray.
  AND outdy = outdir!1
  AND outdz = outdir!2

  LET ndx = VEC nupb // The surface normal direction cosines
  AND ndy = VEC nupb // with the same z sign as for indir.
  AND ndz = VEC nupb

  AND costheta = VEC numupb // The in ray
  AND sinheta = VEC numupb
  AND cosphi = VEC numupb // The out ray
  AND sinphi = VEC numupb

  LET P1x = VEC nupb // The point P1 on the in ray
  AND P1y = VEC nupb // costheta away from P
  AND P1z = VEC nupb

  LET P2x = VEC nupb // The point P2 on the in normal
  AND P2y = VEC nupb // at distance 1 from P
  AND P2z = VEC nupb

  LET Q1x = VEC nupb // The point Q1 on the out normal
  AND Q1y = VEC nupb // cosphi away from P

\end{verbatim}

\end{verbatim}
5.16. A CATADIOPTRIC TELESCOPE

AND Q1z = VEC nupb

LET Q2x = VEC nupb // The point Q2 on the out ray
AND Q2y = VEC nupb // at distance 1 from P
AND Q2z = VEC nupb

// Note that P-P1-P2 and P-Q1-Q2 are both right
// angle triangles lying in the same plane, and that
// P1-P2 has length sintheta and Q1-Q2 has length sinphi.
// By Snell's law, the ratio of these lengths is the
// refractive index.
// ie sintheta = sinphi / index = sinphi * invindex

AND tmp1 = VEC numupb
AND tmp2 = VEC numupb
AND tmp3 = VEC numupb
AND tmp4 = VEC numupb

copy(Px,nupb, ndx,nupb)
copy(Py,nupb, ndy,nupb)
sub (Pz,nupb, c,nupb, ndz,nupb)
// (Px,Py,Pz are the coordinates of
// the entry point on the lens.
// (ndx,ndy,ndz) is a vector parallel to the normal.
// Ensure that the normal and indir have the same z sign,
// negating (ndx,ndy,ndz) if necessary.
UNLESS indz!=0=ndz!=0 DO
  { // The z signs are different, so negate the normal.
    neg(ndx)
    neg(ndy)
    neg(ndz)
  }

normalize(@ndx,nupb) // Direction cosines of the normal
// (ndx,ndy,ndz) are now direction cosines.

inprod(indir,nupb, @ndx,nupb, costheta,numupb)
// theta is the angle between the incident ray and the normal.
mul(costheta,numupb, costheta,numupb, tmp1,numupb)
sub(one,nupb, tmp1,numupb, tmp3,numupb)
sqrt(tmp3,numupb, sintheta,numupb)
imul(sintheta,numupb, invindex,numupb, sinphi,numupb) // Apply Snell's law
\begin{verbatim}
mul(sinphi, numupb, sinphi, numupb, tmp1, numupb)
sub(one, nupb, tmp1, numupb, tmp3, numupb)
sqrt(tmp3, numupb, cosphi, numupb)

// P1 is on the in ray at distance 1 from P
sub(Px, nupb, indx, nupb, P1x, nupb)
sub(Py, nupb, indy, nupb, P1y, nupb)
sub(Pz, nupb, indz, nupb, P1z, nupb)

// Point P2 is on the normal at distance costheta from P
mul(ndx, nupb, costheta, numupb, tmp1, numupb)
sub(Px, nupb, tmp1, numupb, P2x, nupb)
mul(ndy, nupb, costheta, numupb, tmp1, numupb)
sub(Py, nupb, tmp1, numupb, P2y, nupb)
mul(ndz, nupb, costheta, numupb, tmp1, numupb)
sub(Pz, nupb, tmp1, numupb, P2z, nupb)

// Point Q1 is on the normal at distance cosphi from P.
mul(ndx, nupb, cosphi, numupb, tmp1, numupb)
add(Px, nupb, tmp1, numupb, Q1x, nupb)
mul(ndy, nupb, cosphi, numupb, tmp1, numupb)
add(Py, nupb, tmp1, numupb, Q1y, nupb)
mul(ndz, nupb, cosphi, numupb, tmp1, numupb)
add(Pz, nupb, tmp1, numupb, Q1z, nupb)

// Calculate Q2 = Q1 + (P2-P1)*invindex
sub(P2x, nupb, P1x, nupb, tmp1, numupb)
mul(tmp1, numupb, invindex, nupb, tmp2, numupb)
add(Q1x, nupb, tmp2, numupb, Q2x, nupb) // Q2x = Q1x + (P2x-P1x)*invindex
sub(P2y, nupb, P1y, nupb, tmp1, numupb)
mul(tmp1, numupb, invindex, nupb, tmp2, numupb)
add(Q1y, nupb, tmp2, numupb, Q2y, nupb) // Q2y = Q1y + (P2y-P1y)*invindex
sub(P2z, nupb, P1z, nupb, tmp1, numupb)
mul(tmp1, numupb, invindex, nupb, tmp2, numupb)
add(Q1z, nupb, tmp2, numupb, Q2z, nupb) // Q2z = Q1z + (P2z-P1z)*invindex

sub(Q2x, nupb, Px, nupb, outdx, nupb)
sub(Q2y, nupb, Py, nupb, outdy, nupb)
sub(Q2z, nupb, Pz, nupb, outdz, nupb)
normalize(outdir, nupb)

RESULT IS TRUE
\end{verbatim}
The next function, \texttt{reflect}, deals with the reflection of a ray by the silvered surface of the mirror.

It arguments specifying the inward direction (\texttt{indir}) of a ray, the coordinates of the intersection point (\texttt{P}) on a spherical surface, and \texttt{c} is the \textit{z} coordinate of the centre of the sphere. These quantities allow it to calculate the direction cosines of the reflected ray. The calculation is easy since the angle of reflection is equal to the angle of incidence. As in \texttt{refract} we calculate the normal at the intersection point and then \( \cos \theta \) where \( \theta \) is the angle of incidence. Then, as before, we construct a right angled triangle \( P - A - B \) where \( P \) is the intersection point, \( A \) is the point on the incoming ray one unit from \( P \) and \( B \) is the point on the normal at a distance \( \cos \theta \) from \( P \) and on the same side of the mirror as \( A \). We then construct the point \( C \) equal to \( B - (A - B) = 2B - A \). The triangle \( P - C - B \) is clearly a mirror image of \( P - A - B \) and they are both in the same plane, so the line \( P - C \) lies in the reflected ray, giving us the required direction to place in \texttt{outdir}. The definition of \texttt{reflect} is as follows.

\begin{verbatim}
AND reflect(indir, P, c, outdir) = VALOF
{ // This computes the direction cosines of a reflected ray.
    // indir!0,indir!1,indir!2 hold the direction cosines of the in ray.
    // P!0,P!1,P!2 hold the coordinates of the intersection point on
    //     the mirror surface.
    // c is the z coordinate of the centre of the mirror surface.
    // outdir!0,outdir!1,outdir!2 will hold the direction cosines
    //     of the reflected ray.
    // All the numbers above have upperbound nupb.
    // The result is TRUE if the reflection is successful.
    LET costheta = VEC numupb
    LET indx = indir!0 // The direction cosines of the incident ray.
    AND indy = indir!1
    AND indz = indir!2

    LET Px = P!0 // The coordinates of the entry point.
    AND Py = P!1
    AND Pz = P!2

    LET outdx = outdir!0 // To hold the out direction cosines.
    AND outdy = outdir!1
    AND outdz = outdir!2

    LET Nx = VEC nupb // The direction cosines of the normal
    AND Ny = VEC nupb // at the intersection point P.
    AND Nz = VEC nupb

    LET Ax = VEC nupb // The coordinates of the point on the

    LET normal = Nx * Pz + Ny * Py
    LET normal = normal + Nz * Pz
    LET normal = normal * VEC lenupb

    LET normal = normal + Ax
    LET normal = normal - outdx
    LET normal = normal - outdy
    LET normal = normal - outdz

    LET normal = normal + Pz
    LET normal = normal + Py
    LET normal = normal + Pz

    LET normal = normal + Pz
    LET normal = normal + Py
    LET normal = normal + Pz

    LET normal = normal + Pz
    LET normal = normal + Py
    LET normal = normal + Pz

    LET normal = normal + Pz
    LET normal = normal + Py
    LET normal = normal + Pz

    LET normal = normal + Pz
    LET normal = normal + Py
    LET normal = normal + Pz

    LET normal = normal + Pz
    LET normal = normal + Py
    LET normal = normal + Pz

    LET normal = normal + Pz
    LET normal = normal + Py
    LET normal = normal + Pz

    LET normal = normal + Pz
    LET normal = normal + Py
    LET normal = normal + Pz

    LET normal = normal + Pz
    LET normal = normal + Py
    LET normal = normal + Pz

    LET normal = normal + Pz
    LET normal = normal + Py
    LET normal = normal + Pz

    LET normal = normal + Pz
    LET normal = normal + Py
    LET normal = normal + Pz

    LET normal = normal + Pz
    LET normal = normal + Py
    LET normal = normal + Pz

    LET normal = normal + Pz
    LET normal = normal + Py
    LET normal = normal + Pz

    LET normal = normal + Pz
    LET normal = normal + Py
    LET normal = normal + Pz

    LET normal = normal + Pz
    LET normal = normal + Py
    LET normal = normal + Pz

    LET normal = normal + Pz
    LET normal = normal + Py
    LET normal = normal + Pz

    LET normal = normal + Pz
    LET normal = normal + Py
    LET normal = normal + Pz

    LET normal = normal + Pz
    LET normal = normal + Py
    LET normal = normal + Pz

    LET normal = normal + Pz
    LET normal = normal + Py
    LET normal = normal + Pz

    LET normal = normal + Pz
    LET normal = normal + Py
    LET normal = normal + Pz

    LET normal = normal + Pz
    LET normal = normal + Py
    LET normal = normal + Pz

    LET normal = normal + Pz
    LET normal = normal + Py
    LET normal = normal + Pz

    LET normal = normal + Pz
    LET normal = normal + Py
    LET normal = normal + Pz

    LET normal = normal + Pz
    LET normal = normal + Py
    LET normal = normal + Pz

    LET normal = normal + Pz
    LET normal = normal + Py
    LET normal = normal + Pz

    LET normal = normal + Pz
    LET normal = normal + Py
    LET normal = normal + Pz

    LET normal = normal + Pz
    LET normal = normal + Py
    LET normal = normal + Pz

    LET normal = normal + Pz
    LET normal = normal + Py
    LET normal = normal + Pz

    LET normal = normal + Pz
    LET normal = normal + Py
    LET normal = normal + Pz

    LET normal = normal + Pz
    LET normal = normal + Py
    LET normal = normal + Pz

    LET normal = normal + Pz
    LET normal = normal + Py
    LET normal = normal + Pz

    LET normal = normal + Pz
    LET normal = normal + Py
    LET normal = normal + Pz

    LET normal = normal + Pz
    LET normal = normal + Py
    LET normal = normal + Pz

    LET normal = normal + Pz
    LET normal = normal + Py
    LET normal = normal + Pz

    LET normal = normal + Pz
    LET normal = normal + Py
    LET normal = normal + Pz

    LET normal = normal + Pz
    LET normal = normal + Py
    LET normal = normal + Pz

    LET normal = normal + Pz
    LET normal = normal + Py
    LET normal = normal + Pz

    LET normal = normal + Pz
    LET normal = normal + Py
    LET normal = normal + Pz

    LET normal = normal + Pz
    LET normal = normal + Py
    LET normal = normal + Pz

    LET normal = normal + Pz
    LET normal = normal + Py
    LET normal = normal + Pz

    LET normal = normal + Pz
    LET normal = normal + Py
    LET normal = normal + Pz

    LET normal = normal + Pz
    LET normal = normal + Py
    LET normal = normal + Pz

    LET normal = normal + Pz
    LET normal = normal + Py
    LET normal = normal + Pz

    LET normal = normal + Pz
    LET normal = normal + Py
    LET normal = normal + Pz

    LET normal = normal + Pz
    LET normal = normal + Py
    LET normal = normal + Pz

    LET normal = normal + Pz
    LET normal = normal + Py
    LET normal = normal + Pz

    LET normal = normal + Pz
    LET normal = normal + Py
    LET normal = normal + Pz

    LET normal = normal + Pz
    LET normal = normal + Py
    LET normal = normal + Pz

    LET normal = normal + Pz
    LET normal = normal + Py
AND Ay = VEC nupb // inray one unit from P
AND Az = VEC nupb

LET Bx = VEC nupb // The coordinates of the point on the
AND By = VEC nupb // the normal costheta from P
AND Bz = VEC nupb

LET Cx = VEC nupb // The coordinates of the point on the
AND Cy = VEC nupb // reflected ray one unit from P.
AND Cz = VEC nupb // Note B is the mid point of A and C.

AND tmp1 = VEC numupb

// Compute the normal.
copy(Px,nupb, Nx,nupb)
copy(Py,nupb, Ny,nupb)
sub(Pz,nupb, c,nupb, Nz,nupb)
normalize(@Nx,nupb)

// Calculate the coordinates of A = P - indir
sub(Px,nupb, indx,nupb, Ax,nupb)
sub(Py,nupb, indy,nupb, Ay,nupb)
sub(Pz,nupb, indz,nupb, Az,nupb)

// Calculate the coordinated of B = point - N * costheta
inprod(indir,nupb, @Nx,nupb, costheta,numupb)
mul(Nx,nupb, costheta,numupb, tmp1,numupb)
sub(Px,nupb, tmp1,numupb, Bx,nupb)
mul(Ny,nupb, costheta,numupb, tmp1,numupb)
sub(Py,nupb, tmp1,numupb, By,nupb)
mul(Nz,nupb, costheta,numupb, tmp1,numupb)
sub(Pz,nupb, tmp1,numupb, Bz,nupb)

// Calculate the coordinates of C = 2B - A
mulbyk(2, Bx,nupb)
mulbyk(2, By,nupb)
mulbyk(2, Bz,nupb)
sub(Bx,nupb, Ax,nupb, Cx,nupb)
sub(By,nupb, Ay,nupb, Cy,nupb)
sub(Bz,nupb, Az,nupb, Cz,nupb)

// Calculate the direction of the reflected ray normalize(C - P)
sub(Cx,nupb, Px,nupb, outdx,nupb)
The function `raytrace` follows a ray from its initial refraction at the front surface of the objective lens through the other refractions and the reflection in the mirror until it finally leaves the front surface of the mirror and hits the focal plane at \( z = 0 \). The \( x \) and \( y \) coordinates in the focal plane are copied to the arguments `focalx` and `focaly`. The function, although quite long, is simple using `intersect`, `refract` and `reflect` where needed as the ray passes through the five spherical surfaces.

```
AND raytrace(indir, P, colour, focalx, focaly) = VALOF
{ // Trace a ray all the way through the telescope
   // indir!0,indir!1,indir!2 are the direction cosines of the
   // incoming ray.
   // P!0,P!1,P!2 are the coordinates of a point on the incoming ray.
   // colour is either Blue or Red.
   // focalx,focaly are the coordinates in the focal plane resulting
   // from the incoming ray.
   // The result is TRUE if the raytracing was successful.

   // The ray passes through the following
   // (1) the front surface of the objective lens, crown glass
   // (2) the rear surface of the objective lens
   // (3) the front surface of the mirror, flint glass
   // (4) the reflective surface of the mirror
   // (5) back through the front surface of the mirror

   LET t1 = VEC nupb
   LET t2 = VEC nupb
   LET tmp1 = VEC numupb
   LET tmp2 = VEC numupb
   LET tmp3 = VEC numupb
   LET indx = VEC nupb // Private in direction cosines
   AND indy = VEC nupb
   AND indz = VEC nupb

   LET inptx = VEC nupb // Point on an in ray
   AND inpty = VEC nupb
   AND inptz = VEC nupb
```
LET x = VEC nupb // For intersection points
AND y = VEC nupb
AND z = VEC nupb

LET outdx = VEC nupb // Direction cosines of an out ray.
AND outdy = VEC nupb
AND outdz = VEC nupb

AND invindex = ? // The inverse of the refractive index of
// the current surface depending on colour
// and crown or flint glass.

// Front surface of the objective lens

copy(indir!0,nupb, indx,nupb) // In direction to front surface of objective
copy(indir!1,nupb, indy,nupb)
copy(indir!2,nupb, indz,nupb)

copy(P!0,nupb, inptx,nupb) // A point on the incident ray.
copy(P!1,nupb, inpty,nupb)
copy(P!2,nupb, inptz,nupb)

UNLESS intersect(@indx, @inptx, C1, R1, t1, t2) RESULTIS FALSE

// Select the negative root
IF t2!0 DO copy(t2,nupb, t1,nupb)

IF tracing DO
  writef("*nObjective front surface intersection point (x,y,z) is:*n")

mul(indx,nupb, t1,nupb, tmp1,numupb)
add(inptx,nupb, tmp1,numupb, x,nupb)
IF tracing DO
  { writef("x= "); prnum(x,8) }

mul(indy,nupb, t1,nupb, tmp1,numupb)
add(inpty,nupb, tmp1,numupb, y,nupb)
IF tracing DO
  { writef("y= "); prnum(y,8) }

mul(indz,nupb, t1,nupb, tmp1,numupb)
add(inptz,nupb, tmp1,numupb, z,nupb)
IF tracing DO
  { writef("z= "); prnum(z,8) }
// Apply Snell's law to obtain the transmitted direction

// indx, indy, indz hold the direction cosines of the in ray.
// outdir will hold the direction cosines of the out ray.
// (x, y, z) are the coordinates of the entry point on the
// objective lens front surface.
// C1 is the z coordinate of the centre of the lens front surface.
// invindex is the inverse of the refractive index.

// Set the air to glass refractive index for crown glass
// depending on the colour.
invindex := colour=Blue -> crownblueinvindex, crownredinvindex

refract(@indx, @x, C1, invindex, @outdx)

// Now deal with the rear surface of the objective lens.

copy(outdx, nupb, indx, nupb) // Out ray of front surface becomes

copy(outdy, nupb, indy, nupb) // the in ray of the rear surface.

copy(outdz, nupb, indz, nupb)

copy(x, nupb, inptx, nupb)    // The intersection point on the front surface

copy(y, nupb, inpty, nupb)    // is a point on the in ray of the rear surface.

copy(z, nupb, inptz, nupb)

UNLESS intersect(@indx, @inptx, C2, R2, t1, t2) RESULTIS FALSE

// Select the positive root.
UNLESS t2!0 DO copy(t2, nupb, t1, nupb)

IF tracing DO
    writef("t1= "); prnum(t1, nupb)

    mul(indx, nupb, t1, nupb, tmp1, numupb)
    add(inptx, nupb, tmp1, numupb, x, nupb)
    IF tracing DO
        { writef("x= "); prnum(x, 8) }

    mul(indy, nupb, t1, nupb, tmp1, numupb)
    add(infty, nupb, tmp1, numupb, y, nupb)
    IF tracing DO
        { writef("y= "); prnum(y, 8) }

    mul(indz, nupb, t1, nupb, tmp1, numupb)
    add(inptz, nupb, tmp1, numupb, z, nupb)
    IF tracing DO
        { writef("z= "); prnum(z, 8) }
mul(indz, nupb, t1, nupb, tmp1, numupb)
add(inptz, nupb, tmp1, numupb, z, nupb)
IF tracing DO
  { writef("z= "); prnum(z, 8) }

// Set the inverse of the glass to air refractive index for
crown glass depending on the colour.
invindex := colour=Blue -> crownblueindex, crownredindex

// Calculate the new out direction.
refract(@indx, @x, C2, invindex, @outdx)

// Now deal with the front surface of the mirror.
copy(outdx, nupb, indx, nupb)
copy(outdy, nupb, indy, nupb)
copy(outdz, nupb, indz, nupb)

copy(x, nupb, inptx, nupb)  // The intersection point on the
copy(y, nupb, inpty, nupb)  // rear surface of the objective lens.
copy(z, nupb, inptz, nupb)

UNLESS intersect(@indx, @inptx, C3, R3, t1, t2) RESULTIS FALSE

// Select the positive root, one of t1 or t2 is positive.
IF t1!0 DO copy(t2, nupb, t1, nupb)

IF tracing DO
  writef("*nThe mirror front surface intersection point (x,y,z) is:*n")
mul(indx, nupb, t1, nupb, tmp1, nupb)
add(inptx, nupb, tmp1, nupb, x, nupb)  // x = inptx + t1*indx
IF tracing DO
  { writef("x= "); prnum(x, 8) }

mul(indy, nupb, t1, nupb, tmp1, nupb)
add(infty, nupb, tmp1, nupb, y, nupb)  // y = infty + t1*indy
IF tracing DO
  { writef("y= "); prnum(y, 8) }

mul(indz, nupb, t1, nupb, tmp1, nupb)
add(inptz, nupb, tmp1, nupb, z, nupb)  // z = inptz + t1*indz
IF tracing DO
  { writef("z= "); prnum(z, 8) }
// Calculate the distance from the z axis.
mul(x,nupb, x,nupb, tmp1,numupb) // tmp1 = x^2
mul(y,nupb, y,nupb, tmp2,numupb) // tmp2 = y^2
add(tmp1,numupb, tmp2,numupb, tmp3,numupb) // tmp3 = x^2 + y^2
sqrt(tmp3,numupb, tmp1,numupb) // tmp1 = the radius

IF numcmp(tmp1,numupb, MirrorRadius,nupb) > 0 DO
  copy(tmp1,numupb, MirrorRadius,nupb)

IF tracing DO
  { writef("*nMirror radius= "); prnum(MirrorRadius,8) }

// Set the air to glass refractive index for flint glass
// depending on the colour.
invindex := colour=Blue -> flintblueinvindex, flintredinvindex

// Calculate the new out direction
refract(@indx, @x, C3, invindex, @outdx)

// Now deal with the reflecting surface of the mirror.

copy(outdx,nupb, indx,nupb) // Out direction of the front surface is
copy(outdy,nupb, indy,nupb) // the in direction to the silvered surface.
copy(outdz,nupb, indz,nupb)

copy(x,nupb, inptx,nupb) // The intersection point on the front
copy(y,nupb, inpty,nupb) // surface of the mirror.
copy(z,nupb, inptz,nupb)

UNLESS intersect(@indx, @inptx, C4, R4, t1, t2) RESULTIS FALSE

// Select the positive root. One of t1 or t2 is positive.
IF t1!0 DO copy(t2,nupb, t1,nupb)

IF tracing DO
  writef("*nThe mirror reflective surface intersection point (x,y,z) is:*n")

mul(indx,nupb, t1,nupb, tmp1,nupb)
add(inptx,nupb, tmp1,nupb, x,nupb) // x = inptx + t1*indx
IF tracing DO
  { writef("x= "); prnum(x, 8) }

mul(indy,nupb, t1,nupb, tmp1,nupb)
add(inpty,nupb, tmp1,nupb, y,nupb) // y = inpty + t1*indy
CHAPTER 5. INTERACTIVE GRAPHICS IN BCPL USING SDL

IF tracing DO
{ writeln("y= "); prnum(y, 8) }

mul(indz,nupb, t1,nupb, tmp1,nupb)
add(inptz,nupb, tmp1,nupb, z,nupb)  // y = inptz + t1*indz
IF tracing DO
{ writeln("z= "); prnum(z, 8) }

// Calculate the new out direction.
reflect(@indx, @x, C4, @outdx)

// Now deal with the front surface of the mirror again.

copy(outdx,nupb, indx,nupb)
copy(outdy,nupb, indy,nupb)
copy(outdz,nupb, indz,nupb)

copy(x,nupb, inptx,nupb)  // the intersection point on the front surface
copy(y,nupb, inpty,nupb)
UNLESS intersect(@indx, @inptx, C3, R3, t1, t2) RESULTIS FALSE

// Select the smaller root
IF numcmp(t2,nupb, t1,nupb)<0 DO copy(t2,nupb, t1,nupb)

IF tracing DO
  writeln("*nThe mirror front surface intersection point (x,y,z) is:*n")

mul(indx,nupb, t1,nupb, tmp1,nupb)
add(inptx,nupb, tmp1,nupb, x,nupb)
IF tracing DO
{ writeln("x= "); prnum(x, 8) }

mul(indy,nupb, t1,nupb, tmp1,nupb)
add(inpty,nupb, tmp1,nupb, y,nupb)
IF tracing DO
{ writeln("y= "); prnum(y,8) }

mul(indz,nupb, t1,nupb, tmp1,nupb)
add(inptz,nupb, tmp1,nupb, z,nupb)
IF tracing DO
{ writeln("z= "); prnum(z,8) }

// Set the inverse of the glass to air refractive index for flint glass
// depending on the colour.
5.16. A CATADIOPTRIC TELESCOPE

invindex := colour=Blue -> flintblueindex, flintredindex

// Calculate the new out direction
refract(@indx, @x, C3, invindex, @outdx)

TEST iszero(outdz,nupb)
THEN { // In the exceptional case where outdz is zero,
   // focalx and focaly are just x and y.
   copy(x,nupb, focalx,nupb)
   copy(y,nupb, focaly,nupb)
   }
ELSE { div(z,nupb, outdz,nupb, tmp1,numupb)
   mul(outdx,nupb, tmp1,numupb, tmp2,numupb)
   sub(x,nupb, tmp2,numupb, focalx,nupb)
   mul(outdy,nupb, tmp1,numupb, tmp2,numupb)
   sub(y,nupb, tmp2,numupb, focaly,nupb)
   }

RESULTIS TRUE

The next two functions allocate cleared vectors with a specified upperbound.

AND newvec(upb) = VALOF
{ LET p = spacep - upb - 1
   IF p<spacev DO
   { writef("*nMore space needed*n")
     abort(999)
     RESULTIS 0
   }
   spacep := p
   FOR i = 0 TO nupb DO p!i := 0
   RESULTIS p
   }

AND newnum(upb) = newvec(upb)

The next function, telescope, attempts to optimise the design of the telescope by successively making small changes to \( R_1, R_2, R_3 \) and \( R_4 \), preferring the settings that reduce the size of the scattering of points resulting from rays entering the objective lens at different positions. Three point sources are chosen, one on the axis of the telescope and the other two at about 1/8 and 1/4 degree off the axis (in the \( y \) direction). Rays of both blue and red light are used. If the optics of the telescope were perfect and if the focal length was 1000mm, then the three point sources would produce single point images on the focal plane at
about \( x = 0, z = 0 \) and \( y = 0, -2.0833 \) and \( -4.1666 \)mm. The program traces each ray though the telescope measuring its distance from its focal point. The largest of these distances is places in \texttt{spotsize}. Circles of this radius centred at each of the three focal points will enclose all the scattered points. How the program selects better values for \( R_1 \) to \( R_4 \) is explained later.

The function starts as follows.

\begin{verbatim}
AND telescope() BE
{ LET tmp1 = VEC nupb
  AND tmp2 = VEC nupb
  AND tmp3 = VEC nupb
  AND tmp4 = VEC nupb
  AND tmp5 = VEC nupb

  LET delta = VEC nupb // Current change in a radius value
  LET initdelta = VEC nupb // Current initial setting of delta
  LET failcount = -1 // Unset

  // The idea is to try changing each radius \( R_1 \) to \( R_4 \) by a small amount \( \Delta \)
  // either positive or negative. The distance between the average y coordinates
  // of an image spot and the theoretical position of its centre assuming a
  // focal length of 1000mm is placed in dist. \texttt{spotsize+dist} is placed in
  // \texttt{spotvalue} giving a measure of how good the optics are. If \texttt{spotvalue} reduces,
  // the current \( \Delta \) values are doubled and a new setting of \( R_1 \) to \( R_4 \) tried,
  // otherwise start again with a new set of small random \( \Delta \) values, and
  // start again.

  // Every time the sizes of the spots reduce, the file \texttt{catageometry.txt} is
  // written with the new radii \( R_1 \) to \( R_4 \).

  // If the file \texttt{catageometry.txt} exists, it is used to set the initial values
  // of \( R_1 \) to \( R_4 \).

  // Typically every time \texttt{cataopt} runs these radii are improved.

  // Directions 0, 1 and 2 are small angles in the y-z plane
  // used to generate test in rays.

  dir0cx := newnum(nupb) // Direction parallel to the telescope axis
  dir0cy := newnum(nupb)
  dir0cz := newnum(nupb)
  settok(0, dir0cx,nupb)
  settok(0, dir0cy,nupb)
  settok(1, dir0cz,nupb)
\end{verbatim}
5.16. A CATADIOPTRIC TELESCOPE

\[ \text{dir1cx} := \text{newnum(nupb)} \]
\[ \text{dir1cy} := \text{newnum(nupb)} \]
\[ \text{dir1cz} := \text{newnum(nupb)} \]
\[ \text{settok( 0, dir1cx,nupb)} \] // Direction about 1/8 degree off the telescope axis.
\[ \text{settok( -1, dir1cy,nupb)} \] // Note that 1 in 60 is about 1 degree
\[ \text{settok(480, dir1cz,nupb)} \] // so 1 in 480 is about 1/8 degree
\[ \text{normalize(@dir1cx,nupb)} \]
\[ \text{dir2cx} := \text{newnum(nupb)} \]
\[ \text{dir2cy} := \text{newnum(nupb)} \]
\[ \text{dir2cz} := \text{newnum(nupb)} \]
\[ \text{settok( 0, dir2cx,nupb)} \] // Direction about 1/4 degree off the telescope axis.
\[ \text{settok( -1, dir2cy,nupb)} \] // This is a field of view about the size of the moon.
\[ \text{settok(240, dir2cz,nupb)} \] // Note that 1 in 60 is about 1 degree
\[ \text{normalize(@dir2cx,nupb)} \]
\[ \text{Arad} := \text{newnum(nupb)} \] // Radius of the A circle in the objective
\[ \text{settok(50, Arad,nupb)} \]
\[ \text{Brad} := \text{newnum(nupb)} \] // Radius of the B circle in the objective
\[ \text{settok(25, Brad,nupb)} \]
\[ \text{R1} := \text{newnum(nupb)} \] // Objective lens front radius
\[ \text{prevR1} := \text{newnum(nupb)} \]
\[ \text{C1} := \text{newnum(nupb)} \] // Centre of objective lens front
\[ \text{R2} := \text{newnum(nupb)} \] // Objective lens rear radius
\[ \text{prevR2} := \text{newnum(nupb)} \]
\[ \text{C2} := \text{newnum(nupb)} \] // Centre of objective lens rear surface
\[ \text{R3} := \text{newnum(nupb)} \] // Concave mirror front radius
\[ \text{prevR3} := \text{newnum(nupb)} \]
\[ \text{C3} := \text{newnum(nupb)} \] // Centre of concave mirror front surface
\[ \text{R4} := \text{newnum(nupb)} \] // Concave mirror silvered surface radius
\[ \text{prevR4} := \text{newnum(nupb)} \]
\[ \text{C4} := \text{newnum(nupb)} \] // Centre of concave mirror silvered surface
\[ \text{T1} := \text{newnum(nupb)} \] // Objective thickness
\[ \text{T2} := \text{newnum(nupb)} \] // Mirror thickness

\[ \text{MirrorRadius} := \text{newnum(nupb)} \] // Radius of mirror

\[ \text{D} := \text{newnum(nupb)} \] // The distance between the objective and mirror.
\[ \text{F} := \text{newnum(nupb)} \] // This is typically 700mm.
\[ \text{F} := \text{newnum(nupb)} \] // The z coordinate of the focus plane, typically F=0
\[ \text{Inx} := \text{newnum(nupb)} \] // This typically give a focal length of 1000mm.

\[ \text{Inx} := \text{newnum(nupb)} \] // The direction cosines of an in going ray to a
Iny := newnum(nupb) // lens or mirror surface.
Inz := newnum(nupb)

Outx := newnum(nupb) // The direction cosines of an out going ray to a
Outy := newnum(nupb) // lens or mirror surface.
Outz := newnum(nupb)

outdircx := newnum(nupb)
outdircy := newnum(nupb)
outdircz := newnum(nupb)

deltaR1 := newnum(nupb) // Used to make small changes to R1 to R4
deltaR2 := newnum(nupb)
deltaR3 := newnum(nupb)
deltaR4 := newnum(nupb)

root2 := newnum(nupb)
one := newnum(nupb)

// Allocate numbers for the refractive indexes
crownblueindex := newnum(nupb)
crownredindex := newnum(nupb)
flintblueindex := newnum(nupb)
flintredindex := newnum(nupb)

// Allocate numbers for the inverse refractive indexes
crownblueinvindex := newnum(nupb)
crownredinvindex := newnum(nupb)
flintblueinvindex := newnum(nupb)
flintredinvindex := newnum(nupb)

spot0vx := newvec(16+16+2-1) // Space for points of spot0
spot0vy := newvec(16+16+2-1)
spot1vx := newvec(16+16+2-1) // Space for points of spot1
spot1vy := newvec(16+16+2-1)
spot2vx := newvec(16+16+2-1) // Space for points of spot2
spot2vy := newvec(16+16+2-1)

// Note that for spot0 the x coordinates in the focal plane
// are held in spot0vx. The table of subscripts is as follows

// 0 to 7  A circle Blue dots
// 8 to 15 A circle Red  dots
// 16 to 23 B circle Blue dots
// 24 to 31 B circle Red  dots
// 32 C ray Blue dot
// 33 C ray Red dot

// spot0vy holds the y coordinates of spot0 dots

// spot1vx, spot1vy, spot2vx and spot2vy hold the coordinates
// of the spot1 and spot2 Blue and Red dots

FOR i = 0 TO 16+16+2-1 DO
{ // Allocate space for all the spot dot coordinates
  spot0vx!i := newnum(nupb)
  spot0vy!i := newnum(nupb)
  spot1vx!i := newnum(nupb)
  spot1vy!i := newnum(nupb)
  spot2vx!i := newnum(nupb)
  spot2vy!i := newnum(nupb)
}

bestspotsize := newnum(nupb)
spotsize := newnum(nupb)
dist := newnum(nupb)
bestdist := newnum(nupb)
bestspotvalue := newnum(nupb)
spotvalue := newnum(nupb)

UNLESS spotsize DO
{ writef("More space needed\n")
  abort(999)
  RETURN
}

The program continues as follows initialising several global values such as T1 and T2 the thicknesses of the objective lens and mirror measured at their centres. D is set to 700, the z position of the mirror. The square root of 2 is placed in root2 and a high precision representation of the constant 1 is placed in one. The refractive indices for blue and red light for crown and flint glass are placed in suitable variables, and it is also convenient to hold the inverse versions of these values. The program goes on to set well chosen initial values to the radii of the lens and mirror surfaces in R1 to R4. These settings cause each of the selected rays to hit the focal plane at a distance no greater than 0.0217mm from the theoretical centre of the image for a point source from its direction.

// All numbers have been created successfully

// Set the unchanginging geometry of the telescope
settok( 4, T1,nupb)  // Objective lens thickness 4mm at centre.
settok( 4, T2,nupb)  // Mirror thickness 4mm at centre.
settok(700, D,nupb)  // z coordinate of the mirror silvered surface

settok(2, tmp1,nupb)
sqrt(tmp1,nupb, root2,nupb)

settok(1, one,nupb)

// Refractive indices.
// The objective glass is crown and the mirror glass is flint.


//
// blue 486 nm 1.51690 1.6321
// red  640 nm 1.50917 1.6161
//

str2num("1.51690", crownblueindex,nupb)
str2num("1.50917", crownredindex,nupb)
str2num("1.6321", flintblueindex,nupb)
str2num("1.6161", flintredindex,nupb)

inv(crownblueindex,nupb, crownblueinvindex,nupb)
inv(crownredindex,nupb, crownredinvindex,nupb)
in(flintblueindex,nupb, flintblueinvindex,nupb)
in(flintredindex,nupb, flintredinvindex,nupb)

IF tracing DO
{ writef("crownblueindex= "); prnum(crownblueindex, 6)
writef("crownredindex= "); prnum(crownredindex, 6)
writef("flintblueindex= "); prnum(flintblueindex, 6)
writef("flintredindex= "); prnum(flintredindex, 6)

writef("crownblueinvindex= "); prnum(crownblueinvindex, 6)
writef("crownredinvindex= "); prnum(crownredinvindex, 6)
writef("flintblueinvindex= "); prnum(flintblueinvindex, 6)
writef("flintredinvindex= "); prnum(flintredinvindex, 6)
}

// Initialize the setting of the lens and mirror surface radii.
// These are overwritten if file catageometry.txt exists.

// It seems that the initial settings of R1 to R4 often cause
// the iterations to lead to a false minimum. So some
// experimentation was needed before a good result was obtained.
5.16. A CATADIOPTRIC TELESCOPE

//str2num("5000", R1,nupb) // Objective front surface radius
//str2num("5000", R2,nupb) // Objective rear surface radius
//str2num("1100", R3,nupb) // Radius of mirror front surface
//str2num("1200", R4,nupb) // Radius of mirror silvered surface
// This give a spot size of about 0.0173mm after many hours

str2num("+0.1537 9603 6301 4326 5100 0000 0000 E1", R1,nupb)
str2num("+0.4978 7269 7214 4032 0700 0000 0000 E1", R2,nupb)
str2num("+0.0926 7191 6585 7604 6700 0000 0000 E1", R3,nupb)
str2num("+0.1505 7199 0416 5610 7700 0000 0000 E1", R4,nupb)
// This give a spot radius of about 0.0173mm

str2num("+0.1545 1166 8077 8501 2000 0000 0000 E1", R1,nupb)
str2num("+0.4969 1626 0425 6302 7000 0000 0000 E1", R2,nupb)
str2num("+0.0946 9427 9298 7956 8000 0000 0000 E1", R3,nupb)
str2num("+0.1523 9989 0724 6298 1000 0000 0000 E1", R4,nupb)
// This give a spot radius of about
// 0.0217mm for A2 and 0.0153mm for A1 and 0.0082mm for A0
// The Airy disc radius is 0.00671mm

// To show that this is close to the theoretical optimum for a
// telescope with an aperture of 100m consider the following.

// Light is electromagnetic radiation but, unlike radio waves
// which have wave lengths measured in metres, visible light
// has a wave length measured in nano-metres. Blue light is
// typically 486nm and red light is about 640nm.

// If we consider a point source of light with a wave length of
// 550nm at infinity on the axis of a optically perfect telescope,
// rays passing through every point the objective lens will be in
// phase when they reach the focus point. As a result of
// diffraction the image is not a tiny spot but a rather larger
// spot surrounded by light and dark rings. A point of the
// innermost dark ring is where the path lengths from opposite
// edges of the objective lens differ by about one wave length of
// the colour being considered. This is because rays being received
// at this point are out of phase with other rays and so are
// partially cancelled. The size of the bright spot at the centre
// is thus somewhat smaller than the size of the innermost dark
// diffraction ring. By applying simple geometry we can estimate
// the radius of the innermost diffraction ring as (1000/100)*(550/2)
// which is about 2750nm. This is 2750*10^-9m = 2750*10^-6mm
// = 0.00275mm. The diameter of the bright spot will thus be about
// 0.0055mm. Since our best spot size of 0.0093mm is not much
// larger than this, so we are close to the theoretical limit
// of a telescope with an aperture of 100mm.

// As a rule of thumb the maximum usable magnification of a
// telescope is between about 1 and 1.2 times its aperture in
// millimetres. In practice it is far less that because of
// the disturbance caused the atmosphere.

The next part of the program checks if the file catageometry.txt exits,
creating it if necessary using wrgeometry(). It then sets factor, R1, R2, R3
and R4 from the values specified in this file.

factor := 5 // A power of ten
// The initial values of the deltas are random
// integers in the range 0 to 9999 which are then divided
// by 10^factor.
// factor was incremented every time no improvement
// is made after 300 iterations.
// The delta values are doubled every time spotvalue
// reduces.
// Every time an improvement is made, factor, R1, R2, R3
// and R4 are written to the file catageometry.txt.
// If this file exists it is used to set the starting
// values the next time cataopt is run.

geometrystream := findinput("catageometry.txt")

UNLESS geometrystream DO
{ writeln("Calling wrgeometry()*n")
  wrgeometry()
  geometrystream := findinput("catageometry.txt")
}

IF geometrystream DO
{ // File catageometry.txt exists so update factor, factor, R1, R2, R3 and R4
  // from values in this file.
  selectinput(geometrystream)

  factor := readn()
  rdch()

  readline(currentline)
  str2num(currentline, R1,nupb)
5.16. A CATADIOPTRIC TELESCOPE

readline(currentline)
str2num(currentline, R2,nupb)

readline(currentline)
str2num(currentline, R3,nupb)

readline(currentline)
str2num(currentline, R4,nupb)

endstream(geometrystream)

writef("*nInitial state set from file catageometry.txt*n*n")

} IF initfactor>0 DO factor := initfactor

writef("factor = %n*n", factor)
writef("R1= "); prnum(R1, 8)
writef("R2= "); prnum(R2, 8)
writef("R3= "); prnum(R3, 8)
writef("R4= "); prnum(R4, 8)

settok(100, bestspotsize,nupb) // Unset the best spot size
settok( 0, dist,nupb) // Unset dist
settok( 9, bestdist,nupb) // Unset bestdist
settok( 99, bestspotvalue,nupb) // Unset bestspotvalue

setzero(spotsize,nupb)
setzero(dist,nupb)
setzero(spotvalue,nupb)

setzero(deltaR1,nupb) // This causes the first setting of
setzero(deltaR2,nupb) // the spotsize to correspond to the
setzero(deltaR3,nupb) // initial setting of R1 to R4 before
setzero(deltaR4,nupb) // any deltas are applied.
reduced := TRUE

again: // Enter here if R1 to R4 have their initial values or
      // have values that improved the gemometry of the telescope.

      // Save current values of R1 to R4
      copy(R1,nupb, prevR1,nupb)
      copy(R2,nupb, prevR2,nupb)
      copy(R3,nupb, prevR3,nupb)
copy(R4,nupb, prevR4,nupb)

    // prevR1 to prevR4 are the best values so far.

newdelta:

TEST reduced
THEN { // Previous setting of the delta values caused an improvement
    // so double them.
    //writef("*nreduced=TRUE so double the delta values*n")
    mulbyk(2, deltaR1,nupb)
    mulbyk(2, deltaR2,nupb)
    mulbyk(2, deltaR3,nupb)
    mulbyk(2, deltaR4,nupb)
}
ELSE { // The previous setting, if any, made no improvement
    // so choose a new random setting, ensuring that
    // deltaR3 > deltaR4
    LET k = factor

    //writef("*nChoosing a new random setting of the delta values*n")
    settok((randno(9999)-5000) | 1, deltaR1,nupb)
    settok((randno(9999)-5000) | 1, deltaR2,nupb)
    settok((randno(9999)-5000) | 1, deltaR3,nupb)
    settok((randno(9999)-5000) | 1, deltaR4,nupb)
    // Note that "| 1" above ensures all the deltas are nonzero.

    // 70% of the time only change R1 and R2 or R3 and R4.
    IF randno(101)<=70 DO
        TEST randno(101)<=50
        THEN { setzero(deltaR1,nupb)
                setzero(deltaR2,nupb)
            }
        ELSE { setzero(deltaR3,nupb)
                setzero(deltaR4,nupb)
            }
    }

    // Divide the delta values by 10^-factor

    IF k>=4 DO
    { LET e = k/4
        // Divide each delta by 10000^-e
        deltaR1!1 := deltaR1!1 - e
        deltaR2!1 := deltaR2!1 - e
        deltaR3!1 := deltaR3!1 - e
    }
\texttt{deltaR4!1 := deltaR4!1 - e}
\texttt{k := k \text{ MOD} 4}
\}

// Divide each delta by 10^k 
UNTIL \( k \leq 0 \) DO
\{ 
\texttt{divbyk(10, deltaR1,nupb)}
\texttt{divbyk(10, deltaR2,nupb)}
\texttt{divbyk(10, deltaR3,nupb)}
\texttt{divbyk(10, deltaR4,nupb)}
\texttt{k := k-1}
\}

// Add the delta values to the radii.
\texttt{add(deltaR1,nupb, prevR1,nupb, R1,nupb)}
\texttt{add(deltaR2,nupb, prevR2,nupb, R2,nupb)}
\texttt{add(deltaR3,nupb, prevR3,nupb, R3,nupb)}
\texttt{add(deltaR4,nupb, prevR4,nupb, R4,nupb)}

\texttt{IF R1!0 \mid R2!0 \mid R3!0 \mid R4!0 DO}
\{ 
// One of the radii has become negative
\texttt{writef("One of R1 to R4 has become negative, so choose a different delta*n")}
\texttt{writef("R1= "); prnum(R1, 8)}
\texttt{writef("R2= "); prnum(R2, 8)}
\texttt{writef("R3= "); prnum(R3, 8)}
\texttt{writef("R4= "); prnum(R4, 8)}
\texttt{failcount := failcount+1}
\texttt{abort(9999)}
\texttt{GOTO newdelta}
\}

// Insist that R3 is greater than R4
\texttt{IF numcmp(R3,nupb, R4,nupb) > 0 DO}
\{ 
\texttt{failcount := failcount+1}
\texttt{writef("R3 is greater than R4*n")}
\texttt{writef("R3= "); prnum(R3, 8)}
\texttt{writef("R4= "); prnum(R4, 8)}
\texttt{abort(1000)}
\texttt{failcount := failcount+1}
\texttt{GOTO newdelta}
\}

// Initialize the focal plane image
CHAPTER 5. INTERACTIVE GRAPHICS IN BCPL USING SDL

fillsurf(c_gray)
setcolour(c_black)
plotf(10,310, "spotmag = %n", spotmag)
plotf(10,290, "factor = %n", factor)

{ LET n, f1, f2 = 0, 0, 0
  IF dist!1= 1 DO n, f1, f2 := dist!2, dist!3, dist!4
  IF dist!1= 0 DO n, f1, f2 := 0, dist!2, dist!3
  IF dist!1=-1 DO n, f1, f2 := 0, 0, dist!2
  plotf(10,270, "dist =%i2,%z4 %z4 mm", n,f1, f2)
  n, f1, f2 := 0, 0, 0
  IF bestdist!1= 1 DO n, f1, f2 := bestdist!2, bestdist!3, bestdist!4
  IF bestdist!1= 0 DO n, f1, f2 := 0, bestdist!2, bestdist!3
  IF bestdist!1=-1 DO n, f1, f2 := 0, 0, bestdist!2
  plotf(10,250, "best =%i2,%z4 %z4 mm", n,f1, f2)
}
plotf(centrex-85,110, "R1 %i4,%z4 %z4 %z4 mm*n",
      R1!2, R1!3, R1!4, R1!5)
plotf(centrex-85, 90, "R2 %i4,%z4 %z4 %z4 mm*n",
      R2!2, R2!3, R2!4, R2!5)
plotf(centrex-85, 70, "R3 %i4,%z4 %z4 %z4 mm*n",
      R3!2, R3!3, R3!4, R3!5)
plotf(centrex-85, 50, "R4 %i4,%z4 %z4 %z4 mm*n",
      R4!2, R4!3, R4!4, R4!5)

IF spotsize!1=1 TEST spotsize!2>9
THEN plotf(centrex-85, 30, "Spot size %i4,%z4 %z4 %z4 mm",
      spotsize!2, spotsize!3, spotsize!4, spotsize!5)
ELSE plotf(centrex-85, 30, "Spot size %n,%z4 %z4 %z4 %z4 mm",
      spotsize!2, spotsize!3, spotsize!4, spotsize!5, spotsize!6)
IF spotsize!1=0 DO
  plotf(centrex-85, 30, "Spot size 0.%z4 %z4 %z4 %z4 mm",
       spotsize!2, spotsize!3, spotsize!4, spotsize!5)
IF spotsize!1=-1 DO
  plotf(centrex-85, 30, "Spot size 0.0000 %z4 %z4 %z4 %z4 mm",
       spotsize!2, spotsize!3, spotsize!4)

IF bestspotsize!1=1 TEST bestspotsize!2>9
THEN plotf(centrex-85, 10, "Spot size %i4,%z4 %z4 %z4 mm",
      bestspotsize!2, bestspotsize!3, bestspotsize!4, bestspotsize!5)
ELSE plotf(centrex-85, 10, "Spot size %n,%z4 %z4 %z4 %z4 mm",
      bestspotsize!2, bestspotsize!3, bestspotsize!4, bestspotsize!5, bestspotsize!6)
5.16. A CATADIOPTRIC TELESCOPE

IF bestspotsize!1=0 DO
    plotf(centrex-85, 10, "Best size 0.%z4 %z4 %z4 %z4 mm",
        bestspotsize!2, bestspotsize!3, bestspotsize!4, bestspotsize!5)
IF bestspotsize!1=-1 DO
    plotf(centrex-85, 10, "Best size 0.0000 %z4 %z4 %z4 mm",
        bestspotsize!2, bestspotsize!3, bestspotsize!4)

setcolour(c_black)
moveto(0, centrey)
drawby(screenxsize, 0)
moveto(centrex, 0)
drawby(0, screenysize)
updatescreen()

// Calculate the values that depend on the radii R1 to R4

// Calculate the z coordinate of the objective front surface centre
copy(T1,nupb, tmp1,nupb) // The thickness of the objective lens
divbyk(2, tmp1,nupb) // Half of its thickness

sub(R1,nupb, tmp1,nupb, C1,nupb) // Centre of the objective front surface.
sub(tmp1,nupb, R2,nupb, C2,nupb) // Centre of the objective rear surface.
sub(D,nupb, T2,nupb, tmp1,nupb)
sub(tmp1,nupb, R3,nupb, C3,nupb) // Centre of mirror front surface
sub(D,nupb, R4,nupb, C4,nupb) // Centre of mirror silvered surface

The optimisation process involves tracing a selection of rays through the telescope that originate from up to three point sources and entering the telescope at different positions on the objective lens. Each ray arrives as a dot on the focal plane. The program tries to minimise the scattering of these dots. The $x$ and $y$ coordinates of each dot is saved in the vectors such as spot0vx and spot0vy. There are a total of 34 rays for each of the three point sources, but normally, to improve the rate of convergence, only a subset of the possible rays are used. The rays are processes by calls of doray and normally many of these are commented out. The call calcspotsize(spotno) where spotno is 0, 1 or 2, determines the size of the image generated from each of the three directions. The coordinate vectors are initialised with $x$ values of 100 which will never result from a valid ray to allow calcspotsize to ignore coordinates not resulting from calls of doray. The program thus continues as follows.

// Mark all dot coordinates as unset so that calcspotsize
// will only use those that have been defined.
FOR i = 0 TO 16+16+2-1 DO
{
  // Unset image points have x set to 100.
  settok(100, spot0vx!i)
  settok(100, spot1vx!i)
  settok(100, spot2vx!i)
}

setzero(MirrorRadius,nupb)

// The rate of convergence is greatly improved by commenting
// out most of the call of doray, but at least two should be
// left in. Leaving them mostly uncommented causes a prettier
// picture to be drawn. If you leave only two call of doray,
// it is probably best to leave:
// doray(2,'A',0) and doray(2,'A',4).

// Now trace several rays through the telescope, storing the
// image dots in the focal plane coordinate vectors.

doray(0, 'A', 0)
doray(0, 'A', 1)
doray(0, 'A', 2)
doray(0, 'A', 3)
doray(0, 'A', 4)
doray(0, 'A', 5)
doray(0, 'A', 6)
doray(0, 'A', 7)

// doray(0, 'B', 0)
// doray(0, 'B', 1)
// doray(0, 'B', 2)
// doray(0, 'B', 3)
// doray(0, 'B', 4)
// doray(0, 'B', 5)
// doray(0, 'B', 6)
// doray(0, 'B', 7)

doray(0, 'C', 0)

doray(1, 'A', 0)
doray(1, 'A', 1)
doray(1, 'A', 2)
doray(1, 'A', 3)
doray(1, 'A', 4)
5.16. A CATADIOPTRIC TELESCOPE

doray(1, 'A', 5)
doray(1, 'A', 6)
doray(1, 'A', 7)

//doray(1, 'B', 0)
//doray(1, 'B', 1)
//doray(1, 'B', 2)
//doray(1, 'B', 3)
//doray(1, 'B', 4)
//doray(1, 'B', 5)
//doray(1, 'B', 6)
//doray(1, 'B', 7)

doray(1, 'C', 0)

doray(2, 'A', 0)
doray(2, 'A', 1)
doray(2, 'A', 2)
doray(2, 'A', 3)
doray(2, 'A', 4)
doray(2, 'A', 5)
doray(2, 'A', 6)
doray(2, 'A', 7)

//doray(2, 'B', 0)
//doray(2, 'B', 1)
//doray(2, 'B', 2)
//doray(2, 'B', 3)
//doray(2, 'B', 4)
//doray(2, 'B', 5)
//doray(2, 'B', 6)
//doray(2, 'B', 7)

doray(2, 'C', 0)

IF tracing DO
{ writef("*nMirrorRadius= "); prnum(MirrorRadius,8)
newline()
//abort(5100)
}

// Calculate spotsize and dist.

setzero(spotsize,nupb)
setzero(dist,nupb)
calcspotsize(0)
calcspotsize(1)
calcspotsize(2)

// For the current setting of R1 to R4, spotsize is now
// the size of the largest of the images from the selected
// directions, and dist is greatest distance the
// average y is from the theoretical centre of its spot.

// Set spotvalue to spotsize + dist
// It is a measure of how good the optics of the telescope is.

//writef("dist= "); prnum(dist, 8)
//writef("spotsize= "); prnum(spotsize, 8)

add(spotsize,nupb, dist,nupb, spotvalue,nupb)
//writef("spotsize= "); prnum(spotsize, 8)
//writef("spotvalue= "); prnum(spotvalue, 8)
//writef("bestspotvalue= "); prnum(bestspotvalue, 8)

// If spotvalue is smaller than bestspotvalue, the optics of
// the telescope has improved, so the current settings of
// \{tt R1\} to \{tt R4\} are are remembered and bestspotsize,
// bestdist and bestspotvalue updated.

TEST numcmp(spotvalue,nupb, bestspotvalue,nupb) < 0
THEN { reduced := TRUE
  writef("%i4 Spotvalue has reduced*n", iterations)

  //writef("%i4 spotvalue= ", iterations); prnum(spotvalue,8)
  //writef("%i4 bestspotvalue= ", iterations); prnum(bestspotvalue,8)

  //writef("%i4 spotsize= ", iterations); prnum(spotsize,8)
  //writef("%i4 bestspotsize= ", iterations); prnum(bestspotsize,8)

  //writef("%i4 dist= ", iterations); prnum(dist,8)
  //writef("%i4 bestdist= ", iterations); prnum(bestdist,8)

  copy(spotsize,nupb, bestspotsize,nupb)
copy(dist,nupb, bestdist,nupb)
copy(spotvalue,nupb, bestspotvalue,nupb)
wrgeometry()
iterations := iterations-1
failcount := 0
The next function calculates the size of the image generated by a collection of rays taken from a point source of blue and red light from a direction specified by the argument spotno which is 0, 1 or 2. The size is the largest distance of a dot in the focal plane from the theoretical centre for the specified direction. The y coordinate of the centres resulting from directions 0, 1 and 2 are 0mm, -2.0833mm and -4.1866mm, respectively. The x and y coordinates of dots in the focal plane resulting from rays from direction 0 are in spot0vx and spot0vy. For directions 1 and 2, the vectors are spot1vx and spot1vy, and spot2vx and spot2vy.

AND calcspotsize(spotno) BE
{ // This function finds average y coordinate for the current spot
  // placing it in avgy. It then calculates its distance from the
  // theoretical centre of the spot, dist is updated.

  // It then inspects each dot belonging to the specified spot. If

// its distance from (0,avgy) is greater than spotsize, spotsize
// is updated.

// The algorithm attempts to minimise both dist and spotsizesq
// placing more significance on dist since this is a measure
// of how close the focal length is to 1000mm.

LET tmp1 = VEC nupb
LET tmp2 = VEC nupb
LET tmp3 = VEC nupb
LET tmp4 = VEC nupb
LET tmp5 = VEC nupb
LET tmp6 = VEC nupb
LET avgy = VEC nupb
LET count = 0 // Count of dots in this spot
LET spotsizesq = VEC nupb

// Set the theoretical centre of the specified spot to (0,cgy).
LET cgy = spotno=0 ->
    (TABLE FALSE, 0, 0, 0000), // 0.0000 for direction 0
    spotno=1 ->
    (TABLE TRUE, 1, 2, 0833), // -2.0833 for direction 1
    (TABLE TRUE, 1, 4, 1666) // -4.1666 for direction 2
// Note that cgy has upb=3.

// Select the coordinate vectors for the specified spot
LET px = spotno=0 -> spot0vx,
    spotno=1 -> spot1vx,
    spot2vx
LET py = spotno=0 -> spot0vy,
    spotno=1 -> spot1vy,
    spot2vy
setzero(avgy,nupb)

// Calculate avgy and hence dist
FOR i = 0 TO 16+16+2-1 DO
    \ i = 0 to 7   Blue A rays
    \ i = 8 to 15  Red A rays
    \ i = 16 to 23 Blue B rays
    \ i = 24 to 31 Red B rays
    \ i = 32 to 33 Blue and Red C rays
    LET x = px!i
    LET y = py!i

    IF x!1=1 & x!2=100 LOOP // An unset dot has x=100.
add(y, nupb, avgy, nupb, tmp1, nupb)
copy(tmp1, nupb, avgy, nupb)
count := count+1
// writef("%i2 count=%i2 y= ", i, count); prnum(y, 5)
// writef("avgy= ", i); prnum(avgy, 5)
}

// writef("count=%n*n", count)

IF count DO
{ divbyk(count, avgy, nupb) // Compute the average
// writef("average y= "); prnum(avgy, 5)
sub(cgy, 3, avgy, nupb, tmp1, nupb)
// writef("tmp1= "); prnum(tmp1, 5)
// Take its absolute value
tmp1!0 := FALSE
// writef("centre dist= "); prnum(tmp1, 5)
// If greater than dist, update dist.
IF numcmp(tmp1, nupb, dist, nupb) > 0 DO copy(tmp1, nupb, dist, nupb)
// writef("dist= "); prnum(dist, 5)
// abort(8888)
}

// writef("dist= "); prnum(dist, 5)
// abort(8888)
setzero(spotsizesq, nupb)

// Calculate the radius squared
FOR i = 0 TO 16+16+2-1 DO
{ // i = 0 to 7 Blue A rays
  // i = 8 to 15 Red A rays
  // i = 16 to 23 Blue B rays
  // i = 24 to 31 Red B rays
  // i = 32 to 33 Blue and Red C rays
  LET x = px!i
  LET y = py!i

  IF x!1=1 & x!2=100 LOOP // An unset dot has x coordinate equal to 100.

    // writef("x= "); prnum(x, 8)
    mul(x, nupb, x, nupb, tmp1, nupb) // tmp1 = x^2
    // writef("x^2= "); prnum(tmp1, 8)
    // writef("avgy= "); prnum(avgy, 8)

  }
CHAPTER 5. INTERACTIVE GRAPHICS IN BCPL USING SDL

The next function just creates the file catageometry.txt writing to it the current values of factor, R1, R2, R3 and R4.

AND wrgeometry() BE
{ // Create file catageometry.txt with the current settings
  // of factor and R1 to R4.
  LET filename = "catageometry.txt"
  //writef("Calling findoutput(\"%s\")\n", filename)
  geometrystream := findoutput(filename)
  UNLESS geometrystream DO
  { writef("\nUnable to create file: \"%s\"\n", filename) 
    abort(999) 
    RETURN 
  }
  selectoutput(geometrystream)
  writef("\n\n", factor)

5.16. A CATADIOPTRIC TELESCOPE

prnum(R1,8)
prnum(R2,8)
prnum(R3,8)
prnum(R4,8)

writef("*nGives spotsize: "); prnum(spotsize, 8)

endstream(geometrystream)
selectoutput(stdout)
}

AND readline(str) BE
{ LET len = 0
 { LET ch = rdch()
   IF ch = '*n' | ch=endstream | len>255 BREAK
   len := len + 1
   str%len := ch
 } REPEAT

str%0 := len
}

AND doray(n, ch, pos) BE // direction, radius ch, point number
{ LET dir = n=0 -> @dir0cx,
   n=1 -> @dir1cx,
      @dir2cx
 LET radius = ch='A' -> Arad, // Outer circle
   ch='B' -> Brad, // Inner circle
      0 // Centre point
 LET tmp1 = VEC nupb
 LET raddiv = VEC nupb
 LET focalx = ?
 AND focaly = ?

 LET px = n=0 -> spot0vx,
   n=1 -> spot1vx,
      spot2vx

 LET py = n=0 -> spot0vy,
   n=1 -> spot1vy,
      spot2vy
CHAPTER 5. INTERACTIVE GRAPHICS IN BCPL USING SDL

// Calculate the position of the dot coordinates
px, py := px+pos, py+pos // Add the position (0 to 7) in the circle
IF ch = 'A' DO px, py := px+ 0, py+ 0 // pos of dot pos circle A
IF ch = 'B' DO px, py := px+16, py+16 // pos of dot pos circle B
IF ch = 'C' DO px, py := px+32, py+32 // pos of then dot ray C

// abort(9999)
TEST radius=0
THEN setzero(raddiv,nupb)
ELSE div(radius,nupb, root2,nupb, raddiv,nupb)

IF tracing DO
{ writef("*ndoray: %c%n pos=%n*n", ch, n, pos)
  writef("root2= "); prnum(root2,8)
  writef("raddiv= "); prnum(raddiv,8)
}

setzero(Inz,nupb) // All entry points are in the plane z=0
SWITCHON pos INTO
{ DEFAULT: RETURN
  CASE 0: setzero(Inx,nupb)
    TEST radius=0
    THEN setzero(Iny,nupb)
    ELSE copy(radius,nupb, Iny,nupb)
  ENDCASE
  CASE 1: copy(raddiv,nupb, Inx,nupb)
    copy(raddiv,nupb, Iny,nupb)
  ENDCASE
  CASE 2: copy(radius,nupb, Inx,nupb)
    setzero(Iny,nupb)
  ENDCASE
  CASE 3: copy(raddiv,nupb, Inx,nupb)
    copy(raddiv,nupb, Iny,nupb); Iny!0 := TRUE
  ENDCASE
  CASE 4: setzero(Inx,nupb)
    copy(radius,nupb, Iny,nupb); Iny!0 := TRUE
  ENDCASE
  CASE 5: copy(raddiv,nupb, Inx,nupb); Inx!0 := TRUE
    copy(raddiv,nupb, Iny,nupb); Iny!0 := TRUE
  ENDCASE
  CASE 6: copy(radius,nupb, Inx,nupb); Inx!0 := TRUE
    setzero(Iny,nupb)
  ENDCASE
  CASE 7: copy(raddiv,nupb, Inx,nupb); Inx!0 := TRUE
    copy(raddiv,nupb, Iny,nupb)
  ENDCASE
5.16. A CATADIOPTRIC TELESCOPE

//writef("*nIncident ray intersection with the plane z=0*n")
//writef("Inx= "); prnum(Inx,8)
//writef("Iny= "); prnum(Iny,8)
//writef("Inz= "); prnum(Inz,8)

//writef("*nDirection of the incident ray*n")
//writef("dir!0= "); prnum(dir!0,8)
//writef("dir!1= "); prnum(dir!1,8)
//writef("dir!2= "); prnum(dir!2,8)

//writef("*nEntry point %c%n, Direction %n, Blue*n", ch, pos, n)
  focalx, focaly := px!0, py!0 // Location of a blue dot
  raytrace(dir, @Inx, Blue, focalx, focaly)

IF tracing DO
{ newline()
  writef("%c%n Blue x= ",ch,pos); prnum(focalx,8)
  writef("%c%n Blue y= ",ch,pos); prnum(focaly,8)
}

setcolour(c_blue)
//IF pos<4 DO
  drawdot(n, scale(focalx), scale(focaly))

//IF tracing DO
 // abort(5000)

//writef("*nEntry point %c%n, Direction %n, Red*n", ch, pos, n)

TEST ch='C'
THEN focalx, focaly := px!1, py!1 // Location of a red dot for C
ELSE focalx, focaly := px!8, py!8 // Location of a red dot for A or B
  raytrace(dir, @Inx, Red, focalx, focaly)

IF tracing DO
{ newline()
  writef("%c%n Red x= ",ch,pos); prnum(focalx,8)
  writef("%c%n Red y= ",ch,pos); prnum(focaly,8)
}

setcolour(c_red)
//IF pos<4 DO
  drawdot(n, scale(focalx), scale(focaly))
AND scale(num) = VALOF
{ LET res = ?
  LET e = num!1
  IF e>1 D0 res := 9999_9999
  IF e=1 D0 res := num!2 * 10000 + num!3
  IF e=0 D0 res := num!2
  IF e<0 D0 res := 0
  IF num!0 D0 res := -res
  // res is in unit of 1/10000 mm
  // writef("scale: num= "); prnum(num, 5)
  // writef("scale: res=%*n*n", res)
  RESULTIS res
}
This screenshot shows the scattering of blue and red dots corresponding to rays passing through centre and eight equally placed positions on the objective lens 50mm from the z axis from point sources of light on the axis and 1/8 and 1/4 degree off the axis. To make the patterns for each light source more visible they have been magnified by a factor of 20. Without this magnification they would have a size of about 2 pixels. The size of the central image is only about 50% larger than the Airy disc for this telescope. The other two images are about 2 and 3 times the Airy disc size. So the resolution of this telescope is quite good.

5.17 A 3D Demo

The example in this section illustrates how to display a rotating object in three dimensions (3D) with hidden surface removal. When compiled and run the program will create window containing a moving image similar to the following.

By pressing S you can select other possible objects to display, such as the following.
Whatever object is displayed, it will rotate with increasing speed but may be paused by pressing P and the orientation and speed of rotation may be reset by pressing R. The eye position may be moved further from the object by pressing F making it look smaller, and N moves the eye position closer. You can exit from the program by pressing Q.

An important aspect of the problem is how to represent the orientation of the object being displayed. For simplicity, let us assume the object to display is an aircraft with three embedded axes, \( \mathbf{t} \) in the direction of thrust, \( \mathbf{w} \) in the direction of the left (port) wing and \( \mathbf{l} \) in the direction of lift, assumed to be orthogonal to both \( \mathbf{t} \) and \( \mathbf{w} \). We will call the \( \mathbf{t} \), \( \mathbf{w} \) and \( \mathbf{l} \) the body axes, not to be confused with the real world axes \( \mathbf{x}, \mathbf{y} \) and \( \mathbf{z} \). For our purposes we will assume the world is not a sphere like the earth but flat with \( \mathbf{x} \) pointing north, \( \mathbf{y} \) pointing west and \( \mathbf{z} \) pointing up. The orientation of the aircraft can be specified in various ways. A common way is to use Euler angles which give the amount of rotation needed to move the aircraft from a state pointing north with wings level to the required orientation. The rotations are done in a defined order such as (1) rotate about axis \( \mathbf{w} \), then (2) rotate about axis \( \mathbf{l} \) and finally (3) rotate about axis \( \mathbf{t} \). By this means any orientation can be reached. But notice that the order in which the rotations are done is significant.

Another method, particularly favoured by implementors of flight simulators, is to use quaternions. These were discovered by an Irish mathematician William Rowan Hamilton in 1843. We are used to the idea of representing complex numbers in two dimensions with the \( \mathbf{i} \) axis orthogonal to the real axis, and we have seen multiplication of complex numbers can represent rotations and possible scaling in two dimensions. Quaternions are like complex numbers but in a higher number of dimensions. While complex numbers are typically written as \( a + ib \),
quaternions are written as $a + ib + jc + kd$. With complex numbers the $i$ axis is orthogonal to the real axis, but with quartenions the mind blowing idea is that $i$ is still orthogonal to the real axis but so are $j$ and $k$ and furthermore $i$, $j$ and $k$ are orthogonal to each other, so must live in a four dimensional space which is hard to visualise. As with complex numbers, multiplying by $i$ corresponds to a rotation of 90 degrees, and $i^2 = -1$. With quartenions, multiplying by $i$, $j$ and $k$ correspond to different rotations of 90 degrees and $i^2 = j^2 = k^2 = -1$. Furthermore, Hamilton’s major breakthrough was the realisation that $ijk$ also equals $-1$. He was so excited by this discovery that he could not resist the urge to carve $i^2 = j^2 = k^2 = ijk = -1$ into the stone of Brougham Bridge in Dublin. Unfortunately his carving is no longer visible. From these equations it is easy to deduce that $ij = k$, $ji = -k$, $jk = i$, $kj = -i$, $ki = j$ and $ik = -j$. Notice that $ij \neq ji$, so the algebra is not commuttive which is, of course, also true of rotations in three dimensions. If we multiply two quartenions $a_1 + b_1i + c_1j + d_1k$ by $a_2 + b_2i + c_2j + d_2k$ using the normal rules of algebra and simplify the result using the above equations, we obtain

\[
\begin{align*}
(a_1a_2 - b_1b_2 - c_1c_2 + d_1d_2) + \\
(a_1b_2 + b_1a_2 + c_1d_2 - d_1c_2)i + \\
(a_1c_2 - b_1d_2 + c_1a_2 + d_1b_2)j + \\
(a_1d_2 + b_1c_2 - c_1b_2 + d_1a_2)k
\end{align*}
\]

Just as any non zero complex number has an inverse that corresponds to undoing a rotation on 2D, any non zero quartenion also has an inverse corresponding to undoing a 3D rotation. Indeed, there are two inverses depending on whether pre- or post- multiplication is used.

Having just given a very brief introduction to quartenions with hints as to why they are useful for describing 3D rotations, I am going to drop the idea and use yet another mechanism for describing the orientation of the aircraft.

In the programs that follow, I use direction cosines. If we want to specify the direction of thrust $t$ we can use the coordinates of a point $T$ on the unit sphere centred at the origin $O$ with $OT$ parallel to the directions of thrust. In the programs that follows these coordinates are held in the variables $ctx$, $cty$ and $ctz$. They are called direction cosines because, for instance, $ctx$ is the cosine of the angle between the $x$ axis and the direction of thrust. The variables $cwx$, $cwy$ and $cwz$ hold the cosines for direction $w$ and $clx$, $cly$ and $clz$ hold the cosines for $l$. They are held as scaled numbers with 6 digits after the decimal point which provides adequate precision for our purposes. Using directions cosines may seem inefficient since they require 9 variables rather than the three for Euler angles or four for quartenions, but they are easier to understand and use, particularly for the calculations needed to plot instruments such as the artificial horizon or points on the ground as viewed by the pilot. The cost of performing rotations is insignificant compared to other computations performed by the flight simulator.
The program that drew the pictures given above is called `draw3d.b` and it starts as follows.

```bcpl
GET "libhdr"
GET "sdl.h"
GET "sdl.b" // Insert the library source code

GET "libhdr"
GET "sdl.h"

MANIFEST {
    One = 1.000000 // Direction cosines scaling factor
    // ie 6 decimal digits after the decimal point.
    Sps = 20 // Steps per second
}

GLOBAL {
    done:ug

    // =0 for an aircraft, =1 for a hollow cube
    object
    // =2 for coloured triangles
    stepping // =FALSE if not rotating the object

    c_elevator // Controls
    c_aileron
    c_rudder
    c_thrust

    ctx; cty; ctz // Direction cosines of direction t
    cwx; cwy; cwz // Direction cosines of direction w
    clx; cly; clz // Direction cosines of direction l

    cetx; cety; cetz // Eye direction cosines of direction t
    cewx; cewy; cewz // Eye direction cosines of direction w
    celx; cely; celz // Eye direction cosines of direction l

    eyex; eyey; eyez // Relative position of the eye
    eyedist // Eye x or y distance from aircraft

    rtdot; rwdot; rldot // Rotation rates about t, w and l axes

    // Rotational forces are scaled with 6 digits after the decimal point
    // as are direction cosines.
    rft // Rotational force about t axis
    rfw // Rotational force about w axis
```
5.17. A 3D DEMO

```c
rfl  // Rotational force about l axis

cdrawquad3d
cdrawtriangle3d
}

// Insert the definition of drawtigermoth()
GET "drawtigermoth.b"
```

As can be seen this inserts the BCPL source of the SDL library and then declares the global variables used in the program.

The variable `done` is set to `TRUE` when `Q` is pressed causing the program to terminate. The variable `object` specified which of four possible objects is to be drawn. The default value selects a representation of a tiger moth aircraft.

The variable `stepping` can be set to `FALSE` by pressing `P` to temporarily stop the displayed image being rotated.

As stated above the orientation of the displayed object is specified by direction cosines held in the variables such as `ctx`, `cty` and `ctz`. Direction cosines have a remarkable and particularly useful property which is as follows. Suppose `P` and `Q` are two points on the unit sphere with coordinates `(x, y, z)` and `(X, Y, Z)`, respectively, the expression `xX + yY + zZ` is called the inner product of `(x, y, z)` and `(X, Y, Z)` and is often written as `(x, y, z).(X, Y, Z)`. It turns out that its value is the cosine of the angle between the lines `OP` and `OQ`.

We can convince ourselves that this by the following observation. If we rotate `P` and `Q` about the `z`-axis by some arbitrary angle `α`, they move to new positions `P'` and `Q'` with coordinates

\[
\begin{align*}
(x \cos \alpha - y \sin \alpha, x \sin \alpha + y \cos \alpha, z) & \text{ and } (X \cos \alpha - Y \sin \alpha, X \sin \alpha + Y \cos \alpha, Z) \\
X \cos^2 \alpha - xY \cos \alpha \sin \alpha - yX \sin \alpha \cos \alpha + yY \sin^2 \alpha + zZ & = \\
xX \cos^2 \alpha + xY \sin \alpha \cos \alpha + yX \cos \alpha \sin \alpha + yY \cos^2 \alpha + zZ & = \\
xX(\cos^2 \alpha + \sin^2 \alpha) + yY(\cos^2 \alpha + \sin^2 \alpha) + zZ & = \\
xX + yY + zZ
\end{align*}
\]

So, if we take an arbitrary pair of points `P` and `Q` on the unit sphere and rotate them about the `z` axis until `Q` is in the `xz` plane, then rotate them about the
y-axis until $Q$ is on the $x$-axis and finally rotate them about the $x$-axis until $P$ is in the $xy$ plane. Assuming the angle between the original $OP$ and $OQ$ was $\theta$, the angle between their new positions will still be $\theta$ and so the new coordinates of $P$ and $Q$ will be $(\cos \theta, \sin \theta, 0)$ and $(1, 0, 0)$, and their inner product will be $\cos \theta \times 1 + \sin \theta \times 0 + 0 \times 0 = \cos \theta$. This confirms that the inner product of two sets of direction cosines is the cosine of the angle between the two directions they specify.

The BCPL function to calculate the inner product is defined as follows.

```c
LET inprod(a,b,c, x,y,z) =
  // Return the cosine of the angle between two unit vectors.
  muldiv(a, x, One) + muldiv(b, y, One) + muldiv(c, z, One)
```

This function assumes that $x$, $y$ and $z$ are direction cosines represented by scaled numbers with 6 digits after the decimal point. The manifest constant $One=1_000000$ represents one in this representation. This function is used in the definition of `rotate` given below.

```c
AND rotate(t, w, l) BE
{
  // Rotate the orientation of the aircraft
  // t, w and l are assumed to be small and cause
  // rotation about axis t, w, l. Positive values cause
  // anti-clockwise rotations about their axes.

  LET tx = inprod(One, -l, w, ctx,cwx,clx)
  LET wx = inprod( l,One, -t, ctx,cwx,clx)
  LET lx = inprod( -w, t,One, ctx,cwx,clx)

  LET ty = inprod(One, -l, w, cty,cwy,cly)
  LET wy = inprod( l,One, -t, cty,cwy,cly)
  LET ly = inprod( -w, t,One, cty,cwy,cly)

  LET tz = inprod(One, -l, w, ctz,cwz,clz)
  LET wz = inprod( l,One, -t, ctz,cwz,clz)
  LET lz = inprod( -w, t,One, ctz,cwz,clz)

  ctx, cty, ctz := tx, ty, tz
  cwx, cwy, cwz := wx, wy, wz
  clx, cly, clz := lx, ly, lz
}
```
This function is used to make small changes to the orientation of the object being displayed. For simplicity we will assume the object being displayed is an aircraft. Embedded in the aircraft are three axes: \( t \) with direction cosines \( \text{ctx} \), \( \text{cty} \) and \( \text{ctz} \), \( w \) with direction cosines \( \text{cw}x \), \( \text{cw}y \) and \( \text{cw}z \), and \( l \) with direction cosines \( \text{cl}x \), \( \text{cl}y \) and \( \text{cl}z \). The arguments of \( t \), \( w \) and \( l \) are in radians specifying small anti-clockwise rotations about the \( t \)-, \( w \)- and \( l \)- axes, respectively. These angles are also scaled with 6 digits after the decimal point. The variables \( \text{tx} \), \( \text{ty} \) and \( \text{tz} \) are approximately the directions cosines of axis \( t \) after the rotation, and these are calculated using suitable calls of \text{inprod}. Consider the call \text{inprod}(\text{One},-\text{a}, \text{w}, \text{ctx}, \text{cw}x, \text{cl}x) \) that defines \( \text{tx} \) which will be the new value of \( \text{ctx} \). To see how it works, consider the effect of \(-\text{a}\) in the inner product. This should give the amount by which \( \text{ctx} \) is increased due to the small rotation about the \( l \) axis. The \text{inprod} call computes this increase as \text{muldiv}(-\text{a}, \text{cw}x, \text{One}) \). If \( T \) is the point on the unit sphere in direction \( t \) from the origin, then this small rotation will move it to a point \( T' \) in the \( tw \) plane by a distance \( r \). If \( \theta \) is the angle between \( TT' \) and the \( yz \) plane, then the change in \( \text{ctx} \) will be \( r \sin \theta \), but \( TT' \) is parallel to the \( w \) axis and so \( \sin \theta \) is equal to \( \text{cw}x \). Thus the magnitude of the change is \( r \) multiplied by \( \text{cw}x \) and since the rotation was anti-clockwise this value is negated. The other rotations may be checked in the same way.

Since the calculations will inevitably be approximate, two adjustments are made to the new direction cosines. The calls \text{adjustlength} to attempt to ensure the direction cosines remain of unit length, and the calls \text{adjustortho} that attempts to keep the three direction cosines mutually orthogonal. These functions are defined as follows.

\[
\text{AND adjustlength}(v) \ \text{BE} \ \\
\{ \ // \text{This helps to keep vector } v \text{ of unit length} \ \\
\ \ \ \text{LET } x, y, z = v!0, v!1, v!2 \ \\
\ \ \ \text{LET corr} = \text{One} + (\text{inprod}(x, y, z, x, y, z) - \text{One})/2 \ \\
\ \ \ \text{v!0 := muldiv(x, One, corr)} \ \\
\ \ \ \text{v!1 := muldiv(y, One, corr)} \ \\
\ \ \ \text{v!2 := muldiv(z, One, corr)} \ \\
\}\ \\
\]

If we write the distance from \( O \) to \( (x, y, z) \) as \((1 + \epsilon)\), the call \text{inprod}(x, y, z, x, y, z) \) yields the square of this length, namely \((1 + \epsilon)^2\) which equals \((1 + 2\epsilon + \epsilon^2)\). Provided \( \epsilon \) is small this is approximately \((1 + 2\epsilon)\) and so an estimate of \( \epsilon \) is \((\text{inprod}(x, y, z, x, y, z) - \text{One})/2\). The length correction requires us to divide \( x \) by \((1 + \epsilon)\) which is exactly what \( v!0 := \text{muldiv}(x, \text{One}, \text{corr}) \) does since \( \text{corr} \) is set to \((1 + \epsilon)\). The corrections to \( y \) and \( z \) are done in the same way.

The function \text{adjustortho} is defined as follows.

\[
\text{AND adjustortho}(a, b) \ \text{BE} \ \\
\]

If we write the distance from \( O \) to \( (x, y, z) \) as \((1 + \epsilon)\), the call \text{inprod}(x, y, z, x, y, z) \) yields the square of this length, namely \((1 + \epsilon)^2\) which equals \((1 + 2\epsilon + \epsilon^2)\). Provided \( \epsilon \) is small this is approximately \((1 + 2\epsilon)\) and so an estimate of \( \epsilon \) is \((\text{inprod}(x, y, z, x, y, z) - \text{One})/2\). The length correction requires us to divide \( x \) by \((1 + \epsilon)\) which is exactly what \( v!0 := \text{muldiv}(x, \text{One}, \text{corr}) \) does since \( \text{corr} \) is set to \((1 + \epsilon)\). The corrections to \( y \) and \( z \) are done in the same way.

The function \text{adjustortho} is defined as follows.
{ // This helps to keep the unit vector \( \mathbf{b} \) orthogonal to \( \mathbf{a} \)
    LET a0, a1, a2 = a!0, a!1, a!2
    LET b0, b1, b2 = b!0, b!1, b!2
    LET corr = \text{inprod}(a0,a1,a2, b0,b1,b2)
    b!0 := b0 - muldiv(a0, corr, One)
    b!1 := b1 - muldiv(a1, corr, One)
    b!2 := b2 - muldiv(a2, corr, One)
}

In this function, the call \text{inprod}(a0,a1,a2, b0,b1,b2) computes a value that will be zero if the two sets of direction cosines are orthogonal. If not zero, the correction should be small and this proportion of \( \mathbf{a} \) is subtracted from \( \mathbf{b} \).

To demonstrate the need for these two corrections, try commenting out the calls of \text{adjustlength} and \text{adjustortho}. You will find that the images generated by \text{draw3d} soon get seriously distorted.

The object being displayed is rotating at a rate held in the variables \( rtdot \), \( rwdot \) and \( rldot \). These are scaled values with six digits after the decimal point representing anti-clockwise rotations rates about the \( t \), \( w \) and \( l \) axes in radians per second. The orientation of the object is updated many times per second by calls of \text{step}. Its definition is as follows.

LET step() BE
{
    LET step() BE
    { // Apply rotational forces
        rtdot := -c_aileron * 200 / Sps
        rwdot := -c_elevator * 200 / Sps
        rldot := c_rudder * 200 / Sps

        rotate(rtdot/Sps, rwdot/Sps, rldot/Sps)
    }
}

The number of times \text{step} is called per second is held in \text{Sps}. So on each call the angle of rotation about the \( t \)-axis is \( rtdot/Sps \). The rotational angles for the other two axes are calculated in the same way. Every time \text{step} is called the rotational rates are adjusted by rotational forces held in \text{rft}, \text{rfw} and \text{rfl}. These are in units of radians per second per second and are adjusted to suit the stepping rate. In a flight simulator these forces depend on the speed and direction of the airflow around the aircraft and the setting of the flying controls such as the elevator or rudder. In \text{draw3d.b} these controls can be modified using the arrow keys and the characters '<' and '>'. The distance between the eye and the object can be modified by pressing 'F' and 'N'.

The object is displayed by calling \text{plotcraft} defined as follows.
AND plotcraft() BE
   { IF depthscreen FOR i = 0 TO screenxsize*screenysize-1 DO
      depthscreen!i := maxint

   IF object=0 DO
      { // Simple aircraft
         setcolour(maprgb(64,128,64)) // Fuselage
         cdrawtriangle3d(6_000,0,0, 2_000,0,-1_000, -2_000,0,2_000)
         setcolour(maprgb(40,100,40))
         cdrawtriangle3d(2_000,0,-1_000, -2_000,0,2_000, -12_000,0,0)
         setcolour(maprgb(255,255,255))
         cdrawtriangle3d(2_000,0, 1_000, -2_000,0,2_000, 0_800,0,2_000)
         setcolour(maprgb(255,0,0)) // Port wing -- Red
         cdrawtriangle3d(2_500,0,0, -2_500,0,0, -2_000, 18_000,2_000)
         setcolour(maprgb(0,255,0)) // Starboard wing -- Green
         cdrawtriangle3d(2_500,0,0, -2_500,0,0, -2_000,-18_000,2_000)
         setcolour(maprgb(255,0,255)) // Stabliser
         cdrawtriangle3d(-9_000,0,0, -12_000,0,0, -13_000,-4_000,0)
         setcolour(maprgb(255,255,0))
         cdrawtriangle3d(-9_000,0,0, -12_000,0,0, -13_000, 4_000,0)
         setcolour(maprgb(0,255,255)) // Fin
         cdrawtriangle3d(-9_000,0,0, -12_000,0,0, -13_000,0,4_000)
      }

   IF object=1 DO
      { // Create a coloured cube with side length 2s
         LET s = 10_000
         setcolour(maprgb(0,0,0)) // Front
         cdrawquad3d(s,-s,s, s,s,s, s,-s, s,-s,-s)
         setcolour(maprgb(255,255,255)) // Back
         cdrawquad3d(-s,-s,s, -s,s,s, -s,-s, -s,-s,-s)
         setcolour(maprgb(255,0,0)) // Left
         cdrawquad3d( s,s,s, s,s,-s, s,-s,-s, -s,-s,-s)
         setcolour(maprgb(0,255,0)) // Right
         cdrawquad3d( s,-s,s, s,-s,-s, -s,-s,-s, -s,-s,-s)
      }

   IF object=2 DO
      { LET s = 10_000
         LET r = muldiv(s, c_thrust, 32768)
// top
setcolour(maprgb(0,0,0))
cdrawquad3d( r,0,s, 0,r,s, -r,0,s, 0,-r,s)

// top wings
setcolour(maprgb(255,0,0))
cdrawtriangle3d( r, 0, s, s, 0, s, s, 0, r) // N
setcolour(maprgb(0,255,0))
cdrawtriangle3d( 0, r, s, 0, s, 0, s, r) // W
setcolour(maprgb(255,0,0))
cdrawtriangle3d(-r, 0, s, -s, 0, s, -s, 0, r) // S
setcolour(maprgb(0,255,0))
cdrawtriangle3d( 0,-r, s, 0,-s, s, 0,-s, r) // E

// Sides
setcolour(maprgb(128,0,0))
cdrawquad3d(s,0,r, s,r,0, s,0,-r, s,-r,0) // N
setcolour(maprgb(255,128,0))
cdrawquad3d(0,s,r, r,s,0, 0,s,-r, -r,s,0) // W
setcolour(maprgb(255,0,128))
cdrawquad3d(-s,0,r, -s,r,0, -s,0,-r, -s,-r,0) // S
setcolour(maprgb(255,128,128))
cdrawquad3d(0,-s,r, r,-s,0, 0,-s,-r, -r,-s,0) // W

// Centre wings
setcolour(maprgb(255,128,0))
cdrawtriangle3d( s, s, 0, r, s, 0, s, r, 0) // NW
setcolour(maprgb(0,255,128))
cdrawtriangle3d(-s, s, 0, -s, r, 0, -r, s, 0) // SW
setcolour(maprgb(128,0,255))
cdrawtriangle3d(-s,-s, 0, -r,-s, 0, -s,-r, 0) // SE
setcolour(maprgb(127,255,255))
cdrawtriangle3d( s,-s, 0, s,-r, 0, r,-s, 0) // NE

// bottom wings
setcolour(maprgb(255,0,0))
cdrawtriangle3d( r, 0,-s, s, 0,-s, s, 0,-r) // N
setcolour(maprgb(0,255,0))
cdrawtriangle3d( 0, r,-s, 0, s,-s, 0, s,-r) // W
setcolour(maprgb(255,0,255))
cdrawtriangle3d(-r, 0,-s, -s, 0,-s, -s, 0,-r) // S
This function inspects object to see which object to draw drawing it with successive calls of `cdrawquad3d` or `cdrawtriangle3d`. The objects are specified using body coordinates in directions t, w and l, using values representing feet scaled with three digits after the decimal point.

The function `cdrawquad3d` draws a 3D quadrilateral by first rotating the coordinates using suitable calls of `inprod` then transforming them to screen coordinates using `screencoords` before plotting the quadrilateral using the library function `drawquad3d`, defined in `sdl.b`. 

```plaintext
AND cdrawquad3d(x1,y1,z1, x2,y2,z2, x3,y3,z3, x4,y4,z4) BE
{ LET rx1 = inprod(x1,y1,z1, ctx,cwx,clx)
  LET ry1 = inprod(x1,y1,z1, cty,cwy,clx)
  LET rz1 = inprod(x1,y1,z1, ctz,cwz,clz)

  LET rx2 = inprod(x2,y2,z2, ctx,cwx,clx)
  LET ry2 = inprod(x2,y2,z2, cty,cwy,clx)
  LET rz2 = inprod(x2,y2,z2, ctz,cwz,clz)

  LET rx3 = inprod(x3,y3,z3, ctx,cwx,clx)
  LET ry3 = inprod(x3,y3,z3, cty,cwy,clx)
  LET rz3 = inprod(x3,y3,z3, ctz,cwz,clz)

  LET rx4 = inprod(x4,y4,z4, ctx,cwx,clx)
  LET ry4 = inprod(x4,y4,z4, cty,cwy,clx)
  LET rz4 = inprod(x4,y4,z4, ctz,cwz,clz)

  LET sx1,sy1,sz1 = ?,?,?,?
  LET sx2,sy2,sz2 = ?,?,?,?
  LET sx3,sy3,sz3 = ?,?,?,?
  LET sx4,sy4,sz4 = ?,?,?,?
```
FUNCTIONU UNLESS screencoords(rx1-eyex, ry1-eyey, rz1-eyez, @sx1) RETURN
FUNCTIONU UNLESS screencoords(rx2-eyex, ry2-eyey, rz2-eyez, @sx2) RETURN
FUNCTIONU UNLESS screencoords(rx3-eyex, ry3-eyey, rz3-eyez, @sx3) RETURN
FUNCTIONU UNLESS screencoords(rx4-eyex, ry4-eyey, rz4-eyez, @sx4) RETURN

{ drawquad3d(sx1,sy1,sz1, sx2,sy2,sz2, sx3,sy3,sz3, sx4,sy4,sz4) }

AND cdrawtriangle3d(x1,y1,z1, x2,y2,z2, x3,y3,z3) BE
{ LET rx1 = inprod(x1,y1,z1, ctx,cwx,clx)
  LET ry1 = inprod(x1,y1,z1, cty,cwy,cly)
  LET rz1 = inprod(x1,y1,z1, ctz,cwz,clz)

  LET rx2 = inprod(x2,y2,z2, ctx,cwx,clx)
  LET ry2 = inprod(x2,y2,z2, cty,cwy,cly)
  LET rz2 = inprod(x2,y2,z2, ctz,cwz,clz)

  LET rx3 = inprod(x3,y3,z3, ctx,cwx,clx)
  LET ry3 = inprod(x3,y3,z3, cty,cwy,cly)
  LET rz3 = inprod(x3,y3,z3, ctz,cwz,clz)

  LET sx1,sy1,sz1 = ?,?,?
  LET sx2,sy2,sz2 = ?,?,?
  LET sx3,sy3,sz3 = ?,?,?

  UNLESS screencoords(rx1-eyex, ry1-eyey, rz1-eyez, @sx1) RETURN
  UNLESS screencoords(rx2-eyex, ry2-eyey, rz2-eyez, @sx2) RETURN
  UNLESS screencoords(rx3-eyex, ry3-eyey, rz3-eyez, @sx3) RETURN

  drawtriangle3d(sx1,sy1,sz1, sx2,sy2,sz2, sx3,sy3,sz3) }

AND cdrawtriangle3d(x1,y1,z1, x2,y2,z2, x3,y3,z3) BE
{ LET rx1 = inprod(x1,y1,z1, ctx,cwx,clx)
  LET ry1 = inprod(x1,y1,z1, cty,cwy,cly)
  LET rz1 = inprod(x1,y1,z1, ctz,cwz,clz)

  LET rx2 = inprod(x2,y2,z2, ctx,cwx,clx)
  LET ry2 = inprod(x2,y2,z2, cty,cwy,cly)
  LET rz2 = inprod(x2,y2,z2, ctz,cwz,clz)

  LET rx3 = inprod(x3,y3,z3, ctx,cwx,clx)
  LET ry3 = inprod(x3,y3,z3, cty,cwy,cly)
  LET rz3 = inprod(x3,y3,z3, ctz,cwz,clz)

  LET sx1,sy1,sz1 = ?,?,?
  LET sx2,sy2,sz2 = ?,?,?
  LET sx3,sy3,sz3 = ?,?,?

  UNLESS screencoords(rx1-eyex, ry1-eyey, rz1-eyez, @sx1) RETURN
  UNLESS screencoords(rx2-eyex, ry2-eyey, rz2-eyez, @sx2) RETURN
  UNLESS screencoords(rx3-eyex, ry3-eyey, rz3-eyez, @sx3) RETURN

  drawtriangle3d(sx1,sy1,sz1, sx2,sy2,sz2, sx3,sy3,sz3) }

Both cdrawquad3d and cdrawtriangle3d use screencoords to transform the rotated coordinates of an object to screen coordinates, taking account of the orientation of the observer's eye held in directions cosines such as cetx, cewx and celx.

AND screencoords(x,y,z, v) = VALOF
{ // If the point (x,y,z) is in view, set v!0, v!1 and v!2 to
  // the screen coordinates and depth and return TRUE

5.17. A 3D DEMO

// otherwise return FALSE
LET sx = inprod(x,y,z, cewx,cewy,cewz) // Horizontal
LET sy = inprod(x,y,z, celx,cely,celz) // Vertical
LET sz = inprod(x,y,z, cetx,cety,cetz) // Depth
LET screensize = screenxsize>=screenysize -> screenxsize, screenysize

// Test that the point is in view, ie at least 1.000ft in front
// and no more than about 27 degrees (inverse tan 1/2) from the
// direction of view.
IF sz<1_000 &
    muldiv(sz, sz, 2000) >= muldiv(sx, sx, 1000) + muldiv(sy, sy, 1000)
RESULTIS FALSE

// A point screensize pixels away from the centre of the screen is
// 45 degrees from the direction of view.
// Note that many pixels in this range are off the screen.
v!0 := -muldiv(sx, screensize, sz) + screenxsize/2
v!1 := +muldiv(sy, screensize, sz) + screenysize/2
v!2 := sz // This distance into the screen in arbitrary units, used
       // for hidden surface removal.
RESULTIS TRUE
}

The arguments x, y, z are the coordinates of a point relative to the position of
the eye. As can be seen, screencoords checks that the point in at least one foot
in front of the observer and no more than about 27 degrees from the direction of
view. If successful it updates the three elements of vector v with the horizontal,
vertical and depth screen coordinates of the point, returning TRUE to indicate
success. Otherwise it returns FALSE. The depth coordinate is used by the low
level plotting functions to conditionally remove points obscured by a previously
drawn points.

The function plotscreen is called every time the screen has to be updated.
It first fills it with a light blue colour, then sets the eye position and orientation
before plotting calling plotcraft to draw the object.

AND plotscreen() BE
{ fillscreen(maprgb(100,100,255))
    seteyeposition()
    plotcraft()
}

In this program, the orientation of the eye is always looking horizontally due
north and is positioned at a distance eyedist due south of the centre of the
object. As described above, this distance can be adjusted by typing F or N.
AND seteyeposition() BE
{  cctx, ccty, cctz := One, 0, 0
    cewx, cewy, cewz := 0, One, 0
    celx, cely, celz := 0, 0, One
    eyex, eyey, eyez := -eyedist, 0, 0  // Relative eye position
}

The program is controlled using the mouse and keyboard. These interactions are dealt with by processevents whose definition is as follows.

AND processevents() BE WHILE getevent() SWITCH ON eventtype INTO
{  DEFAULT:  LOOP
  CASE sdle_keydown:
    SWITCH ON capitalch(eventa2) INTO
{  DEFAULT:  LOOP
    CASE 'Q':  done := TRUE
      LOOP
    CASE 'S':  // Select next object to display
      object := (object + 1) MOD 4  LOOP
    CASE 'P':  // Toggle stepping
      stepping := ~stepping  LOOP
    CASE 'R':  // Reset the orientation and rotation rate
      cctx, ccty, cctz := One, 0, 0
      cewx, cewy, cewz := 0, One, 0
      celx, cely, celz := 0, 0, One
      rtdot, rwdot, rldot := 0, 0, 0  LOOP
    CASE 'N':  // Reduce eye distance
      eyedist := eyedist*5/6
      IF eyedist<65_000 D0 eyedist := 65_000  LOOP
    CASE 'F':  // Increase eye distance
      eyedist := eyedist*6/5  LOOP
CASE 'Z': c_thrust := c_thrust - 2048
       IF c_thrust < 0 DO c_thrust := 0
       writeln("c_thrust=%n", c_thrust)
       LOOP

CASE 'X': c_thrust := c_thrust + 2048
       IF c_thrust > 32768 DO c_thrust := 32768
       writeln("c_thrust=%n", c_thrust)
       LOOP

CASE ',',: CASE '<': c_rudder := c_rudder - 4096
       IF c_rudder < -32768 DO c_rudder := -32768
       writeln("c_rudder=%n", c_rudder)
       LOOP

CASE '.': CASE '>': c_rudder := c_rudder + 4096
       IF c_rudder > 32768 DO c_rudder := 32768
       writeln("c_rudder=%n", c_rudder)
       LOOP

CASE sdle_arrowup:
       c_elevator := c_elevator + 4096
       IF c_elevator > 32768 DO c_elevator := 32768
       writeln("c_elevator=%n", c_elevator)
       LOOP

CASE sdle_arrowdown:
       c_elevator := c_elevator - 4096
       IF c_elevator < -32768 DO c_elevator := -32768
       writeln("c_elevator=%n", c_elevator)
       LOOP

CASE sdle_arrowsright:
       c_aileron := c_aileron + 4096
       IF c_aileron > 32768 DO c_aileron := 32768
       writeln("c_aileron=%n", c_aileron)
       LOOP

CASE sdle_arrowsleft:
       c_aileron := c_aileron - 4096
       IF c_aileron < -32768 DO c_aileron := -32768
       writeln("c_aileron=%n", c_aileron)
       LOOP

CASE sdle_quit:
writef("QUIT\n");
done := TRUE
LOOP
}

Events are read by calls of getevent which returns TRUE whenever another event is present. The type of event is placed in eventtype. If it is a key down event from the keyboard eventtype=sdle_keydown and eventa2 identifies which key was pressed. The SWITCHON command has cases for each key that affects to program. The code for each is easy to follow. All other keys are ignored at the DEFAULT label. The only mouse event to be handled has type sdle_quit caused by clicking on the little cross at the top right hand corner of the window. As can be seen this sets done to TRUE causing the program to terminate.

Finally, there is the main program start which initialises the variables used by the program, creates a window entitled Draw 3D Demo and enters the main processing loop which repeatedly calls processevents to deal with keyboard and mouse events, before conditionally calling step to rotate the object, followed by calls plotscreen and updatescreen to draw the new state of the object and send it to the display hardware. It then issues a short delay before going round the loop again. It only leaves the loop when done becomes TRUE. This delays briefly before closing the SDL window and terminating the program. The definition of start is as follows.

LET start() = VALOF
{ // The initial direction cosines giving the orientation of
  // the object.
  ctx, cty, ctz := One, 0, 0 // The cosines are scaled with
  cwx, cwy, cwz := 0, One, 0 // six decimal digits
  clx, cly, clz := 0, 0, One // after to decimal point.

  eyedist := 120_000 // Eye distance from the object.
  object := 3 // Tigermoth
  stepping := TRUE
  // Initial rate of rotation about each axis
  rtdot, rwdot, rldot := 0, 0, 0
  c_elevator, c_aileron, c_rudder, c_thrust := -4096*4, 4096*3, 4096*5, 10240

  initsdl()
  mkscreen("Draw 3D Demo", 800, 500)

  done := FALSE

  UNTIL done DO
  { processevents()
5.18. **DRAWTIGERMOTH.B**

A tigermoth is a biplane designed in the 1930s and used for initial pilot training until about 1946. Many still exist and one owned by the Cambridge Flying Group is as follows.

Since I once had a pilot’s licence for the tigermoth, I thought I would implement a simple tigermoth flight simulator. The flight simulator needs a computer model of the aircraft and this is implemented in the file `drawtigermoth.b` which defines the function `drawtigermoth`. It was developed using `draw3d.b` and is in a separate file so that it can be inserted into programs by the directive `GET "drawtigermoth.b"`.

A typical image of this tigermoth model is as follows.
The definition of `drawtigermoth` is as follows.

```plaintext
LET drawtigermoth() BE
{
    // The origin is the centre of gravity
    // All measurements are in feet scaled with three
    // digits after the decimal point.

    // Cockpit floor
    setcolour(maprgb(90,80,30))
    cdrawquad3d (1_000, 0_800, 0_000,
                 1_000,-0_800, 0_000,
                 -5_800,-0_800, 0_000,
                 -5_800, 0_800, 0_000)

    // Left lower wing
    setcolour(maprgb(165,165,30))  // Under surface
    cdrawquad3d(-0_500, 1_000, -2_000, // Panel A
                -3_767, 1_000, -2_218,
                -4_396, 6_000, -1_745,
                -1_129, 6_000, -1_527)

    cdrawquad3d(-3_767, 1_000, -2_218, // Panel B
                -4_917, 1_000, -2_294,
                -5_546, 6_000, -1_821,
                -4_396, 6_000, -1_745)

    cdrawquad3d(-1_129, 6_000, -1_527, // Panel C
                -5_800, 0_800, 0_000,
                -5_800,-0_800, 0_000,
                1_000,-0_800, 0_000,
                1_000, 0_800, 0_000)
}
```
\[ \text{LET } a = \text{muldiv}(0.600, c_{ailerons}, 32768 \times 17) \]

\[
\{ \text{// Aileron deflection 1 inch from hinge}
\text{setcolour(maprgb(155,155,20)) // Under surface}
\text{cdrawquad3d(-4.396, 6.000, -1.745, // Panel D Aileron}
-5.546+3*a, 6.000, -1.821-14*a,
-6.297+3*a, 13.766, -1.255-14*a,
-5.147, 14.166, -1.179)
\}
\]

\[
\text{// Left lower wing upper surface}
\text{setcolour(maprgb(120,140,60))}
\text{cdrawquad3d(-0.500, 1.000, -2.000, // Panel A1}
-1.500, 1.000, -1.800,
-2.129, 6.000, -1.327,
-1.129, 6.000, -1.527)
\]

\[
\text{setcolour(maprgb(120,130,50))}
\text{cdrawquad3d(-1.500, 1.000, -1.800, // Panel A2}
-3.767, 1.000, -2.118,
-4.396, 6.000, -1.645,
-2.129, 6.000, -1.327)
\]

\[
\text{cdrawquad3d(-3.767, 1.000, -2.118, // Panel B}
-4.917, 1.000, -2.294,
-5.546, 6.000, -1.821,
-4.396, 6.000, -1.645)
\]

\[
\text{setcolour(maprgb(120,140,60))}
\text{cdrawquad3d(-1.129, 6.000, -1.527, // Panel C1}
-2.129, 6.000, -1.327,
-2.880, 14.166, -0.761,
-1.880, 14.166, -0.961)
\]

\[
\text{setcolour(maprgb(120,130,50))}
\text{cdrawquad3d(-2.129, 6.000, -1.327, // Panel C2}
-4.396, 6.000, -1.645,
-5.147, 14.166, -1.079,
-2.880, 14.166, -0.761)
\]
{ // Aileron deflection 1 inch from hinge
LET a = muldiv(0.600, c_aileron, 32_768*17)

setcolour(maprgb(120,140,60))
cdrawquad3d(-4_396, 6_000, -1_645, // Panel D Aileron
       -5_546+3*a, 6_000, -1_821-14*a,
       -6_297+3*a, 13_766, -1_255-14*a,
       -5_147,        14_166, -0_979)
}

// Left lower wing tip
setcolour(maprgb(130,150,60))
cdrawtriangle3d(-1_880, 14_167,-1_006,
       -2_880, 14_167,-0_761,
       -3_880, 14_467,-0_980)
setcolour(maprgb(130,150,60))
cdrawtriangle3d(-2_880, 14_167,-0_761,
       -5_147, 14_167,-1_079,
       -3_880, 14_467,-0_980)
setcolour(maprgb(160,160,40))
cdrawtriangle3d(-5_147, 14_167,-1_079,
       -5_147, 14_167,-1_179,
       -3_880, 14_467,-0_980)
setcolour(maprgb(170,170,50))
cdrawtriangle3d(-5_147, 14_167,-1_179,
       -1_880, 14_167,-0_961,
       -3_880, 14_467,-0_980)

// Right lower wing
setcolour(maprgb(165,165,30))  // Under surface

cdrawquad3d(-0_500, -1_000, -2_000, // Panel A
       -3_767, -1_000, -2_218,
       -4_396, -6_000, -1_745,
       -1_129, -6_000, -1_527)

cdrawquad3d(-3_767, -1_000, -2_218, // Panel B
       -4_917, -1_000, -2_294,
       -5_546, -6_000, -1_821,
       -4_396, -6_000, -1_745)

cdrawquad3d(-1_129, -6_000, -1_527, // Panel C
       -4_396, -6_000, -1_745,
       -5_147, -14_166, -1_179,
       -1_880, -14_166, -0_961)
{ // Aileron deflection 1 inch from hinge
    LET a = muldiv(0.600, c_aileron, 32_768*17)

    setcolour(maprgb(155,155,20)) // Under surface
    cdrawquad3d(-4.396, -6.000, -1.745, // Panel D Aileron
                    -5.546+3*a, -6.000, -1.821+14*a,
                    -6.297+3*a,-13.766, -1.255+14*a,
                    -5.147,     -14.166, -1.179)
}

// Right lower wing upper surface
setcolour(maprgb(120,140,60))
    cdrawquad3d(-0.500, -1.000, -2.000, // Panel A1
                    -1.500, -1.000, -1.800,
                    -2.129, -6.000, -1.327,
                    -1.129, -6.000, -1.527)

    setcolour(maprgb(120,130,50))
    cdrawquad3d(-1.500, -1.000, -1.800, // Panel A2
                    -3.767, -1.000, -2.118,
                    -4.396, -6.000, -1.645,
                    -2.129, -6.000, -1.327)

    cdrawquad3d(-3.767, -1.000, -2.118, // Panel B
                    -4.917, -1.000, -2.294,
                    -5.546, -6.000, -1.821,
                    -4.396, -6.000, -1.645)

    setcolour(maprgb(120,140,60))
    cdrawquad3d(-1.129, -6.000, -1.527, // Panel C1
                    -2.129, -6.000, -1.327,
                    -2.880,-14.166, -0.761,
                    -1.880,-14.166, -0.961)

    setcolour(maprgb(120,130,50))
    cdrawquad3d(-2.129, -6.000, -1.327, // Panel C2
                    -4.396, -6.000, -1.645,
                    -5.147,-14.166, -1.079,
                    -2.880,-14.166, -0.761)

} // Aileron deflection 1 inch from hinge
    LET a = muldiv(0.600, c_aileron, 32_768*17)
setcolour(maprgb(120,140,60))
cdrawquad3d(-4_396, -6_000, -1_645, // Panel D Aileron
-5_546+3*a, -6_000, -1_821+14*a,
-6_297+3*a,-13_766, -1_255+14*a,
-5_147, -14_166, -0_979)
}

// Right lower wing tip
setcolour(maprgb(130,150,60))
cdrawtriangle3d(-1_880,-14_167,-1_006,
-2_880,-14_167,-0_761,
-3_880,-14_467,-0_980)
setcolour(maprgb(130,150,60))
cdrawtriangle3d(-2_880,-14_167,-0_761,
-5_147,-14_167,-1_079,
-3_880,-14_467,-0_980)
setcolour(maprgb(160,160,40))
cdrawtriangle3d(-5_147,-14_167,-1_079,
-5_147,-14_167,-1_179,
-3_880,-14_467,-0_980)
setcolour(maprgb(170,170,50))
cdrawtriangle3d(-5_147,-14_167,-1_179,
-1_880,-14_167,-0_961,
-3_880,-14_467,-0_980)

// Left upper wing
setcolour(maprgb(200,200,30)) // Under surface
cdrawquad3d( 1_333, 1_000, 2_900,
-1_967, 1_000, 2_671,
-3_297, 14_167, 3_671,
0_003, 14_167, 3_894)
cdrawquad3d(-1_967, 1_000, 2_671,
-3_084, 2_200, 2_606,
-4_414, 13_767, 3_645,
-3_297, 14_167, 3_671)
setcolour(maprgb(150,170,90)) // Top surface
cdrawquad3d( 1_333, 1_000, 2_900, // Panel A1
0_333, 1_000, 3_100,
-0_997, 14_167, 4_094,
0_003, 14_167, 3_894)
setcolour(maprgb(140,160,80)) // Top surface
cdrawquad3d( 0_333, 1_000, 3_100, // Panel A2
-1_967, 1_000, 2_771,
-3.297, 14.167, 3.771,  
-0.997, 14.167, 4.094)

setcolour(maprgb(150,170,90)) // Top surface
cdrawquad3d(-1.967, 1.000, 2.771, // Panel B
-3.084, 2.200, 2.606,
-4.414, 13.767, 3.645,
-3.297, 14.167, 3.771)

// Left upper wing tip
setcolour(maprgb(130,150,60))
cdrawtriangle3d( 0.003, 14.167, 3.894,
-0.997, 14.167, 4.094,
-1.997, 14.467, 3.874)
setcolour(maprgb(130,150,60))
cdrawtriangle3d(-0.997, 14.167, 4.094,
-3.297, 14.167, 3.771,
-1.997, 14.467, 3.874)
setcolour(maprgb(160,160,40))
cdrawtriangle3d(-3.297, 14.167, 3.771,
-3.297, 14.167, 3.671,
-1.997, 14.467, 3.874)
setcolour(maprgb(170,170,50))
cdrawtriangle3d(-3.297, 14.167, 3.671,
0.003, 14.167, 3.894,
-1.997, 14.467, 3.874)

// Right upper wing
setcolour(maprgb(200,200,30)) // Under surface
cdrawquad3d( 1.333, -1.000, 2.900,
-1.967, -1.000, 2.671,
-3.297, -14.167, 3.671,
0.003, -14.167, 3.894)
cdrawquad3d(-1.967, -1.000, 2.671,
-3.084, -2.200, 2.606,
-4.414, -13.767, 3.645,

setcolour(maprgb(150,170,90)) // Top surface
cdrawquad3d( 1.333, -1.000, 2.900, // Panel A1
0.333, -1.000, 3.100,
-0.997, -14.167, 4.094,
0.003, -14.167, 3.894)
CHAPTER 5. INTERACTIVE GRAPHICS IN BCPL USING SDL

```c
setcolour(maprgb(140,160,80)) // Top surface
cdrawquad3d( 0.333, -1.000, 3.100, // Panel A2
              -1.967, -1.000, 2.771,
              -3.297,-14.167, 3.771,
              -0.997,-14.167, 4.094)

setcolour(maprgb(150,170,90)) // Top surface
cdrawquad3d(-1.967, -1.000, 2.771, // Panel B
             -3.084, -2.200, 2.606,
             -4.414,-13.767, 3.645,
             -3.297,-14.167, 3.771)

// Right upper wing tip
setcolour(maprgb(130,150,60))
cdrawtriangle3d( 0.003,-14.167, 3.894,
                 -0.997,-14.167, 4.094,
                 -1.997,-14.467, 3.874)
setcolour(maprgb(130,150,60))
cdrawtriangle3d(-0.997,-14.167, 4.094,
                 -3.297,-14.167, 3.771,
                 -1.997,-14.467, 3.874)
setcolour(maprgb(160,160,40))
cdrawtriangle3d(-3.297,-14.167, 3.671,
                 -3.297,-14.167, 3.894,
                 0.003,-14.167, 3.894,
                 -1.997,-14.467, 3.874)

// Wing root strut forward left
setcolour(maprgb(80,80,80))
cdrawquad3d( 0.433, 0.950, 2.900,
           0.633, 0.950, 2.900,
           0.633, 1.000, 0,
           0.433, 1.000, 0)

// Wing root strut rear left
setcolour(maprgb(80,80,80))
cdrawquad3d( -1.967, 0.950, 2.616,
           -1.767, 0.950, 2.616,
           -0.868, 1.000, 0,
           -1.068, 1.000, 0)
```
// Wing root strut diag left
setcolour(maprgb(80,80,80))
cdrawquad3d( 0.433, 0.950, 2.900,
  0.633, 0.950, 2.900,
 -0.868, 1.000, 0,
 -1.068, 1.000, 0)

// Wing root strut forward right
setcolour(maprgb(80,80,80))
cdrawquad3d( 0.433, -0.950, 2.900,
  0.633, -0.950, 2.900,
  0.633, -1.000, 0,
  0.433, -1.000, 0)

// Wing root strut rear right
setcolour(maprgb(80,80,80))
cdrawquad3d( -1.967, -0.950, 2.616,
 -1.767, -0.950, 2.616,
 -0.868, -1.000, 0,
 -1.068, -1.000, 0)

// Wing root strut diag right
setcolour(maprgb(80,80,80))
cdrawquad3d( 0.433, -0.950, 2.900,
  0.633, -0.950, 2.900,
 -0.868, -1.000, 0,
 -1.068, -1.000, 0)

// Wing strut forward left
setcolour(maprgb(80,80,80))
cdrawquad3d( -2.200, 10.000, -1.120,
 -2.450, 10.000, -1.120,
 -0.550, 10.000, 3.315,
 -0.300, 10.000, 3.315)

// Wing strut rear left
setcolour(maprgb(80,80,80))
cdrawquad3d( -4.500, 10.000, -1.260,
 -4.750, 10.000, -1.260,
 -2.850, 10.000, 3.210,
 -2.500, 10.000, 3.210)

// Wing strut forward right
setcolour(maprgb(80,80,80))
cdrawquad3d( -2.200, -10.000, -1.120,
CHAPTER 5. INTERACTIVE GRAPHICS IN BCPL USING SDL

-2.450, -10.000, -1.120,
-0.550, -10.000, 3.315,
-0.300, -10.000, 3.315)

// Wing strut rear right
setcolour(maprgb(80,80,80))
cdrawquad3d(-4.500, -10.000, -1.260,
-4.750, -10.000, -1.260,
-2.850, -10.000, 3.210,
-2.500, -10.000, 3.210)

// Wheel strut left
setcolour(maprgb(80,80,80))
cdrawquad3d(-0.768, 1.000, -2.000,
-1.168, 1.000, -2.000,
-0.468, 2.000, -3.800,
-0.068, 2.000, -3.800)

// Wheel strut diag left
setcolour(maprgb(80,80,80))
cdrawquad3d(1.600, 1.000, -2.000,
1.800, 1.000, -2.000,
-0.368, 2.000, -3.800,
-0.168, 2.000, -3.800)

// Wheel strut centre left
setcolour(maprgb(80,80,80))
cdrawquad3d(-0.500, 0.000, -2.900,
-0.650, 0.000, -2.900,
-0.318, 2.000, -3.800,
-0.168, 2.000, -3.800)

// Wheel strut right
setcolour(maprgb(80,80,80))
cdrawquad3d(-0.768, -1.000, -2.000,
-1.168, -1.000, -2.000,
-0.468, -2.000, -3.800,
-0.068, -2.000, -3.800)

// Wheel strut diag right
setcolour(maprgb(80,80,80))
cdrawquad3d(1.600, -1.000, -2.000,
1.800, -1.000, -2.000,
-0.368, -2.000, -3.800,
-0.168, -2.000, -3.800)
// Wheel strut centre right
setcolour(maprgb(80,80,80))
cdrawquad3d( -0_500, -0_000, -2_900,
            -0_650, -0_000, -2_900,
            -0_318, -2_000, -3_800,
            -0_168, -2_000, -3_800)

// Left wheel
setcolour(maprgb(20,20,20))
cdrawquad3d( -0_268, 2_100, -3_800,
            -0_268, 2_100, -3_800-0_700,
            -0_268-0_500, 2_100, -3_800-0_500,
            -0_268-0_700, 2_100, -3_800)
cdrawquad3d( -0_268, 2_100, -3_800,
            -0_268, 2_100, -3_800-0_700,
            -0_268+0_500, 2_100, -3_800+0_500,
            -0_268+0_700, 2_100, -3_800)
cdrawquad3d( -0_268, 2_100, -3_800,
            -0_268, 2_100, -3_800-0_700,
            -0_268-0_500, 2_100, -3_800-0_500,
            -0_268-0_700, 2_100, -3_800)
cdrawquad3d( -0_268, 2_100, -3_800,
            -0_268, 2_100, -3_800-0_700,
            -0_268-0_500, 2_100, -3_800-0_500,
            -0_268-0_700, 2_100, -3_800)

// Right wheel
setcolour(maprgb(20,20,20))
cdrawquad3d( -0_268, -2_100, -3_800,
            -0_268, -2_100, -3_800-0_700,
            -0_268-0_500,-2_100, -3_800-0_500,
            -0_268-0_700,-2_100, -3_800)
cdrawquad3d( -0_268, -2_100, -3_800,
            -0_268, -2_100, -3_800-0_700,
            -0_268+0_500,-2_100, -3_800+0_500,
            -0_268+0_700,-2_100, -3_800)
cdrawquad3d( -0_268, -2_100, -3_800,
            -0_268, -2_100, -3_800-0_700,
            -0_268-0_500,-2_100, -3_800-0_500,
            -0_268-0_700,-2_100, -3_800)
cdrawquad3d( -0_268, -2_100, -3_800,
            -0_268, -2_100, -3_800-0_700,
            -0_268-0_500,-2_100, -3_800-0_500,
// Fueltank front
setcolour(maprgb(200,200,230)) // Top surface
cdrawquad3d( 1_333, 1_000, 2_900,
1_333, -1_000, 2_900,
0_033, -1_000, 3_100,
0_033, 1_000, 3_100)

// Fueltank back
setcolour(maprgb(180,180,210)) // Top surface
cdrawquad3d( 0_033, 1_000, 3_100,
0_033, -1_000, 3_100,
-1_967, -1_000, 2_616,
-1_967, 1_000, 2_616)

// Fueltank left side
setcolour(maprgb(160,160,190))
cdrawtriangle3d( 1_333, 1_000, 2_900,
0_033, 1_000, 3_100,
-1_967, 1_000, 2_616)

// Fueltank right side
setcolour(maprgb(160,160,190))
cdrawtriangle3d(-0_500+1_833, -1_000, -2_000+4_900,
-1_800+1_833, -1_000, -1_800+4_900,
-3_800+1_833, -1_000, -2_284+4_900)

// Fuselage
// Prop shaft
setcolour(maprgb(40,40,90))
cdrawtriangle3d( 5_500, 0, 0,
4_700, 0_200, 0_300,
4_700, 0_200,-0_300)
setcolour(maprgb(60,60,40))
cdrawtriangle3d( 5_500, 0, 0,
4_700, 0_200,-0_300,
4_700,-0_200,-0_300)
setcolour(maprgb(40,40,90))
cdrawtriangle3d( 5_500, 0, 0,
4_700,-0_200,-0_300,
4_700,-0_200, 0_300)
setcolour(maprgb(60,60,40))
5.18. DRAWTIGERMOTH.B

```c

cdrawtriangle3d( 5, 500, 0, 0,
                     4, 700, -0, 200, 0, 300,
                     4, 700, 0, 200, 0, 300)

// Engine front lower centre
setcolour(maprgb(140, 140, 160))
cdrawtriangle3d( 5, 000, 0, 0,
                     4, 500, 0, 550, -1, 750,
                     4, 500, -0, 550, -1, 750)

// Engine front lower left
setcolour(maprgb(140, 120, 130))
cdrawtriangle3d( 5, 000, 0, 0,
                     4, 500, 0, 550, -1, 750,
                     4, 500, 0, 550, 0)

// Engine front lower right
setcolour(maprgb(140, 120, 130))
cdrawtriangle3d( 5, 000, 0, 0,
                     4, 500, -0, 550, -1, 750,
                     4, 500, -0, 550, 0)

// Engine front upper centre
setcolour(maprgb(140, 140, 160))
cdrawtriangle3d( 5, 000, 0, 0,
                     4, 500, 0, 550, 0, 500,
                     4, 500, -0, 550, 0, 500)

// Engine front upper left
setcolour(maprgb(100, 140, 180))
cdrawtriangle3d( 5, 000, 0, 0,
                     4, 500, 0, 550, 0, 500,
                     4, 500, 0, 550, 0)
cdrawtriangle3d( 5, 000, 0, 0,
                     4, 500, -0, 550, 0, 500,
                     4, 500, -0, 550, 0)

// Engine left lower
setcolour(maprgb(80, 80, 60))
cdrawquad3d( 1, 033, 1, 000, 0,
                  1, 800, 1, 000, -2, 000,
                  4, 500, 0, 550, -1, 750,
                  4, 500, 0, 550, 0)
```
// Engine right lower
setcolour(maprgb(80,100,60))
cdrawquad3d( 1_033,-1_000, 0,
 1_800,-1_000, -2_000,
 4_500,-0_550, -1_750,
 4_500,-0_550, 0)

// Engine top left
setcolour(maprgb(100,130,60))
cdrawquad3d( 1_033, 0_900, 0_950,
 1_033, 0_900, 0_000,
 4_500, 0_550, 0_000,
 4_500, 0_550, 0_500)

// Engine top centre
setcolour(maprgb(130,160,90))
cdrawquad3d( 1_033, 0_900, 0_950,
 1_033,-0_900, 0_950,
 4_500,-0_550, 0_500,
 4_500, 0_550, 0_500)

// Engine top right
setcolour(maprgb(100,130,60))
cdrawquad3d( 1_033,-0_900, 0_950,
 1_033,-0_900, 0_000,
 4_500,-0_550, 0_000,
 4_500,-0_550, 0_500)

// Engine bottom
setcolour(maprgb(100,80,50))
cdrawquad3d( 4_500, 0_550, -1_750,
 4_500,-0_550, -1_750,
 1_800,-1_000, -2_000,
 1_800, 1_000, -2_000)

// Front cockpit left
setcolour(maprgb(120,140,60))
cdrawquad3d( -2_000, 1_000, 0_000,
 -2_000, 0_870, 0_600,
 -3_300, 0_870, 0_600,
 -3_300, 1_000, 0_000)

// Front cockpit right
setcolour(maprgb(120,140,60))
cdrawquad3d( -2_000,-1_000, 0_000,
-2_000,-0_870, 0_600,
-3_300,-0_870, 0_600,
-3_300,-1_000, 0_000)

// Top front left
setcolour(maprgb(100,120,40))
cdrawquad3d( 1_033, 0_900, 0_950,
-2_000, 0_750, 1_000,
-2_000, 0_750, 0_000,
1_033, 0_900, 0_000)

// Top front middle
setcolour(maprgb(120,140,60))
cdrawquad3d( 1_033, 0_900, 0_950,
1_033,-0_900, 0_950,
-2_000,-0_750, 1_000,
-2_000, 0_750, 1_000)

// Top front right
setcolour(maprgb(100,120,40))
cdrawquad3d( 1_033,-0_900, 0_950,
-2_000,-0_750, 1_000,
-2_000,-0_750, 0_000,
1_033,-0_900, 0_000)

// Front wind shield
setcolour(maprgb(180,200,150))
cdrawquad3d( -1_300, 0_450, 1_000,
-2_000, 0_450, 1_400,
-2_000,-0_450, 1_400,
-1_300,-0_450, 1_000)
setcolour(maprgb(220,220,180))
cdrawtriangle3d( -1_300, 0_450, 1_000,
-2_000, 0_450, 1_400,
-2_000, 0_650, 1_000)

setcolour(maprgb(170,200,150))
cdrawtriangle3d( -1_300,-0_450, 1_000,
-2_000,-0_450, 1_400,
-2_000,-0_650, 1_000)

// Top left middle
CHAPTER 5. INTERACTIVE GRAPHICS IN BCPL USING SDL

setcolour(maprgb(130,160,90))
cdrawquad3d( -3_300, 0_750, 1_000,
            -3_300, 1_000, 0_000,
            -4_300, 1_000, 0_000,
            -4_300, 0_750, 1_000)

// Top centre middle
setcolour(maprgb(120,140,60))
cdrawquad3d( -3_300, 0_750, 1_000,
            -3_300,-0_750, 1_000,
            -4_300,-0_750, 1_000,
            -4_300, 0_750, 1_000)

// Top right middle
setcolour(maprgb(130,160,90))
cdrawquad3d( -3_300,-0_750, 1_000,
            -3_300,-1_000, 0_000,
            -4_300,-1_000, 0_000,
            -4_300,-0_750, 1_000)

// Rear cockpit left
setcolour(maprgb(120,140,60))
cdrawquad3d( -4_300, 1_000, 0_000,
            -4_300, 0_870, 0_600,
            -5_583, 0_870, 0_600,
            -5_583, 1_000, 0_000)

// Rear wind shield
setcolour(maprgb(180,200,150))
cdrawquad3d( -3_600, 0_450, 1_000,
            -3_600, 0_450, 1_400,
            -4_300, 0_450, 1_400,
            -3_600,-0_450, 1_000)
setcolour(maprgb(220,220,180))
cdrawtriangle3d( -3_600, 0_450, 1_000,
                -4_300, 0_450, 1_400,
                -4_300, 0_650, 1_000)

setcolour(maprgb(170,200,150))
cdrawtriangle3d( -3_600,-0_450, 1_000,
                -4_300,-0_450, 1_400,
                -4_300,-0_650, 1_000)
// Rear cockpit right
setcolour(maprgb(110,140,70))
cdrawquad3d( -4_300,-1_000, 0_000,
         -4_300,-0_870, 0_600,
         -5_583,-0_870, 0_600,
         -5_583,-1_000, 0_000)

// Lower left middle
setcolour(maprgb(140,110,70))
cdrawquad3d( 1_033, 1_000, 0,
         1_800, 1_000, -2_000,
         -3_583, 1_000, -2_238,
         -3_583, 1_000, 0)

// Bottom middle
setcolour(maprgb(120,100,60))
cdrawquad3d( 1_800, 1_000, -2_000,
         -3_583, 1_000, -2_238,
         -3_583,-1_000, -2_238,
         1_800,-1_000, -2_000)

// Lower right middle
setcolour(maprgb(140,110,70))
cdrawquad3d( 1_033,-1_000, 0,
         1_800,-1_000, -2_000,
         -3_583,-1_000, -2_238,
         -3_583,-1_000, 0)

// Lower left back
setcolour(maprgb(160,120,80))
cdrawquad3d( -3_583, 1_000, 0,
         -16_000, 0_050, 0,
         -16_000, 0_050, -0_667,
         -3_583, 1_000, -2_238)

// Bottom back
setcolour(maprgb(130,90,60))
cdrawquad3d( -3_583, 1_000, -2_238,
         -16_000, 0_050, -0_667,
         -16_000,-0_050, -0_667,
         -3_583,-1_000, -2_238)

// Lower right back
setcolour(maprgb(160,140,80))
cdrawquad3d( -3_583,-1_000, 0,
-16_000,-0_050, 0,
-16_000,-0_050, -0_667,
-3_583,-1_000, -2_238)

// Top left back
setcolour(maprgb(130,130,80))
cdrawtriangle3d( -5_583, 0_650, 0_950,
-5_583, 1_000, 0_000,
-13_900, 0_150, 0)

// Top centre back
setcolour(maprgb(130,160,90))
cdrawquad3d( -5_583, 0_650, 0_950,
-5_583,-0_650, 0_950,
-13_900,-0_150, 0,
-13_900, 0_150, 0)

// Top right back
setcolour(maprgb(130,130,80))
cdrawtriangle3d( -5_583,-0_650, 0_950,
-5_583,-1_000, 0_000,
-13_900,-0_150, 0)

// Fin
{ // Rudder deflection 1 inch from hinge
  LET a = muldiv(1_100, c_rudder, 32_768*17)

  setcolour(maprgb(170,180,80))
cdrawquad3d(-14_000, 0_000, 0, // Fin
-16_000, 0_000, 0,
-16_000, 0_000, 1_000,
-15_200, 0_000, 1_000)

  setcolour(maprgb(70,120,40))
cdrawquad3d(-15_200-3*a, 9*a, 1_000, // Rudder
-16_000, 0, 1_000,
-16_800+3*a,-10*a, 3_100,
-16_000, 0, 2_550)
setcolour(maprgb(70, 80,40))
cdrawquad3d(-16_000, 0, 1_000,
-16_800+3*a,-10*a, 3_100,
-17_566+4*a,-14*a, 2_600,
setcolour(maprgb(70,120,40))
cdrawquad3d(-16_000, 0, 1_000, -17_816+4*a,-17*a, 1_667)
setcolour(maprgb(70,80,40))
cdrawquad3d(-16_000, 0, 1_000, -17_566+4*a,-14*a, 0)
setcolour(maprgb(20,20,20))
cdrawquad3d(-16_000, 0, -0_667, -16_200, 0, -0_667)
setcolour(maprgb(160,200,50))
cdrawquad3d(-16_000, 0_000, // Left tailplane
           -13_900, 0_600, 0,
           -14_600, 2_800, 0,
           -16_000, 4_500, 0)
setcolour(maprgb(120,200,50))
cdrawtriangle3d(-13_900, 0_600, 0,
                -13_900,-0_600, 0,
                -16_000, 0_000, 0)
cdrawquad3d(-16_000, 0_000, // Right tailplane
           -13_900,-0_600, 0,
           -14_600,-2_800, 0,
           -16_000,-4_500, 0)
setcolour(maprgb(170,150,80))
cdrawquad3d(-16_000, 0_000, // Left elevator
           -17_200+4*a, 0_600, -15*a, // pt 1
           -17_500+5*a, 0_900, -16*a, // pt 2
           -17_816+4*a,-17*a, 1_667)
setcolour(maprgb(120,170,60))
cdrawquad3d(-16_000, 0_000, 0, // Left elevator
          -17.666+5*a, 2_000, -17*a, // pt 3
          -17.450+4*a, 3_500, -16*a, // pt 4
          -17.200+4*a, 4_650, -14*a) // pt 5

setcolour(maprgb(160,120,40))
cdrawquad3d(-16_000, 0_000, 0, // Left elevator
          -17.200+4*a, 4_650, -14*a, // pt 5
          -16.700+a/2, 4_833, -2*a, // pt 6
          -16_000, 4_500, a) // pt 7

setcolour(maprgb(170,150,80))
cdrawquad3d(-16_000, 0_000, 0, // Right elevator
          -17.200+4*a, -0_600, -15*a, // pt 1
          -17.500+5*a, -0_900, -16*a, // pt 2
          -17.666+5*a, -2_000, -17*a) // pt 3

setcolour(maprgb(120,170,60))
cdrawquad3d(-16_000, 0_000, 0, // Right elevator
          -17.666+5*a, -2_000, -17*a, // pt 3
          -17.450+4*a, -3_500, -16*a, // pt 4
          -17.200+4*a, -4_650, -14*a) // pt 5

setcolour(maprgb(160,120,40))
cdrawquad3d(-16_000, 0_000, 0, // Right elevator
          -17.200+4*a, -4_650, -14*a, // pt 5
          -16.700+a/2, -4_833, -2*a, // pt 6
          -16_000, -4_500, a) // pt 7

5.19 Tigermoth Flight Simulator

This section describes a flight simulator for a De Havilland Tigermoth biplane. A typical image of the flight simulator in use is as follows.
Notice that the USB joystick used has more features than the one shown on page ???. This one is a Cyborg X joystick. It can control the aileron, elevator and rudder. It has two throttle levers which can be locked together. There is an eight direction hat which can be used to change the direction of view of either the pilot or an observer, and there are 12 buttons. It typically costs about £32.

More to follow.

`/*

########### THIS IS UNDER DEVELOPMENT ##################################################

This is a flight simulator based on Jumbo that ran interactively on a PDP 11 generating the pilots view on a Vector General Display.

Originally implemented by Martin Richards in mid 1970s.

Substantially modified my Martin Richards (c) October 2012.

It has been extended to use 32 rather than 16 bit arithmetic.

It is planned that this will simulate the flying characteristics of a De Havilland D.H.82A Tiger Moth which I learnt to fly as a teenager.

Change history

25/01/2013`
Name changed to tiger.b

Controls

Either use a USB Joystick for elevator, ailerons and throttle, or use the keyboard as follows:

Up arrow       Trim joystick forward a bit
Down arrow     Trim joystick backward a bit
Left arrow     Trim joystick left a bit
Right arrow    Trim joystick right a bit

, or <       Trim rudder left
. or >       Trim rudder right
x            Trim more thrust
z            Trim less thrust

0            Display the pilot's view
1,2,3,4,5,6,7,8 Display the aircraft viewed from various angles

f            View aircraft from a greater distance
n            View aircraft from a closer position

p            pause/unpause the simulation

Reset the aircraft on the glide path

t            Reset the aircraft ready for take off -- default
tie stationary on the ground at the end of the runway

b            brake on/off -- not available
u            undercarriage up/down -- not available

q            Quit

There are joystick buttons equivalent to Up arrow, Down arrow, Left Arrow and Right arrow. There are also joystick buttons to trim the rudder left and right, useful for steering on the runway. There are also joystick buttons to toggle gear up/down and brakes on/off.

The display shows various beacons on the ground including the lights on the sides and the ends of the runway.

The display also shows various flight instruments including the artificial horizon, the height and speed and various navigational aids.
5.19. TIGERMOTh FLIGHT SIMULATOR

To help the pilot find the runway.

*/

GET "libhdr"
GET "sdl.h"
GET "sdl.b"
.
GET "libhdr"
GET "sdl.h"

MANIFEST {
One = 1_000000 // Direction cosines scaling factor
   // ie 6 decimal digits after the decimal point.
D45 = 0_707107 // cosine of pi/4
Sps = 10  // Steps per second

// Most measurements are in feet scaled with 3 digits after the decimal point
k_g = 32_000  // Acceleration due to gravity, 32 ft per sec per sec
   // Scaled with 3 digits after the decimal point.
k_drag = k_g/15 // Acceleration due to drag as 100 ft per sec
   // The drag is proportional to the square of the speed.

// Conversion factors
mph2fps  = 5280_000/(60*60)
mph2knots = 128_000/147
}

GLOBAL {
aircraft:ug // Select which aircraft to simulate

stepping   // =FALSE if not stepping the simulation
crashed    // =TRUE if crashed
debugging
testing    // Toggle testing mode
plotusage
done

col_black
col_blue
col_green
col_yellow
col_red
col_majenta
col_cyan
col_white
col_darkgray
col_darkblue
col_darkgreen
col_darkyellow
col_darkred
col_darkmajenta
col_darkcyan
col_gray
col_lightgray
col_lightblue
col_lightgreen
col_lightyellow
col_lightred
col_lightmajenta
col_lightcyan

c_thrust; c_trimthrust
c_aileron; c_trimaileron
c_elevator; c_trimelevator
c_rudder; c_trimmrudder

c_geardown // TRUE or FALSE
c_brakeson // TRUE or FALSE

cctx; cty; ctz // Direction cosines of direction t
cwx; cwy; cwz // Direction cosines of direction w
clx; cly; clz // Direction cosines of direction l

cetx; cety; cetz // Eye direction cosines of direction t
cewx; cewy; cewz // Eye direction cosines of direction w
celx; cely; celz // Eye direction cosines of direction l

cockpitz // Height of the pilots eye

cgx; cgy; cgz // Coordinates of the CG of the aircraft
  // in feet with 3 digits after the decimal point
  // eg cgz=1000.000 represents a height of 1000 ft

cgxdot; cgydot; cgzdot // These are set by step()

eyex; eyey; eyez // Relative position of the eye
eyedist // Eye x or y distance from aircraft

hatdir // Hat direction
5.19. TIGERMOTH FLIGHT SIMULATOR

hatmsecs // msecs of last hat change
eyedir // Eye direction
   // 0 = cockpit view
   // 1,...,8 view from behind, behind-left, etc
cdrawtriangle3d
cdrawquad3d

// Speed in various directions is measured in ft/s scaled
// with 3 digits after the decimal point
// eg 146_666 represents 146.666 ft/s = 100 mph
tdot; wdot; ldot // Speed in t, w and l directions
tdotsq; wdotsq; ldotsq // Speed squared in t, w and l directions

mass // Mass of the aircraft

mit; miw; mil // Moment of inertia about t, w and l axes

rtdot; rwdot; rldot // Rotation rates about t, w and l axes
rdt; rdw; rdl // Rotational damping about t, w and l axes

//Linear forces are scaled with 3 digits after the decimal point
// as are direction cosines.
ft; ft1 // Force and previous force in t direction
fw; fw1 // Force and previous force in w direction
fl; fl1 // Force and previous force in l direction

// Rotational forces are scaled with 6 digits after the decimal point
// as are direction cosines.
rft; rft1 // Current and previous moment about t axis
rfw; rfw1 // Current and previous moment about w axis
rfl; rfl1 // Current and previous moment about l axis

atl; atw; awl // Angle of air flow in planes tl, tw and wl

// Table interpolated by rdtab(angle, tab)
rtltab; rtwtab; rwltab // Rotational tables
tltab; twtab; wltab // Linear tables

usage // 0 to 100 percentage cpu usage
}

// Insert the definition of drawtigermoth()
GET "drawtigermoth.b"

LET inprod(a,b,c, x,y,z) =
// Return the cosine of the angle between two unit vectors.
muldiv(a, x, One) + muldiv(b, y, One) + muldiv(c, z, One)

AND rotate(t, w, l) BE
{ // Rotate the orientation of the aircraft
  // t, w and l are assumed to be small and cause
  // rotation about axis t, w, l. Positive values cause
  // anti-clockwise rotations about their axes.
  LET tx = inprod(One, -l, w, ctx, cwx, clx)
  LET wx = inprod( One, -t, ctx, cwx, clx)
  LET lx = inprod( -w, t, One, ctx, cwx, clx)

  LET ty = inprod(One, -l, w, cty, cwz, clz)
  LET wy = inprod( One, -t, cty, cwz, clz)
  LET ly = inprod( -w, t, One, cty, cwz, clz)

  LET tz = inprod(One, -l, w, ctz, cwz, clz)
  LET wz = inprod( One, -t, ctz, cwz, clz)
  LET lz = inprod( -w, t, One, ctz, cwz, clz)

  ctx, cty, ctz := tx, ty, tz
  cwx, cwz, clz := wx, wy, wz
  clx, cly, clz := lx, ly, lz

  adjustlength(@ctx); adjustlength(@cwx); adjustlength(@clx)
  adjustortho(@ctx, @cwx); adjustortho(@ctx, @clx); adjustortho(@cwx, @clx)
}

AND adjustlength(v) BE
{ // This helps to keep vector v of unit length
  LET x, y, z = v!0, v!1, v!2
  LET corr = One + (inprod(x,y,z, x,y,z) - One)/2
  v!0 := muldiv(x, One, corr)
  v!1 := muldiv(y, One, corr)
  v!2 := muldiv(z, One, corr)
}

AND adjustortho(a, b) BE
{ // This helps to keep the unit vector b orthogonal to a
  LET a0, a1, a2 = a!0, a!1, a!2
  LET b0, b1, b2 = b!0, b!1, b!2
  LET corr = inprod(a0,a1,a2, b0,b1,b2)
  b!0 := b0 - muldiv(a0, corr, One)
  b!1 := b1 - muldiv(a1, corr, One)
\text{AND rdtab(a, tab) = VALOF}

\{ // Perform linear interpolation between appropriate entries
  // in the given table. The first and last entries must be for
  // angles -180.000 and +180.000, respectively.
  // The angle a is scaled with three digits after the decimal point.
  LET p = tab
  LET a0, r0, a1, r1 = ?, ?, ?, ?
  IF a<-180_000 DO a := -180_000
  IF a>+180_000 DO a := +180_000
  WHILE a>p DO p := p+2
  IF a=p RESULTIS p
  a0, r0 := p-2, p-1
  a1, r1 := p, p+1
  RESULTIS r0 + muldiv(r1-r0, a-a0, a1-a0)
\}

\text{AND angle(x, y) = x=0 & y=0 \rightarrow 0, VALOF}

\{ // Calculate an approximation to the angle in degrees between
  // point (x,y) and the x axis. The result is a scaled number with
  // three digits after the decimal point.
  // Points above the x axis have positive angles and
  // points below the x axis have negative angles.
  LET px, py = ABS x, ABS y
  LET t = muldiv(90_000, y, px+py)
  IF x>=0 RESULTIS t
  IF y>=0 RESULTIS 180_000 - t
  RESULTIS -(180_000 + t)
\}

\text{LET step() BE}

\{ // Update the aircraft position, orientation and motion.

  // Calculate the linear and rotational forces on the aircraft
  // In directions t, w and l
  ft, fw, fl := 0, 0, 0 // Initialise all to zero
  rft, rfw, rfl := 0, 0, 0

  // Air flow angles
  atl := angle(tdot, ldot)
  atw := angle(tdot, wdot)
  awl := angle(wdot, ldot)
// Calculate speed squared in the three direction
// scaled so that 100 ft/s squared gives 1.000 scaled
// with 3 digits after the decimal point.
tdotsq := muldiv(tdot, tdot, 10_000_000)
wdotsq := muldiv(wdot, wdot, 10_000_000)
ldotsq := muldiv(ldot, ldot, 10_000_000)

//writef("tdot=%8.3d ldot=%8.3d atl=%7.3d*n", tdot, ldot, atl)
//writef("tdot=%8.3d wdot=%8.3d atw=%7.3d*n", tdot, wdot, atw)
//writef("wdot=%8.3d ldot=%8.3d awl=%7.3d*n", wdot, ldot, awl)
//writef("tdotsq=%8.3d wdotsq=%8.3d ldotsq=%8.3d*n", tdotsq, wdotsq, ldotsq)

// Rotational damping
// rtdot, rwdot and rldot are in radians per second.
rtdot := muldiv(rtdot, rdt, 1_000*Sps)
rdot := muldiv(rwdot, rdw, 1_000*Sps)
rdot := muldiv(rldot, rdl, 1_000*Sps)

// Rotational aerodynamic forces on fixed surfaces

// Dihedral effect
rft := rft + muldiv(-10, wdotsq, 100)

// Stabiliser effect
rfw := rfw + muldiv(-10, ldot, 100)

// Fin effect
rfl := rfl + muldiv(-10, wdotsq, 100)

// Aileron effect
rft := rft + muldiv(-c_aileron, tdot, 200)

// Elevator effect
rfw := rfw - muldiv(c_elevator, tdot+c_thrust, 100)

// Rudder effect
rfl := rfl + muldiv(c_rudder, tdot+c_thrust, 100)

//writef("rft=%9.6d rft1=%9.6d*n", rft, rft1)
//writef("rfw=%9.6d rfw1=%9.6d*n", rfw, rft1)
//writef("rfl=%9.6d rfl1=%9.6d*n", rfl, rft1)

UNLESS testing DO
{ // Do not apply rotations in testing mode
// Apply rotational effects using the trapizoidal rule
// for integration.
rtdot := rtdot + (rft+rft1)/2/Sps
rwdot := rwdot + (rfw+rfw1)/2/Sps
rldot := rldot + (rfl+rfl1)/2/Sps
}
rft1, rfw1, rfl1 := rft, rfw, rfl // Save previous values

// Linear forces

// Gravity effect
ft := ft + muldiv(-k_g, ctz, One) // Gravity in direction t
fw := fw + muldiv(-k_g, cwz, One) // Gravity in direction w
fl := fl + muldiv(-k_g, clz, One) // Gravity in direction l

// Drag effect
ft := ft - muldiv(-k_drag, tdot, 1000000)

// Side effect
fw := fw - muldiv(wdot, 100, 1000)

// Lift effect
{ // Lift is proportions to speed squared (= tdot**2 + ldot**2)
// multiplied by rdtab(angle, tltab)
// When angle=0 and speed=100 ft/sec lift is k_g
// angle(0, tltab) = 267
// so lift = k_g * (rdtab(angle, tltab)/267) * (speed*speed/(100*100)
LET tab = TABLE -180_000, 0,
    -90_000, 500,
    -15_000, 200,
    -11_000, 1000,
        0, 267, // Lift factor when ldot=0
    4_000, 0,
    19_000, -600,
    24_000, -100,
    90_000, -500,
    180_000, 0
LET a = muldiv(k_g, rdtab(atl, tab), 267)
fl := fl + muldiv(a, tdotsq+ldotsq, 1000)
}

// Thrust effect
ft := ft + muldiv(c_thrust, k_g/8, 2*32768)
//writef("ft=%9.3d fw=%9.3d fl=%9.3d*n", ft, fw, fl)

UNLESS testing DO
{ // Do not apply the forces in testing mode

    // Apply linear effects using the trapizoidal rule
    // for integration.
    tdot := tdot + (ft+ft1)/2/Sps
    wdot := wdot + (fw/fw1)/2/Sps
    ldot := ldot + (fl+fl1)/2/Sps

    ft1, fw1, fl1 := ft, fw, fl // Save the previous values

    // Calculate x, y and z speeds
    cgxdot := inprod(ctx,cwx,clx, tdot,wdot,ldot)
    cgydot := inprod(cty,cwy,cly, tdot,wdot,ldot)
    cgzdot := inprod(ctz,cwz,clz, tdot,wdot,ldot)

    // Calculate new x, y and z positions.
    cgx := cgx + cgxdot/Sps
    cgy := cgy + cgydot/Sps
    cgz := cgz + cgzdot/Sps

    rotate(rtdot/Sps, rwdot/Sps, rldot/Sps)

    // Compute the new values of tdot, wdot and ldot
    // from cgxdot, cgydot and cgzdot using the new orientation
    tdot := inprod(cgxdot,cgydot,cgzdot, ctx,cty,ctz)
    wdot := inprod(cgxdot,cgydot,cgzdot, cwx,cwy,cwz)
    ldot := inprod(cgxdot,cgydot,cgzdot, clx,cly,clz)

    //writef("cgx=%9.3d cgy=%9.3d cgz=%9.3d*n", cgx, cgy, cgz)
    //abort(1003)
}

IF cgz < 10_000 DO
{ // The aircraft is near the ground

    IF cgz < 2_000 | clz<0_800000 DO
    { crashed := TRUE
     stepping := FALSE
     RETURN
    }
}
5.19. TIGERMOTH FLIGHT SIMULATOR

AND plotcraft() BE
{ IF depthscreen FOR i = 0 TO screenxsize*screenysize-1 DO
  depthscreen!i := maxint

  //setyeyposition()

  IF aircraft=0 DO
  { // Simple aircraft
    setcolour(maprgb(64,128,64)) // Fuselage
    cdrawtriangle3d(6_000,0,0, 2_000,0,-1_000, -2_000,0,2_000)
    setcolour(maprgb(40,100,40))
    cdrawtriangle3d(2_000,0,-1_000, -2_000,0,2_000, -12_000,0,0)
    setcolour(maprgb(255,255,255))
    cdrawtriangle3d(2_000,0, 1_000, -2_000,0,2_000, 0_800,0,2_000)
    setcolour(maprgb(255,0,0)) // Port wing -- Red
    cdrawtriangle3d(2_500,0,0, -2_500,0,0, -2_000, 18_000,2_000)
    setcolour(maprgb(0,255,0)) // Starboard wing -- Green
    cdrawtriangle3d(2_500,0,0, -2_500,0,0, -2_000,-18_000,2_000)
    setcolour(maprgb(255,0,255)) // Stabliser
    cdrawtriangle3d(-9_000,0,0, -12_000,0,0, -13_000,-4_000,0)
    setcolour(maprgb(255,255,0))
    cdrawtriangle3d(-9_000,0,0, -12_000,0,0, -13_000, 4_000,0)
    setcolour(maprgb(0,255,255)) // Fin
    cdrawtriangle3d(-9_000,0,0, -12_000,0,0, -13_000,0,4_000)
  }

  IF aircraft=1 DO
  { // Draw a Tigermoth
    //writef("Calling drawtigermoth\n")
    drawtigermoth()
    //writef("Returned from drawtigermoth\n")
  }

  IF aircraft=2 DO
  { LET s = 10_000
    LET r = 3_000
    // top
    setcolour(maprgb(0,0,0))
cdrawquad3d(r, 0, s, 0, r, s, -r, 0, s, 0, -r, s)

// top wings
setcolour(maprgb(255, 0, 0))
cdrawtriangle3d(r, 0, s, 0, s, 0, 0, r, s) // N
setcolour(maprgb(0, 255, 0))
cdrawtriangle3d(0, r, s, 0, s, s, 0, r, s) // W
setcolour(maprgb(255, 0, 0))
cdrawtriangle3d(-r, 0, s, -s, 0, s, -s, 0, r) // S
setcolour(maprgb(0, 255, 0))
cdrawtriangle3d(0, -r, s, 0, -s, s, 0, -s, r) // E

// Sides
setcolour(maprgb(128, 0, 0))
cdrawquad3d(s, 0, r, s, r, 0, s, 0, -r, s, -r, 0) // N
setcolour(maprgb(255, 128, 0))
cdrawquad3d(0, s, r, r, s, 0, 0, s, -r, -r, s, 0) // W
setcolour(maprgb(255, 0, 128))
cdrawquad3d(-s, 0, r, -s, r, 0, -s, 0, -r, -s, 0, r) // S
setcolour(maprgb(255, 128, 128))
cdrawquad3d(0, -s, r, r, -s, 0, 0, -s, -r, -r, s, 0) // W

// Centre wings
setcolour(maprgb(255, 128, 0))
cdrawtriangle3d(s, s, 0, r, s, 0, s, r, 0) // NW
setcolour(maprgb(0, 255, 128))
cdrawtriangle3d(-s, s, 0, -s, r, 0, -r, s, 0) // SW
setcolour(maprgb(128, 0, 255))
cdrawtriangle3d(-s, -s, 0, -r, -s, 0, -s, -r, 0) // SE
setcolour(maprgb(127, 255, 255))
cdrawtriangle3d(s, -s, 0, s, -r, 0, r, -s, 0) // NE

// bottom wings
setcolour(maprgb(255, 0, 0))
cdrawtriangle3d(r, 0, -s, 0, -s, s, 0, -r, s) // N
setcolour(maprgb(0, 255, 0))
cdrawtriangle3d(0, r, -s, 0, -s, s, 0, -r, s) // W
setcolour(maprgb(255, 0, 255))
cdrawtriangle3d(-r, 0, -s, 0, -s, -s, 0, -r, -s) // S
setcolour(maprgb(0, 255, 255))
cdrawtriangle3d(0, -r, -s, 0, -s, -s, 0, -r, -s) // E
5.19. TIGERMOTH FLIGHT SIMULATOR

// Bottom
setcolour(maprgb(128,128,128))
cdrawquad3d( r,0,-s, 0,r,-s, -r,0,-s, 0,-r,-s)
}
}

AND gdrawquad3d(x1,y1,z1, x2,y2,z2, x3,y3,z3, x4,y4,z4) BE
{ // Draw a 3D quad (not rotated)
  LET sx1,sy1,sz1 = ?,?,?,?
  LET sx2,sy2,sz2 = ?,?,?,?
  LET sx3,sy3,sz3 = ?,?,?,?
  LET sx4,sy4,sz4 = ?,?,?,?
  UNLESS screencoords(x1-eyex, y1-eyey, z1-eyez, @sx1) RETURN
  UNLESS screencoords(x2-eyex, y2-eyey, z2-eyez, @sx2) RETURN
  UNLESS screencoords(x3-eyex, y3-eyey, z3-eyez, @sx3) RETURN
  UNLESS screencoords(x4-eyex, y4-eyey, z4-eyez, @sx4) RETURN

  //drawquad3d(sx1,sy1,sz1, sx2,sy2,sz2, sx3,sy3,sz3, sx4,sy4,sz4)
}

AND cdrawquad3d(x1,y1,z1, x2,y2,z2, x3,y3,z3, x4,y4,z4) BE
{ LET rx1 = inprod(x1,y1,z1, ctx,cwx,clx)
  LET ry1 = inprod(x1,y1,z1, cty,cwy,cly)
  LET rz1 = inprod(x1,y1,z1, ctz,cwz,clz)

  LET rx2 = inprod(x2,y2,z2, ctx,cwx,clx)
  LET ry2 = inprod(x2,y2,z2, cty,cwy,cly)
  LET rz2 = inprod(x2,y2,z2, ctz,cwz,clz)

  LET rx3 = inprod(x3,y3,z3, ctx,cwx,clx)
  LET ry3 = inprod(x3,y3,z3, cty,cwy,cly)
  LET rz3 = inprod(x3,y3,z3, ctz,cwz,clz)

  LET rx4 = inprod(x4,y4,z4, ctx,cwx,clx)
  LET ry4 = inprod(x4,y4,z4, cty,cwy,cly)
  LET rz4 = inprod(x4,y4,z4, ctz,cwz,clz)

  LET sx1,sy1,sz1 = ?,?,?,?
  LET sx2,sy2,sz2 = ?,?,?,?
  LET sx3,sy3,sz3 = ?,?,?,?
  LET sx4,sy4,sz4 = ?,?,?,?
  //writef("cdrawquad3d called\n")
  UNLESS screencoords(rx1-eyex, ry1-eyey, rz1-eyez, @sx1) RETURN
  UNLESS screencoords(rx2-eyex, ry2-eyey, rz2-eyez, @sx2) RETURN
UNLESS screencoords(rx3-eyex, ry3-eyey, rz3-eyez, @sx3) RETURN
UNLESS screencoords(rx4-eyex, ry4-eyey, rz4-eyez, @sx4) RETURN
//writef("calling drawquad3d*n")
//writef("sx1=%i5 sy1=%i5 sz1=%i5*n", sx1,sy1,sz1)
//writef("sx2=%i5 sy2=%i5 sz2=%i5*n", sx2,sy2,sz2)
//writef("sx3=%i5 sy3=%i5 sz3=%i5*n", sx3,sy3,sz3)
//writef("sx4=%i5 sy4=%i5 sz4=%i5*n", sx4,sy4,sz4)
drawquad3d(sx1,sy1,sz1, sx2,sy2,sz2, sx3,sy3,sz3, sx4,sy4,sz4)
//writef("returned from drawquad3d*n")
}

AND cdrawtriangle3d(x1,y1,z1, x2,y2,z2, x3,y3,z3) BE
{ LET rx1 = inprod(x1,y1,z1, ctx,cwx,clx)
  LET ry1 = inprod(x1,y1,z1, cty,cwy,cly)
  LET rz1 = inprod(x1,y1,z1, ctz,cwz,clz)

  LET rx2 = inprod(x2,y2,z2, ctx,cwx,clx)
  LET ry2 = inprod(x2,y2,z2, cty,cwy,cly)
  LET rz2 = inprod(x2,y2,z2, ctz,cwz,clz)

  LET rx3 = inprod(x3,y3,z3, ctx,cwx,clx)
  LET ry3 = inprod(x3,y3,z3, cty,cwy,cly)
  LET rz3 = inprod(x3,y3,z3, ctz,cwz,clz)

  LET sx1,sy1,sz1 = ?,?,?
  LET sx2,sy2,sz2 = ?,?,?
  LET sx3,sy3,sz3 = ?,?,?

  UNLESS screencoords(rx1-eyex, ry1-eyey, rz1-eyez, @sx1) RETURN
  UNLESS screencoords(rx2-eyex, ry2-eyey, rz2-eyez, @sx2) RETURN
  UNLESS screencoords(rx3-eyex, ry3-eyey, rz3-eyez, @sx3) RETURN

drawtriangle3d(sx1,sy1,sz1, sx2,sy2,sz2, sx3,sy3,sz3)
}

AND screencoords(x,y,z, v) = VALOF
{ // If the point (x,y,z) is in view, set v!0, v!1 and v!2 to
  // the screen coordinates and depth and return TRUE
  // otherwise return FALSE
  LET sx = inprod(x,y,z, cewx,cewy,cewz) // Horizontal
  LET sy = inprod(x,y,z, celx,cely,celz) // Vertical
  LET sz = inprod(x,y,z, cetx,cety,cetz) // Depth

  LET screensize = screenxsize>=screenysize -> screenxsize, screensize

  //writef("screencoords: x=%9.3d y=%9.3d z=%9.3d*n", x,y,z)
  //writef("cetx=%9.6d cety=%9.6d cetz=%9.6d*n", cetx,cety,cetz)
5.19. TIGERMOTH FLIGHT SIMULATOR

//writef("cewx=%9.6d  cewy=%9.6d  cewz=%9.6d\n", cewx,cewy,cewz)
//writef("celx=%9.6d  cely=%9.6d  celz=%9.6d\n", celx,cely,celz)
//writef("eyex=%9.3d  eyey=%9.3d  eyez=%9.3d\n", eyex,eyey,eyez)

// Test that the point is in view, ie at least 1.000ft in front
// and no more than about 27 degrees (inverse tan 1/2) from the
// direction of view.
IF sz<1_000 &
   muldiv(sz, sz, 2000) >= muldiv(sx, sx, 1000) + muldiv(sy, sy, 1000)
RESULTIS FALSE

// A point screensize pixels away from the centre of the screen is
// 45 degrees from the direction of view.
// Note that many pixels in this range are off the screen.
v!0 := -muldiv(sx, screensize, sz)/1 + screenxsize/2
v!1 := +muldiv(sy, screensize, sz)/1 + screenysize/2
v!2 := sz  // This distance into the screen in arbitrary units, used
   // for hidden surface removal.

//writef("in view position=(x=%i4  y=%i4  depth=%n)\n", v!0, v!1, sz)
//abort(1119)
RESULTIS TRUE
}

AND screencoords2(px, py, pz, v) = VALOF
{
   // If the point (px,py,pz) is in the pilot’s field of view
   // set v!0 and v!1 to the screen coordinates and return TRUE
   // otherwise return FALSE
   //writef("px=%9.3d  py=%9.3d  pz=%9.3d\n", px, py, pz)
   //writef("v_t!0=%9.6d  v_t!1=%9.6d  v_t!2=%9.6d\n", v_t!0, v_t!1, v_t!2)
   //writef("v_w!0=%9.6d  v_w!1=%9.6d  v_w!2=%9.6d\n", v_w!0, v_w!1, v_w!2)
   //writef("v_l!0=%9.6d  v_l!1=%9.6d  v_l!2=%9.6d\n", v_l!0, v_l!1, v_l!2)

   LET x = inprod(px,py,pz, cewx,cewy,cewz)
   LET y = inprod(px,py,pz, celx,cely,celz)
   LET z = inprod(px,py,pz, cetx,cety,cetz)
   //writef("x=%9.3d  y=%9.3d  z=%9.3d\n", x, y, z)
   // Test that the point is in front of the aircraft
   // and no more than 45 degrees from the direction of thrust.
   UNLESS z>20 &
      muldiv(z, z, 2000) > muldiv(x, x, 1000) + muldiv(y, y, 1000) DO
      //abort(1001)
      RESULTIS FALSE
   }
   v!0 := -muldiv(x, screenxsize, z) / 1 + screenxsize/2
\[ v!1 := +\text{muldiv}(y, \text{screenxsize}, z) / (1 + \text{screenysize}/2) \]

//\text{writef}("v!0=%d v!1=%d*n", v!0, v!1)

RESULT IS TRUE

AND \text{draw_artificial_horizon()} BE
{ LET lx, ly, lz = ?, ?, ?
  LET rx, ry, rz = ?, ?, ?
  LET x, y, z = ctx, cty, ctz
  setcolour(col_cyan)
  screencoords(cgxdot, cgydot, cgzdot, @lx)
  drawcircle(lx, ly, 5)
  IF screencoords(x-y/4, y+x/4, 0, @lx) &
      screencoords(x+y/4, y-x/4, 0, @rx) DO
    { moveto(lx, ly)
      drawto(rx, ry)
    }
}

AND \text{draw_ground_point}(x, y) BE
{ LET gx, gy, gz = ?, ?, ?
  //\text{newline()}
  //\text{writef}("draw_ground_point: x=%n y=%n*n", x, y)
  //\text{writef}("draw_ground_point: cgx=%n cgy=%n cgz=%n*n", cgx, cgy, cgz)
  IF screencoords(x-cgx, y-cgy, -cgz-cockpitz, @gx) DO
    { drawrect(gx, gy, gx+1, gy+1)
      //\text{updatescreen()}
    }
}

AND \text{drawgroundpoints()} BE
{
  FOR x = 0 TO 200_000 BY 20_000 DO
    { FOR y = -50_000 TO 45_000 BY 5_000 DO
      { LET r = ABS(3*x + 5*y) MOD 23
        setcolour(maprgb(30+r, 30+r, 30+r))
        gdrawquad3d(x, y, 0,
                    x+20_000, y, 0,
                    x+20_000, y+5_000, 0,
                    x, y+5_000, 0)
      }
    }
}
setcolour(col_white)
draw_ground_point(0, 0)
FOR x = 0 TO 3000_000 BY 100_000 DO
  draw_ground_point(x, -50_000)
  draw_ground_point(x, +50_000)
}
draw_ground_point(3000_000, 0)

FOR k = 1000_000 TO 10000_000 BY 1000_000 DO
  IF k>3000_000 DO draw_ground_point( k, 0)
  setcolour(col_lightmajenta)
  draw_ground_point(-k, 0)
setcolour(col_red)
  draw_ground_point( 0, -k)
setcolour(col_green)
  draw_ground_point( 0, k)
}

AND initposition(n) BE SWITCHON n INTO
  { DEFAULT:
  CASE 1: // Take off position
    cgx, cgy, cgz := 100_000, 0, 100_000 //
    tdot, wdot, ldot := 0, 0, 0 // Stationary
    rtdot, rwdot, rldot := 0, 0, 0
    ctx, cty, ctz := One, 0, 0 // Direction cosines with
cwx, cwy, cwz := 0, One, 0 // six decimal digits
    clx, cly, clz := 0, 0, One // after to decimal point.
    ft1, fw1, fl1 := 0, 0, 0 // Previous linear forces
    rft1, rfw1, rfl1 := 0, 0, 0 // Previous rotational forces
    stepping := TRUE
    crashed := FALSE
    RETURN
  CASE 2: // Position on the glide slope
    cgx, cgy, cgz := -4000_000, 0, 1000_000 // height of 1000 ft
    tdot, wdot, ldot := 100_000, 0, 0 // 100 ft/s in direction x
    rtdot, rwdot, rldot := 0, 0, 0
ctx, cty, ctz := One, 0, 0 // Direction cosines with
cwx, cwy, cwz := 0, One, 0 // six decimal digits
clx, cly, clz := 0, 0, One // after to decimal point.

ft1, fw1, fl1 := 0, 0, 0 // Previous linear forces
rft1, rfw1, rfl1 := 0, 0, 0 // Previous rotational forces

stepping := TRUE
crashed := FALSE
RETURN

}
testing := FALSE

initSDL()

mkScreen("Tiger Moth", 800, 600)

// Declare a few colours in the pixel format of the screen

col_black := maprgb(0, 0, 0)
col_blue := maprgb(0, 0, 255)
col_green := maprgb(0, 255, 0)
col_yellow := maprgb(0, 255, 255)
col_red := maprgb(255, 0, 0)
col_majenta := maprgb(255, 0, 255)
col_cyan := maprgb(255, 255, 0)
col_white := maprgb(255, 255, 255)
col_darkgray := maprgb(64, 64, 64)
col_darkblue := maprgb(0, 0, 64)
col_darkgreen := maprgb(0, 64, 0)
col_darkyellow := maprgb(0, 64, 64)
col_darkred := maprgb(64, 0, 0)
col_darkmajenta := maprgb(64, 0, 64)
col_darkcyan := maprgb(64, 64, 0)
col_gray := maprgb(128, 128, 128)
col_lightblue := maprgb(128, 128, 255)
col_lightgreen := maprgb(128, 255, 128)
col_lightyellow := maprgb(128, 255, 255)
col_lightred := maprgb(255, 128, 128)
col_lightmajenta := maprgb(255, 128, 255)
col_lightcyan := maprgb(255, 255, 128)

plotScreen()

done := FALSE
debugging := FALSE
plotusage := FALSE

IF FALSE DO
{ // Test rdtab
  FOR a = -180_000 TO 180_000 BY 1000 DO
    { LET t = TABLE -180_000,0, 0,360, 180_000,0
    IF a MOD 6_000 = 0 DO writef("*n%14:", a/1000)
      writef(" %8.3d", rdtab(a, tltab))
    }
  newline()
  abort(1009)
}
IF FALSE DO
{
    // The angle function
    writef("x=%i5 y=%i5 angle=%9.3d*n", 1000, 1000, angle(1000, 1000))
    writef("x=%i5 y=%i5 angle=%9.3d*n", 0, 1000, angle(0, 1000))
    writef("x=%i5 y=%i5 angle=%9.3d*n",-1000, 1000, angle(-1000, 1000))
    writef("x=%i5 y=%i5 angle=%9.3d*n",-1000,-1000, angle(-1000,-1000))
    writef("x=%i5 y=%i5 angle=%9.3d*n", 1000,-1000, angle(1000,-1000))
    writef("x=%i5 y=%i5 angle=%9.3d*n", 0, -1000, angle(0,-1000))
    writef("x=%i5 y=%i5 angle=%9.3d*n", 60, 1, angle(60, 1))
    writef("x=%i5 y=%i5 angle=%9.3d*n", 60, -1, angle(60, -1))
    writef("x=%i5 y=%i5 angle=%9.3d*n",-1000, 1, angle(-1000, 1))
    writef("x=%i5 y=%i5 angle=%9.3d*n",-1000, -1, angle(-1000, -1))
    abort(1009)
}

aircraft := 1 // The default aircraft -- the tiger moth
//aircraft := 0 // The default aircraft -- the dart
done := FALSE
UNTIL done DO
{
    // Read joystick and keyboard events
    LET t0 = sdlmsecs()
    LET t1 = ?
    //writef("Calling processevents*n")
    processevents()
    IF stepping DO step()
    //writef("x=%9.3d y=%9.3d h=%9.3d tdot=%9.3d*n", cgx, cgy, cgz, tdot)
    plotscreen()
    //writef("Calling updatescreen*n")
    updatescreen()
    t1 := sdlmsecs()
    //writef("time %9.3d %9.3d %9.3d %9.3d*n", t0, t1, t1-t0, t0+100-t1)
    usage := 100*(t1-t0)/100
    //IF t0+100 < t1 DO
    //sdldelay(t0+100-t1)
    sdldelay(100)
    //sdldelay(900)
    //abort(1111)
5.19. TIGERMOTH FLIGHT SIMULATOR

}

writef("*nQuitting*n")
sdldelay(1_000)
closesdl()
RESULTIS 0
}

AND plotscreen() BE
{ LET mx = screenxsize/2
  LET my = screenysize - 70

  seteyeposition()

  fillscreen(col_blue)

  setcolour(col_lightcyan)

  //writef("done=%n*n", done)
drawstring(240, 50, done -> "Quitting", "Tiger Moth Flight Simulator")

  setcolour(col_gray)
  moveto(mx, my)
  drawby(0, cgz/100_000)

  setcolour(col_darkgray)
  drawfillrect(screenxsize-20-100, screenysize-20-100,
               screenxsize-20, screenysize-20)
  drawfillrect(screenxsize-50-100, screenysize-20-100,
               screenxsize-30-100, screenysize-20)
  drawfillrect(screenxsize-20-100, screenysize-50-100,
               screenxsize-20, screenysize-30-100)

  IF crashed DO
  { setcolour(col_red)
    plotf(mx-50, my+10, "CRASHED")
  }

  setcolour(col_red)
  moveto(mx, my)
  drawby(cgx/100_000, cgy/100_000)

  { LET pos = muldiv(40, c_thrust, 32768)
    setcolour(col_red)
    drawfillrect(screenxsize-45-100, pos+screenysize-15-100,
                 screenxsize-45, screenysize-15)
CHAPTER 5. INTERACTIVE GRAPHICS IN BCPL USING SDL

screenxsize-35-100, pos+screenysize- 5-100)

{ LET pos = multdiv(45, c_rudder, 32768)
  setcolour(col_red)
  drawfillrect(pos+screenxsize-25-50, -5+screenysize-40-100,
              pos+screenxsize-15-50, +5+screenysize-40-100)
}

{ LET posx = multdiv(45, c_aileron, 32768)
  LET posy = multdiv(45, c_elevator, 32768)
  setcolour(col_red)
  drawfillrect(posx+screenxsize-25-50, posy+screenysize-25-50,
              posx+screenxsize-15-50, posy+screenysize-15-50)
}

setcolour(col_majenta)
moveto(mx+200, my)
drawby(ctx/20_000, cty/20_000)

setcolour(col_lightblue)

IF debugging DO
  { plotf(20, my, "Thrust=%6i Elevator=%6i Aileron=%6i Rudder=%6i",
          c_thrust, c_elevator, c_aileron, c_rudder)
    plotf(20, my- 15, "x=%9.3d y=%9.3d z=%9.3d", cgx, cgy, cgz)
    plotf(20, my- 30, "tdot=%9.3d wdot=%9.3d ldot=%9.3d", tdot, wdot, ldot)
    plotf(20, my- 45, "atl=%9.3d atw=%9.3d awl=%9.3d", atl, atw, awl)
    plotf(20, my- 60, "ct %9.6d %9.6d %9.6d", ctx,cty,ctz)
    plotf(20, my- 75, "cw %9.6d %9.6d %9.6d", cwx,cwy,cwz)
    plotf(20, my- 90, "cl %9.6d %9.6d %9.6d", clx,cly,clz)
    plotf(20, my-105, "ft =%8.3d fw =%8.3d fl =%8.3d", ft, fw, fl)
    plotf(20, my-120, "rft =%9.6d rfw=%9.6d rfl=%9.6d", rft,rfw,rfl)
  }

IF plotusage DO
  { plotf(20, my-135, "CPU usage = %3i\%", usage)
  }

draw_artificial_horizon()

drawgroundpoints()
IF eyedir DO plotcraft()
updatescreen()
5.19. TIGERMOTH FLIGHT SIMULATOR

AND seteyeposition() BE
{ cetx, cety, cetz := 0, 0, 0
cewx, cewy, cewz := 0, 0, 0
celx, cely, celz := 0, 0, 1
   // Set eye position relative to CG of the aircraft
eyex, eyey, eyez := -eyedist, 0, 0
}

AND seteyeposition1() BE
{ LET d1 = eyedist
   LET d2 = d1*707/1000
   LET d3 = d2/3

cetx, cety, cetz := 0, 0, 0
cewx, cewy, cewz := 0, 0, 0
celx, cely, celz := 0, 0, 1
   // Set eye position relative to CG of the aircraft
eyex, eyey, eyez := -eyedist, 0, 0  // Relative eye position
UNLESS 0<=eyedir<=8 DO eyedir := 1

IF hatdir & sdlmsecs()>hatmsecs+100 DO
{ eyedir := ((angle(ctx, cty)+360_000+22_500) / 45_000) & 7
   // dir = 0  heading N
   // dir = 1  heading NE
   // dir = 2  heading E
   // dir = 3  heading SE
   // dir = 4  heading S
   // dir = 5  heading SW
   // dir = 6  heading W
   // dir = 7  heading NW
SWITCHON hatdir INTO
{ DEFAULT:
CASE #b0001: ENDCASE // Forward
CASE #b0011: eyedir := eyedir+1; ENDCASE // Forward right
CASE #b0010: eyedir := eyedir+2; ENDCASE // Right
CASE #b0110: eyedir := eyedir+3; ENDCASE // Backward right
CASE #b0100: eyedir := eyedir+4; ENDCASE // Backward
CASE #b1100: eyedir := eyedir+5; ENDCASE // Backward left
CASE #b1000: eyedir := eyedir+6; ENDCASE // Left
CASE #b1011: eyedir := eyedir+7; ENDCASE // Forward left
}
eyedir := (eyedir & 7) + 1
hatdir := 0

writef("ctx=%9.6d cty=%9.6d eyedir=%n eyedist=%9.3d\n", ctx, cty, eyedir, eyedist)
//abort(1009)
}

SWITCHON eyedir INTO
{ DEFAULT:

    CASE 0: // Pilot's view
        cetx, cety, cetz := ctx, cty, ctz
        cewx, cewy, cewz := cwx, cwy, cwz
        celx, cely, celz := clx, cly, clz
        eyex, eyey, eyez := 0, 0, 0 // Relative eye position
        RETURN

    CASE 1: // North
        cetx, cety, cetz := One, 0, 0
        cewx, cewy, cewz := 0, One, 0
        celx, cely, celz := 0, 0, One
        eyex, eyey, eyez := -d1, 0, d3 // Relative eye position
        RETURN

    CASE 2: // North east
        cetx, cety, cetz := D45, D45, 0
        cewx, cewy, cewz := -D45, D45, 0
        celx, cely, celz := 0, 0, One
        eyex, eyey, eyez := -d2, -d2, d3 // Relative eye position
        RETURN

    CASE 3: // East
        cetx, cety, cetz := 0, One, 0
        cewx, cewy, cewz := -One, 0, 0
        celx, cely, celz := 0, 0, One
        eyex, eyey, eyez := 0, -d1, d3 // Relative eye position
        RETURN

    CASE 4: // South east
        cetx, cety, cetz := -D45, D45, 0
        cewx, cewy, cewz := -D45, -D45, 0
        celx, cely, celz := 0, 0, One
        eyex, eyey, eyez := d2, -d2, d3 // Relative eye position
        RETURN

}
5.19. **TIGERMOHT FLIGHT SIMULATOR**

CASE 5: // South
    cetx, cety, cetz := -One, 0, 0
    cewx, cewy, cewz := 0, -One, 0
    celx, cely, celz := 0, 0, One
    eyex, eyey, eyez := d1, 0, d3 // Relative eye position
    RETURN

CASE 6: // South west
    cetx, cety, cetz := -D45, -D45, 0
    cewx, cewy, cewz := D45, D45, 0
    celx, cely, celz := 0, 0, One
    eyex, eyey, eyez := d2, d2, d3 // Relative eye position
    RETURN

CASE 7: // West
    cetx, cety, cetz := 0, -One, 0
    cewx, cewy, cewz := One, 0, 0
    celx, cely, celz := 0, 0, One
    eyex, eyey, eyez := 0, d1, d3 // Relative eye position
    RETURN

CASE 8: // North west
    cetx, cety, cetz := D45, -D45, 0
    cewx, cewy, cewz := D45, -D45, 0
    celx, cely, celz := 0, 0, One
    eyex, eyey, eyez := -d2, d2, d3 // Relative eye position
    RETURN

AND processes_events() BE WHILE get_event() SWITCH ON event_type INTO
{ DEFAULT:
    // writef("Unknown event type = %n", event_type)
    LOOP

CASE sdl_keydown:
    SWITCH ON capital_ch(event_a2) INTO
    { DEFAULT: LOOP

    CASE 'Q': done := TRUE; LOOP
    CASE 'D': debugging := ~debugging; LOOP
    CASE 'T': testing := ~testing; LOOP
CHAPTER 5. INTERACTIVE GRAPHICS IN BCPL USING SDL

CASE 'U': plotusage := ~plotusage; LOOP

CASE 'G': // Position aircraft on the glide path
    initposition(2)
    LOOP

CASE 'L': // Position the aircraft ready for take off
    initposition(1)
    LOOP

CASE 'N': // Reduce eye distance
    eyedist := eyedist*5/6
    IF eyedist<60_000 DO eyedist := 60_000
    LOOP

CASE 'F': // Increase eye distance
    eyedist := eyedist*6/5
    LOOP

CASE 'S': aircraft := (aircraft+1) MOD 3; LOOP

CASE 'Z': c_trimthrust := c_trimthrust - 500
    c_thrust := c_thrust - 500; LOOP

CASE 'X': c_trimthrust := c_trimthrust + 500
    c_thrust := c_thrust + 500; LOOP

CASE '<': c_trimrudder := c_trimrudder - 500
    c_rudder := c_rudder - 500; LOOP

CASE '>': c_trimrudder := c_trimrudder + 500
    c_rudder := c_rudder + 500; LOOP

CASE 'O': eyedir, hatdir := 0, 0; LOOP // Pilot’s view

CASE '1': hatdir, hatmsecs := #b0001, 0; LOOP // From behind

CASE '2': hatdir, hatmsecs := #b0011, 0; LOOP // From behind right

CASE '3': hatdir, hatmsecs := #b0110, 0; LOOP // From right

CASE '4': hatdir, hatmsecs := #b1100, 0; LOOP // From in front right

CASE '5': hatdir, hatmsecs := #b0110, 0; LOOP // From in front

CASE '6': hatdir, hatmsecs := #b1100, 0; LOOP // From in front left

CASE '7': hatdir, hatmsecs := #b1000, 0; LOOP // From left

CASE '8': hatdir, hatmsecs := #b1011, 0; LOOP // From behind left

CASE sdle_arrowup: c_trimelevator := c_trimelevator + 500
    c_elevator := c_elevator + 500; LOOP
CASE sdl e_arrowdown: c_trimelevator := c_trimelevator-500
c_elevator := c_elevator-500; LOOP
CASE sdl e_arrowright: c_trimailer on := c_trimailer on +500
c_aileron := c_aileron+500; LOOP
CASE sdl e_arrowleft: c_trimailer on := c_trimailer on -500
c_aileron := c_aileron-500; LOOP
}
LOOP

CASE sdl e_joyaxismotion: // 7
{ LET which = eventa1
LET axis = eventa2
LET value = eventa3
//writef("axismotion: which=%n axis=%n value=%n*n", which, axis, value)
SWITCHON axis INTO
{ DEFAULT: LOOP
CASE 0: c_aileron := c_trimailer on+value; LOOP // Aileron
CASE 1: c_elevator := c_trimailer on-value; LOOP // Elevator
CASE 2: c_thrust := c_trimthrust-value+32768; LOOP // Throttle
CASE 3: c_rudder := c_trimrudder+value; LOOP // Rudder
CASE 4: LOOP // Right throttle
}
}

CASE sdl e_joyhatmotion:
{ LET which = eventa1
LET axis = eventa2
LET value = eventa3
//writef("joyhatmotion %n %n %n*n", eventa1, eventa2, eventa3)
SWITCHON value INTO
{ DEFAULT: LOOP
CASE #b0000: // None
CASE #b0001: // North
CASE #b0011: // North east
CASE #b0010: // East
CASE #b0110: // South east
CASE #b0100: // South
CASE #b1100: // South west
CASE #b1000: // West
CASE #b1001: // North west
IF value>hatdir DO
{ hatdir, hatmsecs := value, sdlmsecs()
//writef("hatdir=%b4 %n msecs*n", hatdir, hatmsecs)
This chapter has used the rather primitive SDL graphics library and has typically drawn everything pixel by pixel, even when drawing 3D images involving hidden surface removal. The result is quite slow but is educational since by
looking at the BCPL graphics library (sdl.h, sdl.b) you can see how lines, circles and other shapes can be drawn. You can also see how hidden surface removal can be implemented. The disadvantage is that the library does not take advantage of the extraordinary power of the graphics hardware available on most modern machines. The next chapter presents a BCPL interface to the much more sophisticated OpenGL library that can take full advantage of the machine’s graphics hardware. This give much improved performance and allows for much more realistic moving images.

Even without using OpenGL, you can considerably improve performance by using the native code implementation of BCPL. For instance, the **bucket** and **tiger** programs can be compiled and run by typing the following.

```bash
cd ../../natbcpl
make -f MakefileRaspiSDL clean
make -f MakefileRaspiSDL bucket
./bucket
./tiger
```
Chapter 6

Interactive Graphics in BCPL using OpenGL

This chapter and the software it describes is still under development but is at last beginning to work. It is possible that I will upgrade to SDL2, provided I can get it to work on the Raspberry Pi, since it has many advantages over the older SDL. In particular, it can interface with OpenGL ES. This upgrade will cause several changes in both this and the previous chapter.

A second major change is that I have at last decided, after 50 years, to add single length floating point operations to the standard BCPL distribution since these are useful when interacting with OpenGL. This is a fairly major change since it also requires an upgrade to the Sial system and the creation of sial-686.b and a major modification to sial-arm.

OpenGL is a sophisticated library providing an efficient way of generating 3D graphical images using the full power of the graphics hardware available on most machines. Unfortunately this library comes in two forms. The full version, called OpenGL, is typically available on desktop and laptop computers while a cutdown version, called OpenGL ES, is typically available on smart phones and tablets where memory and computing power is more restricted. OpenGL ES is the version available on the Raspberry Pi.

Currently, OpenGL ES is often not available on the larger machines, so the BCPL GL library provides the same graphics facilities independent of which version of OpenGL is being used. OpenGL ES is mostly a subset of the features available in the full version of OpenGL. The BCPL GL library is designed to be easy to use and so only provides a subset of this subset.

Currently SDL can call OpenGL directly but not OpenGL ES. Although very simple, SDL provides a good interface with the keyboard, joysticks, the sound system and clocks. If SDL cannot be combined with OpenGL ES, other mechanisms (such as EGL) will be used to access these vital peripheral devices. Whichever version of OpenGL is used, the graphics features available to BCPL will be the same. To access these features the BCPL code will need to insert the
header files cintcode/g/gl.h and cintcode/g/gl.b. The low level OpenGL functions are available via sys calls as defined in sysc/glfn.c, but users will normally use the higher level BCPL functions defined in g/gl.b.

OpenGL makes extensive use of 32-bit floating point numbers but standard BCPL only provides limited floating point facilities via the sys interface. Where OpenGL requires floating point numbers, BCPL programs will normally use scaled fixed point values and have them converted to floating point by functions in the BCPL GL library.

6.1 Introduction to OpenGL

OpenGL is primarily designed to generate 2D images on the screen of 3D scenes composed of huge numbers of points, lines and triangles in three dimensions using the full power of the graphics hardware available on most computers. The graphics hardware is usually sufficiently powerful to display scenes involving hundreds of thousands of triangles with hidden surface removal and sophisticated lighting effects at a sufficient rate to provide smooth moving images.

OpenGL makes extensive use of vertices to represent points, ends of lines and the corners of triangles. Each vertex is specified by up to 8 attributes numbered from 0 to 7, each consisting of four components. Although OpenGL allows other data types, the BCPL interface insists that all attribute components are 32-bit floating point numbers. Attributes are used to represent the coordinates, colours and other properties of the vertices. The GL library provides facilities for defining vertices and transmitting them to the graphics hardware where they can be processed efficiently. Vertices are numbered from zero upwards. Points, lines and triangles can be specified using these vertex numbers. For scenes involving a huge number of triangles, it is usual to specify their vertex numbers in index arrays which can either be held in user memory, or, for greater efficiency, they can be transmitted to memory owned by the graphics hardware.

When the graphics hardware processes a triangle, it must first perform a calculation on each of its vertices to discover their pixel coordinates and other properties before it sets about the rasterisation process of calculating the position and colour of every pixel resulting from the triangle. The vertex computation is typically done by a user provided program called a vertex shader that runs on the graphics hardware. The BCPL GL library has a function to read a vertex shader program from file, compile it and transmit it to the graphics hardware. Each pixel generated during rasterisation involves the executions of another user provided program run on the graphics hardware called a fragment shader. As with vertex shaders, the BCPL GL library has a function to read a fragment shader program from file, compile it and transmit it to the graphics hardware. Vertex and fragment shaders use the same simple programming language that will be described later.
Vertex shaders can access the attributes of the vertex it is processing, and can also access global quantities, called \textit{uniforms}, which are available for all vertices. Uniform variables typically contain data about the rotation and position of objects in the scene being displayed as well as information about how it is being viewed. This might, for instance, be the position and orientation of a camera that is viewing the scene. Every time the graphics hardware generates a new screen image the position and rotations of objects in the scene may change as well as the position and orientation of the camera. Provided the graphics hardware is efficient enough, the whole scene should seem to move smoothly.

The output of vertex shaders are passed to the fragment shader via, so called, \textit{varying variables}. The vertex shader will calculate the value of each varying variable at the position of its vertex, but if a line or triangle is being drawn, the value received by the fragment shader for each pixel will be a linear interpolation of the corresponding varying variables of the vertices that define the line or triangle. So, for instance, the colour can change smoothly over the surface of a displayed triangle. The graphics hardware will perform this interpolation efficiently. Fragment shaders can also access uniform variables. Such data can, for instance, be used to control lighting effects.

In addition to the x-y screen coordinates of the apparent position of a vertex, the vertex shader often calculates the depth into the screen of its position. This value can be used to eliminate pixels that are hidden behind surfaces that are closer to the camera. Again, the graphics hardware can perform this hidden surface removal efficiently.

The shader language allows users to give names to attribute variables using declarations such as \texttt{attribute vec3 a\_position} and \texttt{attribute vec4 a\_colour}. Since the position and colour of vertices can be set up by the user, it is necessary to know which attribute locations are being used for these quantities. The BCPL GL library provides the function \texttt{glGetAttribLocation(...)} to find out where attributes were located after the shaders have been compiled and linked. An alternative mechanism in which the user chooses these locations before linking is available but is not recommended.

### 6.2 Geometric Transformations

Before giving an example program that uses OpenGL, we need to understand some of the mathematics involved in rotating a model in three dimensions and observing it from an eye position that can be moved.

We saw on page 280 that two dimensional rotations can be performed by multiplying the coordinates by a 2 by 2 matrix. It should be of little surprise to find the rotations in three dimensions can be performed using 3 by 3 matrices, however using 4 by 4 matrices turns out to be even more better since it allows for other useful tranformations to be performed in addition to simple rotations.
OpenGL and graphics hardware provide efficient implementations of 4 by 4 matrix operations so we will use these for most of the geometric transformations we need.

When 4 by 4 matrices are multiplied together and the rule is as follows.

\[
\begin{pmatrix}
 a & b & c & d \\
 . & . & . & . \\
 . & . & . & . \\
 x & y & z & w
\end{pmatrix}
\begin{pmatrix}
 . & . & . & . \\
 . & . & . & . \\
 . & . & . & . \\
 . & . & . & .
\end{pmatrix}
= 
\begin{pmatrix}
 . & . & . & . \\
 . & . & . & . \\
 . & . & . & . \\
 t & t & t & t
\end{pmatrix}
\]

where \( t = ax + by + cz + dw \), that is the value in the \( i^{th} \) row and \( j^{th} \) column of the result is the sum of the products of the elements of the \( i^{th} \) row of the left hand matrix with the corresponding elements of the \( j^{th} \) column of the right hand one. The matrices do not have to be square, all that is required is that the number of columns of the left hand matrix must equal the number of rows of the right hand one.

If \( A, B \) and \( C \) are three 4 by 4 matrices then \((AB)C = A(BC)\). This can be seen by considering the product:

\[
\begin{pmatrix}
a_{00} & a_{01} & a_{02} & a_{03} \\
a_{10} & a_{11} & a_{12} & a_{13} \\
a_{20} & a_{21} & a_{22} & a_{23} \\
a_{30} & a_{31} & a_{32} & a_{33}
\end{pmatrix}
\begin{pmatrix}
b_{00} & b_{01} & b_{02} & b_{03} \\
b_{10} & b_{11} & b_{12} & b_{13} \\
b_{20} & b_{21} & b_{22} & b_{23} \\
b_{30} & b_{31} & b_{32} & b_{33}
\end{pmatrix}
\begin{pmatrix}
c_{00} & c_{01} & c_{02} & b_{03} \\
c_{10} & c_{11} & c_{12} & b_{13} \\
c_{20} & c_{21} & c_{22} & b_{23} \\
c_{30} & c_{31} & c_{32} & b_{33}
\end{pmatrix}
\]

It is fairly easy to see that the value in the \( i^{th} \) row and \( j^{th} \) column of the result is the sum of 16 terms of the form \( a_{ip} b_{pq} c_{qj} \) with \( p \) and \( q \) taking all values between 0 and 3 and this is independent of whether the left hand or right hand pair of matrices are multiplied first. This is analogous to the rule in ordinary arithmetic that, for instance, \((10 \times 11) \times 12 = 10 \times (11 \times 12)\). But note that with matrix multiplication \( AB \) is typically not equal to \( BA \), just as rotating an object about the X-axis and then the Y-axis is usually different from first rotating about the Y-axis and then the X-axis.

To gain some feeling for what 4 by 4 matrix multiplication can do we will look at a few special cases. But first we should see how the four coordinates \((x, y, z, w)\) are used to represent a point in three dimensions. The conventional approach is to regard them as, so called, **homogenious** coordinates in which only the ratios between them are significant. So, if all four coordinates are multiplied by the same constant, the result still represents the same point. By convention \((x, y, z, w)\) represents the point whose three dimensional coordinates are \((x/w, y/w, z/w)\). We will often use \((x, y, z, 1)\) to represent a point with coordinates \((x, y, z)\).

The first special case is as follows.
CHAPTER 6. INTERACTIVE GRAPHICS IN BCPL USING OPENGL

\[
\begin{pmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{pmatrix}
\begin{pmatrix}
a & b & c & d \\
e & f & g & h \\
i & j & k & l \\
m & n & o & p
\end{pmatrix}
= 
\begin{pmatrix}
a & b & c & d \\
e & f & g & h \\
i & j & k & l \\
m & n & o & p
\end{pmatrix}
\]

Since this matrix leaves its operand unchanged it is called the identity matrix. Another special case is the following.

\[
\begin{pmatrix}
1 & 0 & 0 & X \\
0 & 1 & 0 & Y \\
0 & 0 & 1 & Z \\
0 & 0 & 0 & 1
\end{pmatrix}
\begin{pmatrix}
x \\
y \\
z \\
1
\end{pmatrix}
= 
\begin{pmatrix}
x + X \\
y + Y \\
z + Z \\
1
\end{pmatrix}
\]

This is called a translation matrix since it moves every point of a model by the same amount in three dimensions without rotation. The following matrix will rotate every vertex of the model about the Z-axis by an angle \( \theta \).

\[
\begin{pmatrix}
\cos \theta & -\sin \theta & 0 & 0 \\
\sin \theta & \cos \theta & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{pmatrix}
\begin{pmatrix}
x \\
y \\
z \\
1
\end{pmatrix}
= 
\begin{pmatrix}
x \cos \theta - y \sin \theta \\
x \sin \theta + y \cos \theta \\
z \\
1
\end{pmatrix}
\]

You can see this since it leaves \( z \) and \( w \) unchanged while replacing \( x \) and \( y \) by \( x \cos \theta - y \sin \theta \) and \( x \sin \theta + y \cos \theta \), respectively, which corresponds to a clockwise rotation of angle \( \theta \) when viewing along the z-axis from the origin.

These are two other similar matrices for rotations about the X and Y axes, namely:

\[
\begin{pmatrix}
1 & 0 & 0 & 0 \\
0 & \cos \theta & -\sin \theta & 0 \\
0 & \sin \theta & \cos \theta & 0 \\
0 & 0 & 0 & 1
\end{pmatrix}
\begin{pmatrix}
x \\
y \\
z \\
1
\end{pmatrix}
= 
\begin{pmatrix}
x \\
y \cos \theta - z \sin \theta \\
y \sin \theta + z \cos \theta \\
1
\end{pmatrix}
\]

and

\[
\begin{pmatrix}
\cos \theta & 0 & \sin \theta & 0 \\
0 & 1 & 0 & 0 \\
-\sin \theta & 0 & \cos \theta & 0 \\
0 & 0 & 0 & 1
\end{pmatrix}
\begin{pmatrix}
x \\
y \\
z \\
1
\end{pmatrix}
= 
\begin{pmatrix}
x \cos \theta + z \sin \theta \\
y \\
-x \sin \theta + z \cos \theta \\
1
\end{pmatrix}
\]
6.3. **VIEWING THE SCENE**

If \((a, b, c)\), \((d, e, f)\) and \((g, h, i)\) are direction cosines, they will correspond to three mutually orthogonal points on the unit sphere centred at the origin. The following matrix will then rotate the model coordinates \((1, 0, 0, 1)\), \((0, 1, 0, 1)\) and \((0, 0, 1, 1)\) to \((a, b, c, 1)\), \((d, e, f, 1)\) and \((g, h, i, 1)\).

\[
\begin{pmatrix}
  a & d & g & 0 \\
  b & e & h & 0 \\
  c & f & i & 0 \\
  0 & 0 & 0 & 1 
\end{pmatrix}
\]

Thus it will rotate the model about the origin, without deformation, to any desired orientation.

### 6.3 Viewing the Scene

Suppose we have a model specified by vertices with \(xyz\) coordinates near the origin \((O)\) and we wish to view it from an eye position \((P)\) whose coordinates are \((a,b,c)\) as shown in the following diagram.

![Diagram](image.png)

A good strategy is to think of the eye as rigidly attached to the model and perform two rotations of both the model and the eye. The first is an anticlockwise rotation \((R1)\) of \(\theta\) degrees about the Y-axis to bring the eye position into the YZ plane. The second rotation \((R2)\) is of \(\phi\) degrees clockwise about the X-axis to bring the eye position onto the Z-axis. Since the eye is rigidly connected to the model, its shape, as seen from the eye, will not have changed, however the XY plane will now be parallel to the display screen, so the \(x\) and \(y\) coordinates will respectively represent horizontal and vertical displacements on the screen, and \(z\) will be a measure of the depth of the vertex into the screen. Notice that we do not need to calculate the angles \(\theta\) and \(\phi\) since we only need their cosines and sines. These are as follows:
\[
\cos \theta = \frac{RQ}{OQ} = \frac{c}{\sqrt{a^2+c^2}} \\
\sin \theta = \frac{RQ}{OQ} = \frac{a}{\sqrt{a^2+c^2}} \\
\cos \phi = \frac{OQ}{OP} = \frac{\sqrt{a^2+c^2}}{\sqrt{a^2+b^2+c^2}} \\
\sin \phi = \frac{QP}{OP} = \frac{b}{\sqrt{a^2+b^2+c^2}}
\]

We can thus easily construct the matrices for the two rotations \(R_1\) and \(R_2\) that will move the eye position from \(P\) to a point on the Z axis. These can be multiplied together to give a single matrix to perform both rotations. Care is needed since the first rotation is anti-clockwise about the Y axis while the second is clockwise about the X axis. After these two rotations the eye position will be on the \(z\) axis at the same distance from the origin as it was before the rotations. However, it is sometimes convenient the change the distance between the eye and the centre of the model to, say, \(d\) units. We can do this and change the origin to the eye position by multiplying by the matrix:

\[
\begin{pmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & -d \\
0 & 0 & 0 & 1
\end{pmatrix}
\]

Notice that this moves every vertex of the model in the negative \(z\) direction by a distance \(d\).

The next transformation to apply calculates the perspective view in which distant features of the model look smaller than those that are close to the eye. The following diagram shows the YZ plane when viewed from the X direction.

\[
\frac{ST}{OT} = \frac{PQ}{OQ}
\]

It is easy to see that the triangles OST and OPQ have the same shape (mathematically they are similar). This implies that
and so

\[ ST = \frac{OT}{OQ} \times \frac{y}{z} \]

Thus the y position on the screen depends on the y value of the point divided by its z value and multiplied by a scaling factor. The important thing to note is that this projection requires a division by z but this cannot be done by matrix multiplication. However, all is not lost. The following diagram shows what is required.

It shows a truncated pyramid with a face \((ABCD)\) near the origin (where the eye is placed) and a more distant similar face \((PQRS)\) labelled far. Parts of the scene that are behind the eye or too close will not be displayed, nor will parts that are too distant or out of the field of view. So only points inside the truncated pyramid contribute to the final image on the screen. All other parts of the scene are said to be culled.

The details of the truncated pyramid can be completely specified by \(n\) and \(f\) the distances from the origin of the near and far faces, and \((l,b)\) and \((r,t)\) the \(xy\) coordinates of \(D\) and \(B\). Using these six values we can construct the following extraordinary 4 by 4 matrix.

\[
P = \begin{pmatrix}
\frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\
0 & \frac{2n}{l-b} & \frac{r+l}{l-b} & 0 \\
0 & 0 & \frac{f+n}{f-n} & \frac{2nf}{f-n} \\
0 & 0 & -1 & 0
\end{pmatrix}
\]
The first thing to notice is that, once the six values \( n, f, l, r, b \) and \( t \) are known, the matrix just contains 16 constant elements and so corresponds to a linear transformation, and linear transformations have the useful property that straight lines map into straight lines. It turns out that \( \mathbf{P} \) transforms the truncated pyramid into a cube whose \( x, y \) and \( z \) coordinates all range from -1 to +1.

We can see this by considering what happens to each of the 8 vertices of the truncated pyramid. But first observe what happens when \( \mathbf{P} \) is applied to a point with homogeneous coordinates \((x, y, z, 1)\).

\[
\mathbf{P} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} +\frac{2nx}{r-l} + \frac{(r+l)z}{r-l} \\ +\frac{2ny}{t-b} + \frac{(t+b)z}{t-b} \\ -\frac{2nf}{f-n} + \frac{(f+n)z}{f-n} \\ -z \end{pmatrix}
\]

So the result represents a point with the following \( xyz \) coordinates.

\[
\begin{pmatrix} +\frac{2nl}{(r-l)n} - \frac{r+l}{r-l} \\ +\frac{2nt}{(t-b)n} - \frac{t+b}{t-b} \\ -\frac{2nf}{(f-n)n} + \frac{f+n}{f-n} \end{pmatrix}
\]

So when \( \mathbf{P} \) is applied to point \( A \) whose coordinates are \((l, t, -n)\) the result is:

\[
\begin{pmatrix} +\frac{2nl}{(r-l)n} - \frac{r+l}{r-l} \\ +\frac{2nt}{(t-b)n} - \frac{t+b}{t-b} \\ -\frac{2nf}{(f-n)n} + \frac{f+n}{f-n} \end{pmatrix} = \begin{pmatrix} -1 \\ +1 \\ -1 \end{pmatrix}
\]

So \( A(l, t, -n) \) maps to \((-1, +1, -1)\) and using similar algebra it is easy to see that the points \( B(r, t, -n), C(r, b, -n) \) and \( D(l, b, -n) \) map into \((+1, +1, -1), (+1, -1, -1) \) and \((-1, -1, -1), \) respectively.

Since \( OAP \) is a straight line, the coordinates of \( P \) are just those of \( A \) multiplied by a scaling factor of \( f/n \) The coordinates of \( P \) are thus \((lf/n, tf/n, -f)\) and when we apply \( \mathbf{P} \) the result is:

\[
\begin{pmatrix} +\frac{2nl}{(r-l)n} - \frac{r+l}{r-l} \\ +\frac{2nt}{(t-b)n} - \frac{t+b}{t-b} \\ -\frac{2nf}{(f-n)n} + \frac{f+n}{f-n} \end{pmatrix} = \begin{pmatrix} -1 \\ +1 \\ -1 \end{pmatrix}
\]
So \( P(lf/n, tf/n, -f) \) maps to \((-1, +1, +1)\) and, using similar algebra, it is easy to see that the points \( Q(rf/n, tf/n, -f) \), \( R(rf/n, bf/n, -f) \) and \( S(lf/n, bf/n, -f) \) map into \((+1, +1, +1), (+1, -1, +1)\) and \((-1, -1, +1)\), respectively. Thus the eight vertices of the truncated pyramid map into the eight corners of a \( 2 \times 2 \times 2 \) cube centred at the origin, and since the mapping is linear, the faces of the truncated pyramid map into the faces of the cube.

We can multiply all the transformation matrices described above to construct a single 4 by 4 matrix that will do the entire transformation. This can be transmitted to an OpenGL uniform variable where it can be used efficiently by the vertex shader for every vertex in the scene.

Finally, we can tell OpenGL the position, width and height of a rectangle on the screen to display the object. OpenGL will then efficiently transform the cube coordinates to screen coordinates using the \( z \) component to eliminate hidden surfaces.

### 6.4 A first OpenGL example

This example displays a rotating image containing either tigermoth or a hollow coloured cube modified to look somewhat like a missile with control surfaces. The rate of rotation about the three axes can be controlled by pressing \(<, >\) and the arrow keys. The model can be moved forward and back \((F,B)\), left and right \((L,R)\) and up and down \((U,D)\). The eye looks toward the centre of the model in a direction controlled by \(0, 1, 2, 3, 4, 5, 6\) and \(7\). The eye height is controlled by \(8\) and \(9\), and the eye distance is controlled by + and -. The program can be compiled and run by typing the following two commands.

```
c b gltst
gltst
```

The following is a typical frame generated by this program.
If the switch argument OBJ is given, the vertex and index data will be copied to the graphics hardware where it will be processed more efficiently. The source of the program is called bcplprogs/raspi/gltst.b and is as follows. (A description of how it works will be added in due course.)

/*
This program is a simple demonstration of the OpenGL interface.

The BCPL GL library is in g/gl.b with header g/gl.h and is designed to work unchanged with either OpenGL using SDL or OpenGL ES using EGL.

Implemented by Martin Richards (c) July 2014

History

23/03/15
Simplified this program to only display gltst.mdl with limited control.
The tigermoth is now displayed in the flight simulator gltiger.b.

20/12/14
Modified the cube to be like a square missile with control surfaces.

03/12/14
Began conversion to use floating point numbers.

Command argument:

OBJ Use OpenGL Objects for vertex and index data
-d Turn on debugging

Controls:

Q causes quit
P Output debugging info to the terminal
S Stop/start the stepping the image

Rotational controls

Right/left arrow Increase/decrease rotation rate about direction of thrust
Up/Down arrow Increase/decrease rotation rate about direction of right wing
> < Increase/decrease rotation rate about direction of lift

0,1,2,3,4,5,6,7 Set eye direction -- The eye is always looking towards the origin.
+,,- Increase/decrease eye distance
The transformations

The model is represented using three axes \( t \) (the direction of thrust), \( w \) the direction of the left wing and \( l \) (the direction of lift, orthogonal to \( t \) and \( w \)). These use the right hand convention, ie \( t \) is forward, \( w \) is left and \( l \) is up.

Real world coordinate use axes \( x \) (right), \( y \) (up) and \( z \) (towards the viewer). These also use the right hand convention.

\[
\begin{align*}
ctx; cty; ctz & \quad // \text{Direction cosines of direction } t \\
cwx; cwy; cwz & \quad // \text{Direction cosines of direction } w \\
clx; cly; clz & \quad // \text{Direction cosines of direction } l
\end{align*}
\]

\( \text{eyex, eyey, eyez} \) specify a point on the line of sight between the eye and the origin. The line of sight is towards the origin from this point.

\( \text{eyedistance} \) holds the distance between the eye and the origin.

Since standard BCPL now supports floating point operations and the latest Raspberry Pi (Model B-2) has proper support for floating point this program will phase out scales fixed point arithmetic and use floating point instead. This is a simple but extensive change.

*/

GET "libhdr"
GET "gl.h"
GET "gl.b" // Insert the library source code
GET "libhdr"
GET "gl.h"

GLOBAL {
    done:ug
    stepping
    debug
    glprog
    Vshader
    Fshader

    VertexLoc // Attribute variable locations
    ColorLoc
    DataLoc // data[0]=ctl data[1]=value
MatrixLoc       // Uniform variable locations
ControlLoc

CosElevator
SinElevator
CosRudder
SinRudder
CosAileron
SinAileron

modelfile

// The following variables are floating point number

ctx; cty; ctz       // Direction cosines of direction t
cwx; cwy; cwz       // Direction cosines of direction w
clx; cly; clz       // Direction cosines of direction l

rtdot; rwdot; rldot  // Anti-clockwise rotation rates
                     // about the t, w and l axes

eyex; eyey; eyez     // Coordinates of a point on the line of sight
                     // from to eye to the origin (0.0, 0.0, 0.0).
eyedistance        // The distance between the eye and the origin.

// The next four variables must be in consecutive locations
// since @VertexData is passed to loadmodel.
VertexData          // Vector of 32-bit floating point numbers
VertexDataSize      // = number of numbers in VertexData
IndexData           // Vector of 16-bit unsigned integers
IndexDataSize       // = number of 16-bit integers in IndexData

useObjects          // = TRUE if using OpenGL Objects
VertexBuffer
IndexBuffer

projectionMatrix    // is the matrix used by the vertex shader
                     // to transform the vertex coordinates to
                     // screen coordinates.
workMatrix          // is used when constructing the projection matrix.
}

LET start() = VALOF
{ LET m1 = VEC 15
LET m2 = VEC 15
LET argv = VEC 50
LET modelfile = "gltst.mdl"

projectionMatrix, workMatrix := m1, m2

UNLESS rdargs("obj/s,-d/s", argv, 50) DO
  { writef("Bad arguments for gltst*n")
    RETURN
  }

useObjects := argv!0  // obj/s
debug := argv!1       // -d/s

UNLESS glInit() DO
  { writef("*nOpenGL not available*n")
    RESULTIS 0
  }

writef("start: calling glMkScreen*n")
// Create an OpenGL window
screenxsize := glMkScreen("OpenGL First Test", 800, 680)
screenysize := result2
UNLESS screenxsize DO
  { writef("*nUnable to create an OpenGL window*n")
    RESULTIS 0
  }
writef("Screen Size is %n x %n*n", screenxsize, screenysize)

writef("start: calling glMkProg ")
glprog := glMkProg()
writef("=> glprog=%n*n", glprog);

IF glprog<0 DO
  { writef("*nUnable to create a GL program*n")
    RESULTIS 0
  }

// Read and Compile the vertex shader
writef("start: calling CompileV(%n,gltstVshader.sdr) ",glprog)
Vshader := Compileshader(glprog, TRUE, "gltstVshader.sdr")
writef("=> Vshader=%n*n", Vshader)

// Read and Compile the fragment shader
writef("start: calling CompileF(%n,gltstFshader.sdr) ",glprog)
Fshader := Compileshader(glprog, FALSE, "gltstFshader.sdr")
writef("=> Fshader=%n*", Fshader)

// Link the program
writef("start: calling glLinkProg(%n)*", glprog)
UNLESS glLinkProg(glprog) DO
    { writef("Unable to link a GL program"n)
        RESULTIS 0
    }

writef("start: calling glUseProgram(%n)*", glprog)	glUseProgram(glprog)

// Get attribute locations after linking
VertexLoc := glGetAttribLocation(glprog, "g_vVertex")
ColorLoc := glGetAttribLocation(glprog, "g_vColor")
DataLoc := glGetAttribLocation(glprog, "g_vData")
writef("VertexLoc=%n*", VertexLoc)
writef("ColorLoc=%n*", ColorLoc)
writef("DataLoc=%n*", DataLoc)

// Get uniform locations after linking
MatrixLoc := glGetUniformLocation(glprog, "matrix")
ControlLoc := glGetUniformLocation(glprog, "control")
writef("MatrixLoc=%n*", MatrixLoc)
writef("ControlLoc=%n*", ControlLoc)

// Load model
UNLESS loadmodel(modelfile, @VertexData) DO
    { writef("Unable to load model: %s*n", modelfile)
        RESULTIS 0
    }

IF debug DO
    { // Output the vertex and index data
        // as a debugging aid
        FOR i = 0 TO VertexDataSize-1 DO
6.4. A FIRST OPENGL EXAMPLE

{ IF i MOD 8 = 0 DO newline()
   writef(" %8.3d", sc3(VertexData!i))
}
newline()
FOR i = 0 TO (IndexDataSize-1)/2 DO
{ LET w = IndexData!i
   IF i MOD 6 = 0 DO writef("%n%i6: ", 2*i)
   writef(" %i5 %i5", w & #xFFFF, w>>16)
}
newline()
}

sys(Sys_gl, GL_Enable, GL_DEPTH_TEST) // This call is necessary
sys(Sys_gl, GL_DepthFunc, GL_LESS) // This the default

// A pixel written if incoming depth < buffer depth
// This assumes positive Z is into the screen, but
// remember the depth test is performed after all other
// transformations have been done.

TEST useObjects
THEN {
  // Setup the model using OpenGL objects
  writef("start: VertexDataSize=%n*n", VertexDataSize)
  VertexBuffer := sys(Sys_gl, GL_GenVertexBuffer, VertexDataSize, VertexData)

  // Tell GL the positions in VertexData of the xyz fields,
  // ie the first 3 words of each 8 word item in VertexData
  sys(Sys_gl, GL_EnableVertexAttribArray, VertexLoc);
  sys(Sys_gl, GL_VertexData, VertexLoc, // Attribute number for xyz data
       3, // 3 floats for xyz
       8, // 8 floats per vertex item in vertexData
       0) // Offset in words of the xyz data
  writef("start: VertexData xyz data copied to graphics object %n*n", VertexBuffer)

  // Tell GL the positions in VertexData of the rgb fields,
  // ie the second 3 words of each 8 word item in VertexData
  sys(Sys_gl, GL_EnableVertexAttribArray, ColorLoc);
  sys(Sys_gl, GL_VertexData, ColorLoc, // Attribute number rgb data
       3, // 3 floats for rgb data
       8, // 8 floats per vertex item in vertexData
       3) // Offset in words of the rgb data
542

CHAPTER 6. INTERACTIVE GRAPHICS IN BCPL USING OPENGL

writef("start: ColourData rgb data copied to graphics object %n*n", VertexBuffer)
// Tell GL the positions in VertexData of the kd fields,
// ie word 6 of each 8 word item in VertexData
sys(Sys_gl, GL_EnableVertexAttribArray, DataLoc);
sys(Sys_gl, GL_VertexData,
DataLoc,
// Attribute number rgb data
2,
// 2 floats for kd data
8,
// 8 floats per vertex item in vertexData
6)
// Offset in words of the kd data
writef("start: VertexData kd data copied to graphics object %n*n", VertexBuffer)
// VertexData can now be freed
//freevec(VertexData)
writef("start: IndexDataSize=%n*n", IndexDataSize)
IndexBuffer := sys(Sys_gl, GL_GenIndexBuffer, IndexData, IndexDataSize)
writef("start: IndexData copied to graphics memory object %n*n", IndexBuffer)
// IndexData can now be freed
//freevec(IndexData)
} ELSE {
// Setup the model not using objects
sys(Sys_gl, GL_EnableVertexAttribArray, VertexLoc);
sys(Sys_gl, GL_EnableVertexAttribArray, ColorLoc);
sys(Sys_gl, GL_EnableVertexAttribArray, DataLoc);
// The next call tells GL where the xyz fields of
// attribute VertexLoc appear in VertexData. It says
// that each vertex is specified by items consisting
// 8 words. The first 3 words of each item contains
// the xyz values.
glVertexData(VertexLoc,
3,
// 3 Values x, y, z
8,
// Stride of 8 words (=32 bytes)
// ie 8 values in VertexData per vertex
VertexData)
// Position of xyz value of vertex 0
// The next call tells GL where the rgb fields of
// attribute ColorLoc appear in VertexData. It says
// they are in 3 words at position 3 of each 8 word item.
glVertexData(ColorLoc,


6.4. A FIRST OPENGL EXAMPLE

3,       // 3 Values r, g, b
8,       // Stride in words (=32 bytes)
          // ie 8 values in VertexData per vertex
VertexData+3) // Position of rgb values of vertex 0

// The next call tells GL where the kd fields of
// attribute ColorLoc appear in VertexData. It says
// they are in the last 2 words of each 8 word item.
glVertexData(DataLoc,
    2,       // 2 Values k, d
    8,       // Stride in words (=32 bytes)
          // ie 8 values in VertexData per vertex
    VertexData+6) // Position of kd values of vertex 0
}

// Initialise the state
done := FALSE
stepping := FALSE

// Set the initial direction cosines to orient t, w and l in
// directions -z, -x and y, ie viewing the aircraft from behind.
ctx, cty, ctz := 0.0, 0.0, -1.0
cwd, cwy, cwz := -1.0, 0.0, 0.0
cld, clx, clz := 0.0, 1.0, 0.0

// rtdot, rwdot, rldot := 0.0, 0.0, 0.0
rtdot, rwdot, rldot := 0.003, 0.001, 0.002 // Rotate the model slowly

eyex, eyey, eyez := 0.0, 0.0, 1.0
eyedistance := 150.000

IF debug DO
{
glSetvec( workMatrix, 16,
    2.0, 0.0, 0.0, 0.0,
    0.0, 1.0, 0.0, 0.0,
    0.0, 0.0, 1.0, 0.0,
    0.0, 0.0, 0.0, 10.0
)

glSetvec( projectionMatrix, 16,
    1.0, 2.0, 3.0, 4.0,
    5.0, 6.0, 7.0, 8.0,
    9.0, 10.0, 11.0, 12.0,
}
13.0, 14.0, 15.0, 16.0
)
newline()
prmat(workMatrix)
writef("times\n")
prmat(projectionMatrix)
g1Mat4mul(workMatrix, projectionMatrix, projectionMatrix)
writef("gives\n")
prmat(projectionMatrix)
abort(1000)
}

UNTIL done DO
{
processevents()

// Only rotate the object if not stepping
UNLESS stepping DO
{
// If not stepping adjust the orientation of the model.
rotate(rtdot, rwdot, rldot)
}

// Set the model rotation matrix from model
// coordinates (t,w,l) to world coordinates (x,y,z)
g1Setvec( projectionMatrix, 16,
    ctx, cty, ctz, 0.0, // column 1
    cxw, cwy, cwz, 0.0, // column 2
    cwx, cly, clz, 0.0, // column 3
    0.0, 0.0, 0.0, 1.0 // column 4
)

// Rotate the model and eye until the eye is on the z axis
{
LET ex, ey, ez = eyex, eyey, eyez
LET oq = g1Radius2(ex, ez)
LET op = g1Radius3(ex, ey, ez)
LET cos_theta = ez #/ oq
LET sin_theta = ex #/ oq
LET cos_phi = oq #/ op
LET sin_phi = ey #/ op

// Rotate anti-clockwise about Y axis by angle theta
g1Setvec( workMatrix, 16,
    cos_theta, 0.0, sin_theta, 0.0, // column 1
    0.0, 1.0, 0.0, 0.0, // column 2
    #+#sin_theta, 0.0, cos_theta, 0.0, // column 3

6.4. A FIRST OPENGL EXAMPLE

0.0, 0.0, 0.0, 1.0 // column 4
}

glMat4mul(workMatrix, projectionMatrix, projectionMatrix)

// Rotate clockwise about X axis by angle phi
glSetvec( workMatrix, 16,
    1.0, 0.0, 0.0, 0.0, // column 1
    0.0, cos_phi, -sin_phi, 0.0, // column 2
    0.0, sin_phi, cos_phi, 0.0, // column 3
    0.0, 0.0, 0.0, 1.0 // column 4
)

glMat4mul(workMatrix, projectionMatrix, projectionMatrix)

// Change the origin to the eye position on the z axis by
// moving the model eyedistance in the negative z direction.
glSetvec( workMatrix, 16,
    1.0, 0.0, 0.0, 0.0, // column 1
    0.0, 1.0, 0.0, 0.0, // column 2
    0.0, 0.0, 1.0, 0.0, // column 3
    0.0, 0.0, -eyedistance, 1.0 // column 4
)

glMat4mul(workMatrix, projectionMatrix, projectionMatrix)
}

{ // Define the truncated pyramid for the view projection
// using the frustum transformation.
LET n, f = 0.1, 5000.0
LET fan, fsn = f#+n, f#-n
LET n2 = 2.0#*n
LET l, r = #-0.5, 0.5
LET ral, rsl = r#+l, r#-l
LET b, t = #-0.5, 0.5
LET tab, tsb = t#+b, t#-b

LET aspect = FLOAT screenxsize #/ FLOAT screenysize
LET fv = 2.0 #/ 0.5 // Half field of view at unit distance

glSetvec( workMatrix, 16,
    fv #/ aspect, 0.0, 0.0, 0.0, // column 1
    0.0, fv, 0.0, 0.0, // column 2
    0.0, 0.0, (f #+ n) #/ (n #- f), #-1.0, // column 3
    0.0, 0.0, (2.0 #* f #* n) #/ (n #- f), 0.0 // column 4
)
// The perspective matrix could be set more conveniently using
// glSetPerspective library function defined in g/gl.b
// glSetPerspective(workMatrix,
//                aspect, // Aspect ratio
//                0.5, // Field of view at unit distance
//                0.1, // Distance to near limit
//                5000.0) // Distance to far limit

g1Mat4mul(workMatrix, projectionMatrix, projectionMatrix)
}

// Send the matrix to uniform variable "matrix" for use by the
// vertex shader.
g1UniformMatrix4fv(MatrixLoc, glprog, projectionMatrix)

// Calculate the cosines and sines of the control surfaces.
{ LET RudderAngle = rldot * 100.0
  CosRudder := sys(Sys_flt, fl_cos, RudderAngle)
  SinRudder := sys(Sys_flt, fl_sin, RudderAngle)
}
{ LET ElevatorAngle = rwdot * 100.0
  CosElevator := sys(Sys_flt, fl_cos, ElevatorAngle)
  SinElevator := sys(Sys_flt, fl_sin, ElevatorAngle)
}
{ LET AileronAngle = rtdot * 100.0
  CosAileron := sys(Sys_flt, fl_cos, AileronAngle)
  SinAileron := sys(Sys_flt, fl_sin, AileronAngle)
}

// Send them to the graphics hardware as elements of the
// uniform 4x4 matrix "control" for use by the vertex shader.
{ LET control = VEC 15
  FOR i = 0 TO 15 DO control!i := 0.0

  control!00 := CosRudder // 0 0
  control!01 := SinRudder  // 0 1
  control!02 := CosElevator // 0 2
  control!03 := SinElevator // 0 3
  control!04 := CosAileron  // 1 0
  control!05 := SinAileron  // 1 1

  // Send the control values to the graphics hardware
6.4. A FIRST OPENGL EXAMPLE

```c
    glUniformMatrix4fv(ControlLoc, glprog, control)
}

// Draw a new image
    glClearColour(130, 130, 250, 255)
    glClearBuffer() // Clear colour and depth buffers
    drawmodel()

    glSwapBuffers()

delay(0.020) // Delay for 1/50 sec
}

sys(Sys_gl, GL_DisableVertexAttribArray, VertexLoc)
sys(Sys_gl, GL_DisableVertexAttribArray, ColorLoc)
sys(Sys_gl, GL_DisableVertexAttribArray, DataLoc)

delay(0.050)
    glClose()

RESULTIS 0
}

AND Compileshader(prog, isVshader, filename) = VALOF
{ // Create and compile a shader whose source code is
// in a given file.
// isVshader=TRUE if compiling a vertex shader
// isVshader=FALSE if compiling a fragment shader
    LET oldin = input()
    LET oldout = output()
    LET buf = 0
    LET shader = 0
    LET ramstream = findinoutput("RAM:"
    LET instream = findinput(filename)
    UNLESS ramstream & instream DO
    { writef("Compileshader: Trouble with i/o streams*n")
      RESULTIS -1
    }

    //Copy shader program to RAM:
    //writef("Compiling shader %s*n", filename)
    selectoutput(ramstream)
    selectinput(instream)
```
{ LET ch = rdch()
  IF ch=endstreamch BREAK
  wrch(ch)
} REPEAT

wrch(0) // Place the terminating byte

selectoutput(oldout)
endstream(instream)
selectinput(oldin)

buf := ramstream!scb_buf

shader := sys(Sys_gl,
             (isVshader -> GL_CompileVshader, GL_CompileFshader),
             prog,
             buf)

//writef("Compileshader: shader=%n*n", shader)
endstream(ramstream)
RESULTIS shader

AND drawmodel() BE
  TEST useObjects
  THEN { // Draw triangles using vertex and index data
          // held in graphics objects
    glDrawTriangles(IndexDataSize, 0)
  }
  ELSE { // Draw triangles using vertex and index data
          // held in main memory
    glDrawTriangles(IndexDataSize, IndexData)
  }

AND processevents() BE WHILE getevent() SWITCHON eventtype INTO
{ DEFAULT:
  //writef("processevents: Unknown event type = %n*n", eventtype)
  LOOP
}

CASE sdle_keydown:
  SWITCHON capitalch(eventa2) INTO
  { DEFAULT: LOOP

    CASE 'Q': done := TRUE
    LOOP

    SETUP: intr = intr + 1

    CASE 'W': inter := inter + 1
    LOOP
CASE 'A': abort(5555)
  LOOP

CASE 'P': // Print direction cosines and other data
  newline()
  writef("ct  %9.6d %9.6d %9.6d rtdot=%9.6d*n",
         sc6(ctx),sc6(cty),sc6(ctz), sc6(rtdot))
  writef("cw  %9.6d %9.6d %9.6d rwdot=%9.6d*n",
         sc6(cwx),sc6(cwy),sc6(cwz), sc6(rwdot))
  writef("cl  %9.6d %9.6d %9.6d rldot=%9.6d*n",
         sc6(clx),sc6(cly),sc6(clz), sc6(rldot))
  newline()
  writef("eyepos %9.3d %9.3d %9.3d*n",
          sc3(eyex), sc3(eyey), sc3(eyez))
  writef("eyedistance = %9.3d*n", sc3(eyedistance))
  LOOP

CASE 'S': stepping := ~stepping
  LOOP

CASE '0': eyex, eyez := 0.000, 1.000; LOOP
CASE '1': eyex, eyez := 0.707, 0.707; LOOP
CASE '2': eyex, eyez := 1.000, #1.000; LOOP
CASE '3': eyex, eyez := 0.707, #0.707; LOOP
CASE '4': eyex, eyez := 0.000, #0.000; LOOP
CASE '5': eyex, eyez := #0.707, #0.707; LOOP
CASE '6': eyex, eyez := #0.7000, 0.0000; LOOP
CASE '7': eyex, eyez := #0.707, 0.707; LOOP

CASE '=':
CASE '+': eyedistance := eyedistance * 1.1; LOOP

CASE '_':
CASE '-': IF eyedistance >= 1.0 DO
  eyedistance := eyedistance / 1.1
  LOOP

CASE '>': CASE '.': rldot := rldot + 0.0005; LOOP
CASE '<': CASE '.': rldot := rldot - 0.0005; LOOP

CASE sdlle_arrowdown: rwdot := rwdot + 0.0005; LOOP
CASE sdlle_arrowup: rwdot := rwdot - 0.0005; LOOP
CASE sdlle_arrowleft: rtdot := rtdot + 0.0005; LOOP
CASE sdle_arrowright: rtdot := rtdot #- 0.0005; LOOP
}
LOOP

CASE sdle_quit: // 12
  writef("QUIT\n");
  sys(Sys_gl, GL_Quit)
  LOOP
}

// Conversion functions between floating point and scaled values.
AND sc3(x) = glF2N(1_000, x)
AND sc6(x) = glF2N(1_000_000, x)

AND inprod(a,b,c, x,y,z) =
  \text{Return the cosine of the angle between two unit vectors.}
  \frac{a \#* x \#* b \#* y \#* c \#* z}{
  \sqrt{a^2 + b^2 + c^2} \sqrt{x^2 + y^2 + z^2}
}

AND rotate(t, w, l) BE
{
  \text{// Rotate the orientation of the aircraft}
  \text{// t, w and l are assumed to be small and cause}
  \text{// rotation about axis t, w, l. Positive values cause}
  \text{// anti-clockwise rotations about their axes.}

  LET tx = inprod(1.0, #-l, w, ctx,cwx,clx)
  LET wx = inprod( l, 1.0, #-t, ctx,cwx,clx)
  LET lx = inprod(#-w, t, 1.0, ctx,cwx,clx)
  LET ty = inprod(1.0, #-l, w, cty,cwy,clz)
  LET wy = inprod( l, 1.0, #-t, cty,cwy,clz)
  LET ly = inprod(#-w, t, 1.0, cty,cwy,clz)
  LET tz = inprod(1.0, #-l, w, ctz,cwz,clz)
  LET wz = inprod( l, 1.0, #-t, ctz,cwz,clz)
  LET lz = inprod(#-w, t, 1.0, ctz,cwz,clz)

  ctx, cty, ctz := tx, ty, tz
  cwx, cwy, cwz := wx, wy, wz
  clx, cly, clz := lx, ly, lz

  adjustlength(@ctx); adjustlength(@cwx); adjustlength(@clx)
  adjustortho(@ctx, @cwx); adjustortho(@ctx, @clx); adjustortho(@cwx, @clx)
}

AND adjustlength(v) BE
6.4. A FIRST OPENGL EXAMPLE

{ // This helps to keep vector v of unit length
    LET r = glRadius3(v!0, v!1, v!2)
    v!0 := v!0 #/ r
    v!1 := v!1 #/ r
    v!2 := v!2 #/ r
}

AND adjustortho(a, b) BE
{ // This helps to keep the unit vector b orthogonal to a
    LET a0, a1, a2 = a!0, a!1, a!2
    LET b0, b1, b2 = b!0, b!1, b!2
    LET corr = inprod(a0,a1,a2, b0,b1,b2)
    b!0 := b0 #- a0 #* corr
    b!1 := b1 #- a1 #* corr
    b!2 := b2 #- a2 #* corr
}

AND prmat(m) BE
{ prf8_3(m! 0)
    prf8_3(m! 4)
    prf8_3(m! 8)
    prf8_3(m!12)
    newline()
    prf8_3(m! 1)
    prf8_3(m! 5)
    prf8_3(m! 9)
    prf8_3(m!13)
    newline()
    prf8_3(m! 2)
    prf8_3(m! 6)
    prf8_3(m!10)
    prf8_3(m!14)
    newline()
    prf8_3(m! 3)
    prf8_3(m! 7)
    prf8_3(m!11)
    prf8_3(m!15)
    newline()
}

AND prv(v) BE
{ prf8_3(v!0)
    prf8_3(v!1)
    prf8_3(v!2)
    prf8_3(v!3)
AND prf8_3(x) BE writef(" %8.3d", sc3(x))

AND dbmatrix(m) BE //IF FALSE DO
{ LET x,y,z,w = ?,?,?,?
  LET v = @x
  LET n, p, one = #-0.5, #+0.5, 1.0
  prmat(m); newline()
  x,y,z,w := 1.0,0.0,0.0,1.0
  prv(v); glMat4mulV(m, v, v); writef(" => "); prv(v); newline()
  x,y,z,w := 0.0,1.0,0.0,1.0
  prv(v); glMat4mulV(m, v, v); writef(" => "); prv(v); newline()
  x,y,z,w := 0.0,0.0,1.0,1.0
  prv(v); glMat4mulV(m, v, v); writef(" => "); prv(v); newline()
  x,y,z,w := n,n,p,one
  prv(v); glMat4mulV(m, v, v); writef(" => "); prv(v); newline()
  x,y,z,w := p,n,n,one
  prv(v); glMat4mulV(m, v, v); writef(" => "); prv(v); newline()
  x,y,z,w := n,n,n,one
  prv(v); glMat4mulV(m, v, v); writef(" => "); prv(v); newline()
  x,y,z,w := n,p,p,one
  prv(v); glMat4mulV(m, v, v); writef(" => "); prv(v); newline()
  x,y,z,w := p,p,p,one
  prv(v); glMat4mulV(m, v, v); writef(" => "); prv(v); newline()
  x,y,z,w := p,p,n,one
  prv(v); glMat4mulV(m, v, v); writef(" => "); prv(v); newline()
  x,y,z,w := n,p,n,one
  prv(v); glMat4mulV(m, v, v); writef(" => "); prv(v); newline()
}

This program reads gltst.mdl, which was hand written, to obtain the vertex and index data representing the cube-like missile with control surfaces. The file gltst.mdl is as follows.

// This file holds the vertex and index data used by gltst.b

// Implemented by Martin Richards (c) June 2014

// OpenGL uses the right hand convention so for world coordinates
// we have:

// positive X is to the right
// positive Y is up and
// positive Z is towards the viewer
// so negative Z is into the screen

// The model uses axes T (thrust), W(left wing), L(lift) when
// representing an aircraft. These also use the right hand convention.

// The \( v \) parameters are
//
// \[
// t \quad w \quad l \quad r \quad g \quad b \quad k \quad d
// \]

// ie \( t \) = direction of thrust
// \( w \) = direction of left wing
// \( l \) = direction of lift

// \( k = 0 \) fixed surface
// \( k = 1 \) rudder
// \( k = 2 \) elevator
// \( k = 3 \) left aileron
// \( k = 4 \) right aileron

s 1000  // Scale: fixed point 1000 represents floating point 1.000

v  // Vertices: \( x \ y \ z \ r \ g \ b \)

// \( t \) is forward (direction of thrust)
// \( w \) is left (direction of left wing)
// \( l \) is up (direction of lift)

// \( t \quad w \quad l \quad r \quad g \quad b \quad k \quad d \)
+10500 +10500 -10500 1000 1000 0000 0 0  // 0 front left bottom yellow
+10500 -10500 -10500 1000 1000 0000 0 0  // 1 front right bottom yellow
-10500 -10500 -10500 1000 1000 0000 0 0  // 2 back right bottom yellow
-10500 +10500 -10500 1000 1000 0000 0 0  // 3 back left bottom yellow
+10500 -10500 -10500 0000 1000 0000 0 0  // 4 front right bottom green
+10500 -10500 +10500 0000 1000 0000 0 0  // 5 front right top green
-10500 -10500 +10500 0000 1000 0000 0 0  // 6 back right top green
-10500 -10500 -10500 0000 1000 0000 0 0  // 7 back right bottom green
+10500 -10500 +10500 0200 0400 1000 0 0  // 8 front right top light blue
+10500 +10500 +10500 0200 0400 1000 0 0  // 9 front left top light blue
-10500 +10500 +10500 0200 0400 1000 0 0  // 10 back left top light blue
\[
\begin{array}{cccccccc}
-10500 & -10500 & +10500 & 0200 & 0400 & 1000 & 0 & 0 & // 11 \ back \ right \ top \quad \text{light blue}
\end{array}
\]

\[
\begin{array}{cccccccc}
+10500 & +10500 & -10500 & 1000 & 0000 & 0000 & 0 & 0 & // 12 \ front \ left \ bottom \quad \text{red}
\end{array}
\]

\[
\begin{array}{cccccccc}
-10500 & +10500 & -10500 & 1000 & 0000 & 0000 & 0 & 0 & // 13 \ back \ left \ bottom \quad \text{red}
\end{array}
\]

\[
\begin{array}{cccccccc}
-10500 & +10500 & +10500 & 1000 & 0000 & 0000 & 0 & 0 & // 14 \ back \ left \ top \quad \text{red}
\end{array}
\]

\[
\begin{array}{cccccccc}
+10500 & +10500 & +10500 & 1000 & 0000 & 0000 & 0 & 0 & // 15 \ front \ left \ top \quad \text{red}
\end{array}
\]

\[
\begin{array}{cccccccc}
+10500 & +10500 & -10500 & 0000 & 0000 & 1000 & 0 & 0 & // 16 \ front \ left \ bottom \quad \text{blue}
\end{array}
\]

\[
\begin{array}{cccccccc}
+10500 & +10500 & +10500 & 0000 & 0000 & 1000 & 0 & 0 & // 17 \ front \ left \ top \quad \text{blue}
\end{array}
\]

\[
\begin{array}{cccccccc}
+10500 & -10500 & +10500 & 0000 & 0000 & 1000 & 0 & 0 & // 18 \ front \ right \ top \quad \text{blue}
\end{array}
\]

\[
\begin{array}{cccccccc}
+10500 & -10500 & -10500 & 0000 & 0000 & 1000 & 0 & 0 & // 19 \ front \ right \ bottom \quad \text{blue}
\end{array}
\]

\[
\begin{array}{cccccccc}
// \ \text{Rudder}
-10500 & 00000 & -10500 & 0000 & 0000 & 0000 & 0000 & 0 & // 20 \ back \ middle \ bottom \quad \text{black}
\end{array}
\]

\[
\begin{array}{cccccccc}
-10500 & 00000 & 10500 & 0000 & 0000 & 0000 & 0000 & 0 & // 21 \ back \ middle \ top \quad \text{black}
\end{array}
\]

\[
\begin{array}{cccccccc}
-15500 & 00000 & 10500 & 0000 & 0000 & 0000 & 1000 & 5000 & // 22 \ end \ middle \ top \quad \text{black}
\end{array}
\]

\[
\begin{array}{cccccccc}
-15500 & 00000 & -10500 & 0000 & 0000 & 0000 & 1000 & 5000 & // 23 \ end \ middle \ bottom \quad \text{black}
\end{array}
\]

\[
\begin{array}{cccccccc}
// \ \text{Elevator}
-10500 & +10500 & 00000 & 1000 & 1000 & 1000 & 0000 & 0 & // 24 \ back \ left \ middle \quad \text{white}
\end{array}
\]

\[
\begin{array}{cccccccc}
-10500 & -10500 & 00000 & 1000 & 1000 & 1000 & 0000 & 0 & // 25 \ back \ right \ middle \quad \text{white}
\end{array}
\]

\[
\begin{array}{cccccccc}
-15500 & -10500 & 00000 & 1000 & 1000 & 1000 & 2000 & 5000 & // 26 \ end \ right \ middle \quad \text{white}
\end{array}
\]

\[
\begin{array}{cccccccc}
-15500 & +10500 & 00000 & 1000 & 1000 & 1000 & 2000 & 5000 & // 27 \ end \ left \ middle \quad \text{white}
\end{array}
\]

\[
\begin{array}{cccccccc}
// \ \text{Left Aileron}
-10500 & +10500 & 00000 & 0300 & 0300 & 0300 & 0000 & 0 & // 28 \ back \ fleft \ middle \quad \text{gray}
\end{array}
\]

\[
\begin{array}{cccccccc}
-10500 & +15500 & 00000 & 0300 & 0300 & 0300 & 0000 & 0 & // 29 \ back \ fleft \ middle \quad \text{gray}
\end{array}
\]

\[
\begin{array}{cccccccc}
-15500 & +15500 & 00000 & 0300 & 0300 & 0300 & 3000 & 5000 & // 30 \ end \ fleft \ middle \quad \text{gray}
\end{array}
\]

\[
\begin{array}{cccccccc}
-15500 & +10500 & 00000 & 0300 & 0300 & 0300 & 3000 & 5000 & // 31 \ end \ left \ middle \quad \text{gray}
\end{array}
\]

\[
\begin{array}{cccccccc}
// \ \text{Right Aileron}
-10500 & -10500 & 00000 & 0300 & 0300 & 0300 & 0000 & 0 & // 32 \ back \ fright \ middle \quad \text{gray}
\end{array}
\]

\[
\begin{array}{cccccccc}
-10500 & -15500 & 00000 & 0300 & 0300 & 0300 & 0000 & 0 & // 33 \ back \ fright \ middle \quad \text{gray}
\end{array}
\]

\[
\begin{array}{cccccccc}
-15500 & -15500 & 00000 & 0300 & 0300 & 0300 & 4000 & 5000 & // 34 \ end \ fright \ middle \quad \text{gray}
\end{array}
\]

\[
\begin{array}{cccccccc}
-15500 & -10500 & 00000 & 0300 & 0300 & 0300 & 4000 & 5000 & // 35 \ end \ right \ middle \quad \text{gray}
\end{array}
\]

\[
\begin{array}{cccccccc}
i \quad // \ \text{Triangle indices always 16-bit unsigned integers}
0 & 1 & 2 & 0 & 2 & 3 & // \ \text{yellow base}
4 & 5 & 6 & 4 & 6 & 7 & // \ \text{green right side}
8 & 9 & 10 & 8 & 10 & 11 & // \ \text{blue top}
12 & 13 & 14 & 12 & 14 & 15 & // \ \text{red left side}
16 & 17 & 18 & 16 & 18 & 19 & // \ \text{blue forward face}
20 & 21 & 22 & 20 & 22 & 23 & // \ \text{Black rudder}
\end{array}
\]
24 25 26 24 26 27 // white elevator
28 29 30 28 30 31 // gray left aileron
32 33 34 32 34 35 // gray right aileron

The program also reads the vertex and fragment shader programs from file. These are called `gltstVshader.sdr` and `gltstFshader.sdr`. They are as follows.

```
uniform mat4 matrix; // Rotation and translation matrix
uniform mat4 control; // Control matrix

attribute vec4 g_vVertex;
attribute vec4 g_vColor;
attribute vec2 g_vData; // data[0]=ctrl data[1]=value

varying vec4 g_vVSColor;

void main()
{
  float ctrl = g_vData[0];

  g_vVSColor = g_vColor;

  // For fun, use the xyz coordinates to adjust the colour a little
  // g_vVSColor = g_vColor*0.9 + g_vVertex * 0.40;

  // Deal with the control surfaces

  if(ctrl > 0.0) {
    float dist = g_vData[1];

    vec4 Pos = g_vVertex;
    Pos.w = 1.0;

    if(ctrl==1.0) { // Rudder
      float cr = control[0][0];
      float sr = control[0][1];
      Pos.x += dist * (1.0-cr);
      Pos.y += dist * sr;
    }
    if(ctrl==2.0) { // Elevator
      float ce = control[0][2];
      float se = control[0][3];
      Pos.x += dist * (1.0 - ce);
      Pos.z += dist * se;
    }
  }
```
if(ctrl==3.0) { // Left aileron
float ca = control[1][0];
float sa = control[1][1];
Pos.x += dist * (1.0 - ca);
Pos.z += dist * sa;
}
if(ctrl==4.0) { // Right aileron
float ca = control[1][0];
float sa = control[1][1];
Pos.x += dist * (1.0 - ca);
Pos.z -= dist * sa;
}
// Rotate and translate the control surface
gl_Position = (matrix * Pos);

} else {

// Rotate and translate the model
gl_Position = (matrix * g_vVertex);
}

//#ifdef GL_ES
//precision mediump float;
//#endif

varying vec4 g_vVSColor;

void main()
{
gl_FragColor = g_vVSColor;
}

When gltiger is called it displays a rotating tigermoth a runway and some mountainous terrain. This image is composed of a large number of coloured triangles in 3D, giving a typical image such as the following.
The 3D triangles are specified in the file `tigermothmodel.mdl` whose structure is similar to that of `gltst.mdl` given above. It is convenient to generate `tigermothmodel.mdl` using a program (`mktigermothmodel.b`) whose is as follows.

```c
/*
This program creates the file tigermothmodel.mdl representing
a tiger moth aircraft in .mdl format for use by the OpenGL program
gltiger.b

Implemented by Martin Richards (c) February 2014

############# UNDER DEVELOPMENT ##################################

OpenGL vertex data is stored as follows

vec3 position -- t(direction of thrust), w(direction of left wing),
-- and l(direction of lift)
vec3 colour -- r, g, b
vec2 data  data[0] =1 rudder,
=2 elevator,
=3 left aileron,
=4 right aileron
=5 landscape and runway
data[1] = distance from hinge in inches, to be multiplied
by the sine or cosine of control surface angle
```
The program outputs vertex and index items representing the mode. It used a self extending vector for the vertices so that when vertices can be reused. Every value of vertex data is represented by scaled fixed point numbers with 3 digits after the decimal point.

In the .mdl language

s is followed by the scaling factor
v says the following values are vertex data
i say the following values are indices.
z marks the end of file
*/

GET "libhdr"

GLOBAL {
    stdin:ug
    stdout
    cur_r; cur_g; cur_b
    // If p is a self expanding array
    // p!0 = number of elements in the array
    // p!1 is current getvec’d vector for the array
    // p!2 is the upb of the current vector
    // push(p, x) will push a value into the array.
    // p!0=p!2 The array is expanded, typically double in size.
    ///push

    ///varray    // Self expanding array of vertices
    addvertex  // Find or create a vertex, returning the vertex number
    vertexcount // Index of the next vertex to be created
    hashtab   // hash table for vertices

    spacev
    spacep
    spacet
    newvec

    tracing
    tostream
}

MANIFEST {
    // Vertex structure
    v_x=0; v_y; v_z
6.4. A FIRST OPENGL EXAMPLE

v_r; v_g; v_b
v_k; v_d // Control surface, distance from hinge
v_n // Vertex number
v_chain // Hash chain
v_size // Number of words in a vertex node
v_upb = v_size-1

hashtabsiz = 541
hashtabupb = hashtabsiz-1

spaceupb = 500_000 * v_size

runwaylength = 600_000
runwaywidth = 40_000
landsize = 20_000_000
}

LET start() = VALOF
{ LET stdin = input()
  LET stdout = output()
  LET toname = "tigermothmodel.mdl"
  LET ht = VEC hashtabsiz
  LET argv = VEC 50
  ///LET vp, vv, vt = 0, 0, 0 // The vertex array self expanding array
  ///varray := @vp
  vertexcount := 0

  hashtab := ht
  FOR i = 0 TO hashtabupb DO hashtab!i := 0

  UNLESS rdargs("to/k,-t/s", argv, 50) DO
  { writef("Bad arguments for mktigermothmodel*n")
    RESULTIS 0
  }

  IF argv!0 DO toname := argv!0
  tracing := argv!1

  toostream := findoutput(toname)
  UNLESS toname DO
  { writef("trouble with file: %s*n", toname)
    RESULTIS 0
  }
spacev := getvec(spaceupb)
spacet := @spacev!spaceupb
spacep := spacet

UNLESS spacep DO
{ writef("Unable to allocate %n words of space*n")
  GOTO fin
}

colour(0,0,0)

selectoutput(tostream)

mktigermothmodel()

endstream(tostream)
selectoutput(stdout)
writef("Space used %n out of %n*n", spacet-spacep, spacet)

fin:
  IF spacev DO freevec(spacev)
  RESULTIS 0
}

AND newvec(upb) = VALOF
{ LET p = spacep - upb - 1
  IF p < spacev DO
  { writef("error: spacev is not large enough*n")
    abort(999)
  }
  spacep := p
  RESULTIS p
}

AND colour(r, g, b) BE
  cur_r, cur_g, cur_b := 1_000*r/255, 1_000*g/255, 1_000*b/255

AND findvertex(t,w,l, r,g,b, k,d) = VALOF
{ // Return the pointer to the matching vertex node,
  // creating one if necessary.
  // t,w,h, etc are floating point numbers but the hash
  // computation just treats them as bit patterns to
  // produce a hash value in the range 0 to hashtabupb.
  LET hashval = ((t+w+l+r+g+b+k+d)>>1) MOD hashtabsize
  LET p = hashtab!hashval
WHILE p DO // Search down the hash chain
{ IF p!v_x=t & p!v_y=w & p!v_z=l &
  p!v_r=r & p!v_g=g & p!v_b=b &
  p!v_k=k & p!v_d=d RESULTIS p // Vertex found
  p := p!v_chain
}
// Vertex not found
p := newvec(v_upb)
p!v_x, p!v_y, p!v_z := t, w, l
p!v_r, p!v_g, p!v_b := r, g, b
p!v_k, p!v_d := k, d
p!v_n := vertexcount
p!v_chain := hashtab!hashval
hashtab!hashval := p
writef("v %i6 %i6 %i6 %i4 %i4 %i4 %i4 %i6 // %i3*n",
  t, w, l, r, g, b, 1000*k, d, vertexcount)
vertexcount := vertexcount+1
RESULTIS p
}

AND addvertex(t, w, l, k, d) = findvertex(t, w, l, cur_r, cur_g, cur_b, k, d)

AND addlandvertex(n, w, h, r, g, b) = VALOF
{ colour(r, g, b)
  RESULTIS addvertex(n, w, h, 5, 0)
}

AND triangle(a, b, c, d, e, f, g, h, i) BE
{ // a, b, c are in directions forward, left and up
  // store as openGL t,w,l which are forward, left, up.
  // ie set t, w, l to a, b, c
  // do the same for def and ghi
  LET v0 = addvertex(a, b, c, 0, 0)!v_n
  LET v1 = addvertex(d, e, f, 0, 0)!v_n
  LET v2 = addvertex(g, h, i, 0, 0)!v_n
  writef("i %i4 %i4 %i4*n", v0, v1, v2)
}

AND quad(a, b, c, d, e, f, g, h, i, j, k, l) BE
{ // a, b, c are in directions forward, left and up
  // store as openGL t,w,l which are forward,left, up.
  // ie set x, y, z to a, b, c
  // do the same for def, ghi and jkl
  LET v0 = addvertex(a, b, c, 0, 0)!v_n
  LET v1 = addvertex(d, e, f, 0, 0)!v_n

LET v2 = addvertex(g,h,i, 0, 0)!v_n
LET v3 = addvertex(j,k,l, 0, 0)!v_n
writef("i %i4 %i4 %i4*n", v0, v1, v2)
writef("i %i4 %i4 %i4*n", v0, v2, v3)

AND quadkd(a,b,c,k1,d1, d,e,f,k2,d2, g,h,i,k3,d3, j,k,l,k4,d4) BE
{ // a, b, c are in directions forward, left and up
 // store as openGL t,w,l which are forward, left, up
 // ie set x, y, z to a, b, c
 // do the same for def, ghi and jkl
LET v0 = addvertex(a,b,c, k1, d1)!v_n
LET v1 = addvertex(d,e,f, k2, d2)!v_n
LET v2 = addvertex(g,h,i, k3, d3)!v_n
LET v3 = addvertex(j,k,l, k4, d4)!v_n
writef("i %i4 %i4 %i4*n", v0, v1, v2)
writef("i %i4 %i4 %i4*n", v0, v2, v3)
}

AND quadland(x1,y1,z1, r1,g1,b1,
 x2,y2,z2, r2,g2,b2,
 x3,y3,z3, r3,g3,b3,
 x4,y4,z4, r4,g4,b4) BE
{ // 3D coords and colours of the the vertices of a quad
 // of landscape or runway
LET v0 = addlandvertex(x1,y1,z1, r1,g1,b1)!v_n
LET v1 = addlandvertex(x2,y2,z2, r2,g2,b2)!v_n
LET v2 = addlandvertex(x3,y3,z3, r3,g3,b3)!v_n
LET v3 = addlandvertex(x4,y4,z4, r4,g4,b4)!v_n
writef("i %i4 %i4 %i4*n", v0, v1, v2)
writef("i %i4 %i4 %i4*n", v0, v2, v3)
}

AND mktigermothmodel() BE
{ // The origin is the centre of gravity of the tigermoth
 // For landsacpe and the runway, the origin is the start of the runway

 // The tigermoth coordinates are as follows

 // first  t is the distance forward of the centre of gravity
 // second  w is the distance left of the centre of gravity
 // third  l is the distance above the centre of gravity

writef("// Tiger Moth Model*n")
newline()
writef("// The v parameters are\n")
writef("//  t  w  l  r  g  b  k  d\n")
newline()
writef("// ie t = direction of thrust\n")
writef("//  w = direction of left wing\n")
writef("//  l = direction of lift\n")
newline()
writef("//  k = 0 fixed surface\n")
writef("//  k = 1 rudder\n")
writef("//  k = 2 elevator\n")
writef("//  k = 3 left aileron\n")
writef("//  k = 4 right aileron\n")
writef("//  k = 5 landscape and runway\n")
newline()
writef("s 1000\n")

writef("// Cockpit floor\n")
colour(90,80,30)
quad( 1_000, 0_800, 0_000,
    1_000,-0_800, 0_000,
    -5_800,-0_800, 0_000,
    -5_800, 0_800, 0_000)

writef("// Left lower wing\n")
colour(165,165,30)  // Under surface
   // -t  w  l
quad(-0_500, 1_000, -2_000,  // Panel A
    -3_767, 1_000, -2_218,
    -4_396, 6_000, -1_745,
    -1_129, 6_000, -1_527)

colour(155,155,20)  // Under surface
quadkd(-4_396, 6_000, -1_745, 0, 0, // Panel D left Aileron
    -5_546, 6_000, -1_821, 3, 1_150,
    -6_297, 13_766, -1_255, 3, 1_150,
    -5_147, 14_166, -1_179, 0, 0)

colour(155,155,60)
//colour(255,155,60)
quad(-3_767, 1_000, -2_218,  // Panel B
    -4_917, 1_000, -2_294,
    -5_546, 6_000, -1_821,
    -4_396, 6_000, -1_745)
colour(155,155,90)
quad(-1.129, 6.000, -1.527, // Panel C
   -4.396, 6.000, -1.745,
   -5.147, 14.166, -1.179,
   -1.880, 14.166, -0.961)
writef("// Left lower wing upper surface\n")
colour(120,140,60)
quad(-0.500, 1.000, -2.000, // Panel A1
   -1.500, 1.000, -1.800,
   -2.129, 6.000, -1.327,
   -1.129, 6.000, -1.527)
colour(120,130,50)
quad(-1.500, 1.000, -1.800, // Panel A2
   -3.767, 1.000, -2.118,
   -4.396, 6.000, -1.645,
   -2.129, 6.000, -1.327)
quad(-3.767, 1.000, -2.118, // Panel B
   -4.917, 1.000, -2.294,
   -5.546, 6.000, -1.821,
   -4.396, 6.000, -1.645)
colour(120,140,60)
quad(-1.129, 6.000, -1.527, // Panel C1
   -2.129, 6.000, -1.327,
   -2.880, 14.166, -0.761,
   -1.880, 14.166, -0.961)
colour(120,130,50)
quad(-2.129, 6.000, -1.327, // Panel C2
   -4.396, 6.000, -1.645,
   -5.147, 14.166, -1.079,
   -2.880, 14.166, -0.761)
colour(120,140,60)
quadkd(-4.396, 6.000, -1.645, 0, 0, // Panel D Aileron
   -5.546, 6.000, -1.821, 3, 1.150,
   -6.297, 13.766, -1.255, 3, 1.150,
   -5.147, 14.166, -1.079, 0, 0)
writef("// Left lower wing tip\n")
colour(130,150,60)
triangle(-1.880, 14.167, -1.006, 
-2.880, 14.167, -0.761, 
-3.880, 14.467, -0.980) 
colour(130,150,60) 
triangle(-2.880, 14.167, -0.761, 
-5.147, 14.167, -1.079, 
-3.880, 14.467, -0.980) 
colour(160,160,40) 
triangle(-5.147, 14.167, -1.079, 
-5.147, 14.167, -1.179, 
-3.880, 14.467, -0.980) 
colour(170,170,50) 
triangle(-5.147, 14.167, -1.179, 
-1.880, 14.167, -0.961, 
-3.880, 14.467, -0.980) 

colour(165,165,30) // Under surface 
quad(-0.500, -1.000, -2.000, // Panel A 
-3.767, -1.000, -2.218, 
-4.396, -6.000, -1.745, 
-1.129, -6.000, -1.527) 

quad(-3.767, -1.000, -2.218, // Panel B 
-4.917, -1.000, -2.294, 
-5.546, -6.000, -1.821, 
-4.396, -6.000, -1.745) 

quad(-1.129, -6.000, -1.527, // Panel C 
-4.396, -6.000, -1.745, 
-5.147,-14.166, -1.179, 
-1.880,-14.166, -0.961) 

colour(155,155,20) // Under surface 
quad(-4.396, -6.000, -1.745, 0, // Panel D Aileron 
-5.546, -6.000, -1.821, 4, 1.150, 
-6.297,-13.766, -1.255, 4, 1.150, 
-5.147,-14.166, -1.179, 0, 0) 

writef("// Right lower wing upper surface\n") 
colour(120,140,60) 
quad(-0.500, -1.000, -2.000, // Panel A1 
-1.500, -1.000, -1.800,
CHAPTER 6. INTERACTIVE GRAPHICS IN BCPL USING OPENGL

\[-2.129, -6.000, -1.327,\]
\[-1.129, -6.000, -1.527]\)

\[colour(120,130,50)\]
quad(-1_500, -1_000, -1_800, // Panel A2
-3_767, -1_000, -2_118,
-4_396, -6_000, -1_645,
-2_129, -6_000, -1_327)

quad(-3_767, -1_000, -2_118, // Panel B
-4_917, -1_000, -2_294,
-5_546, -6_000, -1_821,
-4_396, -6_000, -1_645)

\[colour(120,140,60)\]
quad(-1_129, -6_000, -1_527, // Panel C1
-2_129, -6_000, -1_327,
-2_880,-14_166, -0_761,
-1_880,-14_166, -0_961)

\[colour(120,130,50)\]
quad(-2_129, -6_000, -1_327, // Panel C2
-4_396, -6_000, -1_645,
-5_147,-14_166, -1_079,
-2_880,-14_166, -0_761)

\[colour(120,140,60)\]
quad(-4_396, -6_000, -1_645, 0, 0, // Panel D Aileron
-5_546, -6_000, -1_821, 4, 1_150,
-6_297,-13_766, -1_255, 4, 1_150,
-5_147,-14_166, -1_079, 0, 0)

\[writef("// Right lower wing tip\n")\]
\[colour(130,150,60)\]
triangle(-1_880,-14_167,-1_006,
-2_880,-14_167,-0_761,
-3_880,-14_467,-0_980)
\[colour(130,150,60)\]
triangle(-2_880,-14_167,-0_761,
-5_147,-14_167,-1_079,
-3_880,-14_467,-0_980)
\[colour(160,160,40)\]
triangle(-5_147,-14_167,-1_079,
-5_147,-14_167,-1_179,
-3_880,-14_467,-0_980)
6.4. A FIRST OPENGL EXAMPLE

colour(170,170,50)
triangle(-5_147,-14_167,-1_179,
-1_880,-14_167,-0_961,
-3_880,-14_467,-0_980)

writf(" // Left upper wing*n")
colour(200,200,30) // Under surface
quad( 1_333, 1_000, 2_900,
-1_967, 1_000, 2_671,
-3_297, 14_167, 3_671,
0_003, 14_167, 3_894)
quad(-1_967, 1_000, 2_671,
-3_084, 2_200, 2_606,
-4_414, 13_767, 3_645,
-3_297, 14_167, 3_894)

colour(150,170,90) // Top surface
quad( 1_333, 1_000, 2_900, // Panel A1
0_333, 1_000, 3_100,
-0_997, 14_167, 4_094,
0_003, 14_167, 3_894)

colour(140,160,80) // Top surface
quad( 0_333, 1_000, 3_100, // Panel A2
-1_967, 1_000, 2_771,
-3_297, 14_167, 3_771,
-0_997, 14_167, 4_094)

colour(150,170,90) // Top surface
quad(-1_967, 1_000, 2_771, // Panel B
-3_084, 2_200, 2_606,
-4_414, 13_767, 3_645,
-3_297, 14_167, 3_771)

writf(" // Left upper wing tip*n")
colour(130,150,60)
triangle( 0_003, 14_167, 3_894,
-0_997, 14_167, 4_094,
-1_997, 14_467, 3_874)
colour(130,150,60)
triangle(-0_997, 14_167, 4_094,
-3_297, 14_167, 3_771,
-1_997, 14_467, 3_874)
colour(160,160,40)
triangle(-3_297, 14_167, 3_771,
```
-3.297, 14_167, 3.671,
-1.997, 14_467, 3.874)
colour(170,170,50)
triangle(-3.297, 14_167, 3.671,
    0.003, 14_167, 3.894,
-1.997, 14_467, 3.874)

writef("// Right upper wing*n")
colour(200,200,30) // Under surface
quad( 1.333, -1.000, 2.900,
      -1.967, -1.000, 2.671,
      -3.297,-14_167, 3.671,
      0.003,-14_167, 3.894)
quad(-1.967, -1.000, 2.671,
      -3.084, -2.200, 2.606,
      -4.414,-13_767, 3.645,
      -3.297,-14_167, 3.671)

colour(150,170,90) // Top surface
quad( 1.333, -1.000, 2.900, // Panel A1
      0.333, -1.000, 3.100,
      -0.997,-14_167, 4.094,
      0.003,-14_167, 3.894)

colour(140,160,80) // Top surface
quad( 0.333, -1.000, 3.100, // Panel A2
      -1.967, -1.000, 2.771,
      -3.297,-14_167, 3.771,
      -0.997,-14_167, 4.094)

colour(150,170,90) // Top surface
quad(-1.967, -1.000, 2.771, // Panel B
    -3.084, -2.200, 2.606,
    -4.414,-13_767, 3.645,
    -3.297,-14_167, 3.771)

writef("// Right upper wing tip*n")
colour(130,150,60)
triangle( 0.003,-14_167, 3.894,
      -0.997,-14_167, 4.094,
      -1.997,-14_467, 3.874)
colour(130,150,60)
triangle(-0.997,-14_167, 4.094,
    -3.297,-14_167, 3.771,
```
6.4. A FIRST OPENGL EXAMPLE

```
-1.997, -14.467, 3.874)
colour(160,160,40)
triangle(-3.297, -14.167, 3.771,
        -3.297, -14.167, 3.671,
        -1.997, -14.467, 3.874)
colour(170,170,50)
triangle(-3.297, -14.167, 3.671,
        0.003, -14.167, 3.894,
        -1.997, -14.467, 3.874)

writef(" // Wing root strut forward left*n")
colour(80,80,80)
//quad( 0.433, 0.950, 2.900,
      //      0.633, 0.950, 2.900,
      //      0.633, 1.000, 0,
      //      0.433, 1.000, 0)
strut(0.433, 0.950, 2.900,
      0.433, 1.000, 0)

writef(" // Wing root strut rear left*n")
colour(80,80,80)
//quad( -1.967, 0.950, 2.616,
       //      -1.767, 0.950, 2.616,
       //      -0.868, 1.000, 0,
       //      -1.068, 1.000, 0)
strut(-1.967, 0.950, 2.616,
     -1.068, 1.000, 0)

writef("// Wing root strut diag left*n")
colour(80,80,80)
//quad( 0.433, 0.950, 2.900,
       //      0.633, 0.950, 2.900,
       //      -0.868, 1.000, 0,
       //      -1.068, 1.000, 0)
strut(0.433, 0.950, 2.900,
      -1.068, 1.000, 0)

writef("// Wing root strut forward right*n")
colour(80,80,80)
//quad( 0.433, -0.950, 2.900,
       //      0.633, -0.950, 2.900,
       //      0.633, -1.000, 0,
       //      0.433, -1.000, 0)
strut(0.433, -0.950, 2.900,
      0.433, -1.000, 0)
```
CHAPTER 6. INTERACTIVE GRAPHICS IN BCPL USING OPENGL

0_433, -1_000, 0)

writef(" // Wing root strut rear right*n")
colour(80,80,80)
//quad( -1_967, -0_950, 2_616,
// -1_767, -0_950, 2_616,
// -0_868, -1_000, 0,
// -1_068, -1_000, 0)
strut(-1_967, -0_950, 2_616,
-1_068, -1_000, 0)

writef("// Wing root strut diag right*n")
colour(80,80,80)
//quad( 0_433, -0_950, 2_900,
// 0_633, -0_950, 2_900,
// -0_868, -1_000, 0,
// -1_068, -1_000, 0)
strut( 0_433, -0_950, 2_900,
-1_068, -1_000, 0)

writef("// Wing strut forward left*n")
colour(80,80,80)
//quad( -2_200, 10_000, -1_120,
// -2_450, 10_000, -1_120,
// -0_550, 10_000, 3_315,
// -0_300, 10_000, 3_315)
strut(-2_200, 10_000, -1_120,
-0_300, 10_000, 3_445)

writef("// Wing strut rear left*n")
colour(80,80,80)
//quad( -4_500, 10_000, -1_260,
// -4_750, 10_000, -1_260,
// -2_850, 10_000, 3_210,
// -2_500, 10_000, 3_210)
strut(-4_500, 10_000, -1_260,
-2_500, 10_000, 3_410)

writef("// Wing strut forward right*n")
colour(80,80,80)
//quad( -2_200, -10_000, -1_120,
// -2_450, -10_000, -1_120,
// -0_550, -10_000, 3_445,
// -0_300, -10_000, 3_445)
strut(-2_200, -10_000, -1_120,
-0.300, -10.000, 3.445)

\text{writef}("// Wing strut rear right*n")
\text{colour}(80,80,80)
//quad( -4.500, -10.000, -1.260,
//  -4.750, -10.000, -1.260,
//  -2.850, -10.000, 3.210,
//  -2.500, -10.000, 3.210)
\text{strut}(-4.500, -10.000, -1.260,
 -2.500, -10.000, 3.410)

\text{writef}("// Wheel strut left*n")
\text{colour}(80,80,80)
//quad( -0.768, 1.000, -2.000,
//  -1.168, 1.000, -2.000,
//  -0.468, 2.000, -3.800,
//  -0.068, 2.000, -3.800)
\text{strut}(-0.768, 1.000, -2.000,
-0.068, 2.000, -3.800)

\text{writef}(" // Wheel strut diag left*n")
\text{colour}(80,80,80)
//quad( 1.600, 1.000, -2.000,
//  1.800, 1.000, -2.000,
//  -0.368, 2.000, -3.800,
//  -0.168, 2.000, -3.800)
\text{strut}(1.600, 1.000, -2.000,
-0.168, 2.000, -3.800)

\text{writef}("// Wheel strut centre left*n")
\text{colour}(80,80,80)
//quad( -0.500, 0.000, -2.900,
//  -0.650, 0.000, -2.900,
//  -0.318, 2.000, -3.800,
//  -0.168, 2.000, -3.800)
\text{strut}(-0.500, 0.000, -2.900,
-0.168, 2.000, -3.800)

\text{writef}("// Wheel strut right*n")
\text{colour}(80,80,80)
//quad( -0.768, -1.000, -2.000,
//  -1.168, -1.000, -2.000,
//  -0.468, -2.000, -3.800,
//  -0.068, -2.000, -3.800)
\text{strut}(-0.768, -1.000, -2.000,
\texttt{writef("// Wheel strut diag right\*n")}
colour(80,80,80)
\texttt{//quad( 1.600, -1.000, -2.000,}
   \texttt{// 1.800, -1.000, -2.000,}
   \texttt{// -0.368, -2.000, -3.800,}
   \texttt{// -0.168, -2.000, -3.800)}
\texttt{strut( 1.600, -1.000, -2.000,}
   \texttt{-0.168, -2.000, -3.800)}

\texttt{writef("// Wheel strut centre right\*n")}
colour(80,80,80)
\texttt{//quad( -0.500, -0.000, -2.900,}
   \texttt{// -0.650, -0.000, -2.900,}
   \texttt{// -0.318, -2.000, -3.800,}
   \texttt{// -0.168, -2.000, -3.800)}
\texttt{strut(-0.500, -0.000, -2.900,}
   \texttt{-0.168, -2.000, -3.800)}

\texttt{writef("// Left wheel\*n")}
colour(20,20,20)
\texttt{quad( -0.268, 2.000, -3.800,}
   \texttt{-0.268, 2.100, -3.800-0.700,}
   \texttt{-0.268-0.500, 2.100, -3.800-0.500,}
   \texttt{-0.268-0.700, 2.100, -3.800)}
\texttt{quad( -0.268, 2.000, -3.800,}
   \texttt{-0.268, 2.100, -3.800-0.700,}
   \texttt{-0.268+0.500, 2.100, -3.800-0.500,}
   \texttt{-0.268+0.700, 2.100, -3.800)}
\texttt{quad( -0.268, 2.000, -3.800,}
   \texttt{-0.268, 2.100, -3.800+0.700,}
   \texttt{-0.268-0.500, 2.100, -3.800+0.500,}
   \texttt{-0.268-0.700, 2.100, -3.800)}
\texttt{quad( -0.268, 2.000, -3.800,}
   \texttt{-0.268, 2.100, -3.800+0.700,}
   \texttt{-0.268+0.500, 2.100, -3.800+0.500,}
   \texttt{-0.268+0.700, 2.100, -3.800)}
\texttt{quad( -0.268, 2.200, -3.800,}
   \texttt{-0.268, 2.100, -3.800-0.700,}
   \texttt{-0.268-0.500, 2.100, -3.800-0.500,}
   \texttt{-0.268-0.700, 2.100, -3.800)}
\texttt{quad( -0.268, 2.200, -3.800,
6.4. A FIRST OPENGL EXAMPLE

\begin{verbatim}
-0.268, 2.100, -3.800-0.700,
-0.268+0.500, 2.100, -3.800-0.500,
-0.268+0.700, 2.100, -3.800

quad(-0.268, 2.200, -3.800,
-0.268, 2.100, -3.800+0.700,
-0.268-0.500, 2.100, -3.800+0.500,
-0.268-0.700, 2.100, -3.800)

quad(-0.268, 2.100, -3.800,
-0.268+0.500, 2.100, -3.800+0.500,
-0.268+0.700, 2.100, -3.800)

writef("// Right wheel*n")

colour(20,20,20)

quad(-0.268, -2.000, -3.800,
-0.268, -2.100, -3.800-0.700,
-0.268-0.500,-2.100, -3.800-0.500,
-0.268-0.700,-2.100, -3.800)

quad(-0.268, -2.000, -3.800,
-0.268, -2.100, -3.800-0.700,
-0.268+0.500,-2.100, -3.800-0.500,
-0.268+0.700,-2.100, -3.800)

quad(-0.268, -2.000, -3.800,
-0.268, -2.100, -3.800+0.700,
-0.268-0.500,-2.100, -3.800+0.500,
-0.268-0.700,-2.100, -3.800)

quad(-0.268, -2.000, -3.800,
-0.268, -2.100, -3.800+0.700,
-0.268+0.500,-2.100, -3.800+0.500,
-0.268+0.700,-2.100, -3.800)

quad(-0.268, -2.000, -3.800,
-0.268, -2.100, -3.800+0.700,
-0.268-0.500,-2.100, -3.800+0.500,
-0.268-0.700,-2.100, -3.800)

quad(-0.268, -2.100, -3.800+0.700,
\end{verbatim}
-0.268+0.500,-2.100, -3.800+0.500,
-0.268+0.700,-2.100, -3.800)

writef("// Fueltank front*n")
colour(200,200,230) // Top surface
quad( 1_333, 1_000, 2_900,
    1_333, -1_000, 2_900,
    0_033, -1_000, 3_100,
    0_033, 1_000, 3_100)

writef("// Fueltank back*n")
colour(180,180,210) // Top surface
quad( 0_033, 1_000, 3_100,
    0_033, -1_000, 3_100,
    -1.967, -1_000, 2_616,
    -1.967, 1_000, 2_616)

writef("// Fueltank left side*n")
colour(160,160,190)
triangle( 1_333, 1_000, 2_900,
        0_033, 1_000, 3_100,
        -1.967, 1_000, 2_616)

writef("// Fueltank right side*n")
colour(160,160,190)
triangle(-0.500+1_833, -1_000, -2.000+4_900,
        -1.800+1_833, -1_000, -1.800+4_900,
        -3.800+1_833, -1_000, -2.284+4_900)

writef("// Fuselage*n")

writef("// Prop shaft*n")
colour(40,40,90)
triangle( 5_500, 0, 0,
        4_700, 0_200, 0_300,
        4_700, 0_200,-0_300)
colour(60,60,40)
triangle( 5_500, 0, 0,
        4_700, 0_200,-0_300,
        4_700,-0_200,-0_300)
colour(40,40,90)
triangle( 5_500, 0, 0,
        4_700,-0_200,-0_300,
        4_700,-0_200, 0_300)
colour(60,60,40)
6.4. A FIRST OPENGL EXAMPLE

triangle( 5_500, 0, 0, 
        4_700,-0_200, 0_300, 
        4_700, 0_200, 0_300)

writef("// Engine front lower centre*n")
colour(140,140,160)
triangle( 5_000, 0, 0, 
        4_500, 0_350, -1_750, 
        4_500,-0_350, -1_750)

writef("// Engine front lower left*n")
colour(140,120,130)
triangle( 5_000, 0, 0, 
        4_500, 0_350, -1_750, 
        4_500, 0_550, 0)
writef("// Engine front lower right*n")
colour(140,120,130)
triangle( 5_000, 0, 0, 
        4_500,-0_350, -1_750, 
        4_500,-0_550, 0)

writef("// Engine front upper centre*n")
colour(140,140,160)
triangle( 5_000, 0, 0, 
        4_500, 0_350, 0_500, 
        4_500,-0_350, 0_500)

writef("// Engine front upper left and right*n")
colour(100,140,180)
triangle( 5_000, 0, 0, 
        4_500, 0_350, 0_500, 
        4_500, 0_550, 0)
triangle( 5_000, 0, 0, 
        4_500,-0_350, 0_500, 
        4_500,-0_550, 0)

writef("// Engine left lower*n")
colour(80,80,60)
quad( 1_033, 1_000, 0, 
      1_800, 1_000, -2_000, 
      4_500, 0_350, -1_750, 
      4_500, 0_550, 0)
writef(" // Engine right lower*n")
colour(80,100,60)
quad( _033,-_000, 0,
   _800,-_000, -2_000,
   4_500,-0_350, -1_750,
   4_500,-0_550, 0)

writef(" // Engine top left*n")
colour(100,130,60)
quad( _033, 0_750, 0_950,
     _033, 1_000, 0_000,
     4_500, 0_550, 0_000,
     4_500, 0_350, 0_500)

writef(" // Engine top centre*n")
colour(130,160,90)
quad( _033, 0_750, 0_950,
     _033,-0_750, 0_950,
     4_500,-0_350, 0_500,
     4_500, 0_350, 0_500)

writef(" // Engine top right*n")
colour(100,130,60)
quad( _033,-0_750, 0_950,
     _033,-1_000, 0_000,
     4_500,-0_550, 0_000,
     4_500,-0_350, 0_500)

writef(" // Engine bottom*n")
colour(100,80,50)
quad( 4_500, 0_350, -1_750,
     4_500,-0_350, -1_750,
     1_800,-1_000, -2_000,
     1_800, 1_000, -2_000)

writef(" // Front cockpit left*n")
colour(120,140,60)
quad( -2_000, 1_000, 0_000,
     -2_000, 0_853, 0_600,
     -3_300, 0_853, 0_600,
     -3_300, 1_000, 0_000)

writef(" // Front cockpit right*n")
colour(120,140,60)
quad( -2.000,-1.000, 0.000,
-2.000,-0.853, 0.600,
-3.300,-0.853, 0.600,
-3.300,-1.000, 0.000)

writef("// Top front left*n")
colour(100,120,40)
quad( 1.033, 0.750, 0.950,
-2.000, 0.750, 1.000,
-2.000, 1.000, 0.000,
1.033, 1.000, 0.000)

writef("// Top front middle*n")
colour(120,140,60)
quad( 1.033, 0.750, 0.950,
1.033,-0.750, 0.950,
-2.000,-0.750, 1.000,
-2.000, 0.750, 1.000)

writef("// Top front right*n")
colour(100,120,40)
quad( 1.033,-0.750, 0.950,
-2.000,-0.750, 1.000,
-2.000,-1.000, 0.000,
1.033,-1.000, 0.000)

writef(" // Front wind shield*n")
colour(180,200,150)
quad( -1.300, 0.450, 1.000, // Centre
-2.000, 0.450, 1.400,
-2.000,-0.450, 1.400,
-1.300,-0.450, 1.000)
colour(220,220,180)
triangle( -1.300, 0.450, 1.000, // Left
-2.000, 0.450, 1.400,
-2.000, 0.650, 1.000)
triangle( -1.300,-0.450, 1.000, // Right
-2.000,-0.450, 1.400,
-2.000,-0.650, 1.000)

writef("// Top left middle*n")
colour(120,165,90)
quad( -3.300, 0.750, 1.000,
CHAPTER 6. INTERACTIVE GRAPHICS IN BCPL USING OPENGL

-3_300, 1_000, 0_000,
-4_300, 1_000, 0_000,
-4_300, 0_750, 1_000)

writef("// Top centre middle\n")
colour(120,140,60)
quad( -3_300, 0_750, 1_000,
    -3_300,-0_750, 1_000,
    -4_300,-0_750, 1_000,
    -4_300, 0_750, 1_000)

writef("// Top right middle\n")
colour(130,160,90)
quad( -3_300,-0_750, 1_000,
    -3_300,-1_000, 0_000,
    -4_300,-1_000, 0_000,
    -4_300,-0_750, 1_000)

writef("// Rear cockpit left\n")
colour(120,140,60)
quad( -4_300, 1_000, 0_000,
    -4_300, 0_840, 0_600,
    -5_583, 0_770, 0_600,
    -5_583, 1_000, 0_000)

writef("// Rear wind shield\n")
colour(180,200,150)
quad( -3_600, 0_450, 1_000, // Centre
    -4_300, 0_450, 1_400,
    -4_300,-0_450, 1_400,
    -3_600,-0_450, 1_000)
colour(220,220,180)
triangle( -3_600, 0_450, 1_000, // Left
     -4_300, 0_450, 1_400,
     -4_300, 0_650, 1_000)
triangle( -3_600,-0_450, 1_000, // Right
     -4_300,-0_450, 1_400,
     -4_300,-0_650, 1_000)

writef("// Rear cockpit right\n")
colour(110,140,70)
quad( -4_300,-1_000, 0_000,
    -4_300,-0_840, 0_600,
6.4. A FIRST OPENGL EXAMPLE

-5.583, -0.770, 0.600,
-5.583, -1.000, 0.000)

writef("// Lower left middle*n")
colour(140,110,70)
quad( 1.033, 1.000, 0,
    1.800, 1.000, -2.000,
    -3.583, 1.000, -2.238,
    -3.300, 1.000, 0)

colour(155,100,70)
triangle( -3.300, 1.000, 0,
    -3.583, 1.000, -2.238,
    -5.583, 1.000, 0)

writef("// Bottom middle*n")
colour(120,100,60)
quad( 1.800, 1.000, -2.000,
    -3.583, 1.000, -2.238,
    -3.583, -1.000, -2.238,
    -3.300, -1.000, -2.000)

colour(120,100,70)
triangle( -3.300, -1.000, 0,
    -3.583, -1.000, -2.238,
    -5.583, -1.000, 0)

writef("// Lower right middle*n")
colour(140,100,70)
quad( 1.033, -1.000, 0,
    1.800, -1.000, -2.000,
    -3.583, -1.000, -2.238,
    -3.300, -1.000, 0)

colour(120,100,70)
triangle( -3.300, -1.000, 0,
    -3.583, -1.000, -2.238,
    -5.583, -1.000, 0)

writef("// Lower left back*n")
colour(165,115,80)
quad( -5.583, 1.000, 0,
    -16.000, 0.050, 0,
    -16.000, 0.050, -0.667,
    -3.583, 1.000, -2.238)

writef("// Bottom back*n")
colour(130,90,60)
quad( -3.583, 1.000, -2.238,
    -16.000, 0.050, -0.667,
    -16.000, -0.050, -0.667,
writef("// Lower right back\n")
colour(150,140,80)
quad( -5_583,-1_000, 0,
    -16_000,-0_050, 0,
    -16_000,-0_050, -0_667,
    -3_583,-1_000, -2_238)

writef("// Top left back\n")
colour(130,125,85)
triangle( -5_583, 0_650, 0_950,
    -5_583, 1_000, 0_000,
    -16_000, 0_050, 0)

writef("// Top centre back\n")
colour(130,160,90)
quad( -5_583, 0_650, 0_950,
    -5_583,-0_650, 0_950,
    -16_000,-0_050, 0,
    -16_000, 0_050, 0)

writef("// Top right back\n")
colour(130,120,80)
triangle( -5_583,-0_650, 0_950,
    -5_583,-1_000, 0_000,
    -16_000,-0_050, 0)

writef("// End back\n")
colour(120,165,95)
quad(-16_000, 0_050, 0,
    -16_000,-0_050, 0,
    -16_000,-0_050, -0_667,
    -16_000, 0_050, -0_667)

writef("// Fin\n")

colour(170,180,80)
quad(-14_000, 0_000, 0,  // Fin
    -16_000, 0_050, 0,
    -16_000, 0_100, 1_000,
    -15_200, 0_000, 1_000)
quad(-14_000, 0_000, 0,  // Fin
    -16_000,-0_050, 0,
    -16_000,-0_100, 1_000,
6.4. A FIRST OPENGL EXAMPLE

\[
\text{colour}(70,120,40) \\
\text{quadkd}(-15,200, 0, 1, 0, -0.8, // Rudder} \\
-16,000, 100, 1, 0, 0, \\
-16,800, 0, 3, 100, 1, 0.8, \\
-16,000, 0, 2.55, 0, 0) \\
\text{colour}(70,125,30) \\
\text{quadkd}(-15,200, 0, 1, 0, -0.8, // Rudder} \\
-16,000, -100, 1, 0, 0, \\
-16,800, 0, 3, 100, 1, 0.8, \\
-16,000, 0, 2.55, 0, 0) \\
\text{colour}(70, 80, 40) \\
\text{quadkd}(-16,000, 100, 1, 0, 0, \\
-16,800, 0, 3, 100, 1, 0.8, \\
-17,566, 0, 2, 600, 1, 1.566, \\
-17,816, 0, 1, 667, 1, 1.816) \\
\text{quadkd}(-16,000, -100, 1, 0, 0, \\
-16,800, 0, 3, 100, 1, 0.8, \\
-17,566, 0, 2, 600, 1, 1.566, \\
-17,816, 0, 1, 667, 1, 1.866) \\
\text{colour}(70,120,40) \\
\text{quadkd}(-16,000, 100, 1, 0, 0, \\
-17,816, 0, 1, 667, 1, 1.816, \\
-17,816, 0, 1, 000, 1, 1.816, \\
-17,566, 0, 0, 1, 1.566) \\
\text{quadkd}(-16,000, -100, 1, 0, 0, \\
-17,816, 0, 1, 667, 1, 1.816, \\
-17,816, 0, 1, 000, 1, 1.816, \\
-17,566, 0, 0, 1, 1.566) \\
\text{colour}(70, 80, 40) \\
\text{quadkd}(-16,000, 100, 1, 0, 0, \\
-17,566, 0, 0, 1, 1.566, \\
-17,000, 0,-0.583, 1, 1.000, \\
-16,000, 0,-0.667, 0, 0) \\
\text{quadkd}(-16,000, -100, 1, 0, 0, \\
-17,566, 0, 0, 1, 1.566, \\
-17,000, 0,-0.583, 1, 1.000, \\
-16,000, 0,-0.667, 0, 0)
\]

\text{writef}// Tail skid\n
\text{colour}(40, 40, 40) \\
\text{quadkd}(-16,000, 0, -0.667, 0, 0, \\
-16,200, 0, -0.667, 1, 0.200, \\
-16,500, 0, -0.900, 1, 0.500,
writef("// Tailplane and elevator\n")

colour(120,180,50)
triangle(-16_000, 0_000, 100,
-13_900, 0_600, 0,
-13_900,-0_600, 0)
triangle(-16_000, 0_000,-100,
-13_900, 0_600, 0,
-13_900,-0_600, 0)

colour(120,200,70)
triangle(-16_000, 0_000,-100,
-13_900, 0_600, 0,
-16_000, 2_800,-100)

colour(100,200,50)
quad(-16_000, 2_800, 100, // Left tailplane upper
-13_900, 0_600, 0,
-14_600, 2_800, 0,
-16_000, 4_500, 0)

colour(120,180,50)
triangle(-16_000, 0_000, 100,
-13_900, 0_600, 0,
-16_000, 2_800, 100)

colour(100,200,50)
quad(-16_000, 2_800,-100, // Left tailplane lower
-13_900, 0_600, 0,
-14_600, 2_800, 0,
-16_000, 4_500, 0)

colour(120,200,70)
triangle(-16_000, 0_000,-100,
-13_900, 0_600, 0,
-16_000, 2_800,-100)

colour(120,200,50)
quad(-16_000,-2_800, 100, // Right tailplane upper
-13_900,-0_600, 0,
-14_600,-2_800, 0,
-16_000,-4_500, 0)

colour(120,180,50)
triangle(-16_000, 0_000, 100,
-13_900,-0_600, 0,
-16_000,-2_800, 100)

colour(100,200,50)
quad(-16_000,-2_800,-100, // Right tailplane lower
-13_900,-0_600, 0,
-14.600,-2.800, 0,
-16.000,-4.500, 0)
colour(120,200,70)
triangle(-16.000, 0.000,-100,
-13.900,-0.600, 0,
-16.000,-2.800,-100)

colour(165,100,50)
quadkd(-16.000, 0, 100, 0, 0, // Left elevator
-17.200, 0.600, 0, 2, 1.200, // pt 1
-17.500, 0.900, 0, 2, 1.500, // pt 2
-16.000, 2.800, 100, 0, 0)
quadkd(-16.000, 0,-100, 0, 0, // Left elevator
-17.200, 0.600, 0, 2, 1.200, // pt 1
-17.500, 0.900, 0, 2, 1.500, // pt 2
-16.000, 2.800,-100, 0, 0)

colour(170,150,80)
quadkd(-16.000, 2.800, 100, 0, 0, // Left elevator
-17.650, 3.500, 0, 2, 1.650) // pt 4
quadkd(-16.000, 2.800,-100, 0, 0, // Left elevator
-17.650, 3.500, 0, 2, 1.650) // pt 4

colour(120,170,60)
quadkd(-16.000, 2.800, 100, 0, 0, // Left elevator
-17.650, 3.500, 0, 2, 1.650, // pt 4
-16.700, 4.833, 0, 2, 0.700) // pt 6
quadkd(-16.000, 2.800,-100, 0, 0, // Left elevator
-17.650, 3.500, 0, 2, 1.650, // pt 4
-16.700, 4.833, 0, 2, 0.700) // pt 6

colour(160,120,40)
quadkd(-16.000, 2.800, 100, 0, 0, // Left elevator
-16.700, 4.833, 0, 2, 0.700, // pt 6
-16.300, 4.750, 0, 2, 0.300, // pt 7
-16.000, 4.500, 0, 0, 0) // pt 8
quadkd(-16.000, 2.800,-100, 0, 0, // Left elevator
-16.700, 4.833, 0, 2, 0.700, // pt 6
-16.300, 4.750, 0, 2, 0.300, // pt 7
-16_000, 4_500, 0, 0, 0) // pt 8

colour(165,100,50)
quadkd(-16_000, 0, 100, 0, 0, // Right elevator
  -17_200,-0_600, 0, 2, 1_200, // pt 1
  -17_500,-0_900, 0, 2, 1_500, // pt 2
  -16_000,-2_800, 100, 0, 0)
quadkd(-16_000, 0,-100, 0, 0, // Right elevator
  -17_200,-0_600, 0, 2, 1_200, // pt 1
  -17_500,-0_900, 0, 2, 1_500, // pt 2
  -16_000,-2_800,-100, 0, 0)

colour(170,150,80)
quadkd(-16_000,-2_800, 100, 0, 0, // Right elevator
  -17_500,-0_900, 0, 2, 1_500, // pt 2
  -17_666,-2_000, 0, 2, 1_666, // pt 3
  -17_650,-3_500, 0, 2, 1_650) // pt 4
quadkd(-16_000,-2_800,-100, 0, 0, // Right elevator
  -17_500,-0_900, 0, 2, 1_500, // pt 2
  -17_666,-2_000, 0, 2, 1_666, // pt 3
  -17_650,-3_500, 0, 2, 1_650) // pt 4

colour(120,170,60)
quadkd(-16_000,-2_800, 100, 0, 0, // Right elevator
  -17_650,-3_500, 0, 2, 1_650, // pt 4
  -17_200,-4_650, 0, 2, 1_200, // pt 5
  -16_700,-4_833, 0, 2, 0_700) // pt 6
quadkd(-16_000,-2_800,-100, 0, 0, // Right elevator
  -17_650,-3_500, 0, 2, 1_650, // pt 4
  -17_200,-4_650, 0, 2, 1_200, // pt 5
  -16_700,-4_833, 0, 2, 0_700) // pt 6

colour(160,120,40)
quadkd(-16_000,-2_800, 100, 0, 0, // Right elevator
  -16_700,-4_833, 0, 2, 0_700, // pt 6
  -16_300,-4_750, 0, 2, 0_300, // pt 7
  -16_000,-4_500, 0, 2, 0) // pt 8
quadkd(-16_000,-2_800,-100, 0, 0, // Right elevator
  -16_700,-4_833, 0, 2, 0_700, // pt 6
  -16_300,-4_750, 0, 2, 0_300, // pt 7
  -16_000,-4_500, 0, 0, 0) // pt 8
6.4. A FIRST OPENGL EXAMPLE

colour(165,100,50)
quadkd(-16_000, 0, 100, 0, 0, // Right elevator
  -17_200,-0_600, 0, 2, 1_200, // pt 1
  -17_500,-0_900, 0, 2, 1_500, // pt 2
  -16_000,-2_800, 100, 0, 0)
quadkd(-16_000, 0, -100, 0, 0, // Right elevator
  -17_200,-0_600, 0, 2, 1_200, // pt 1
  -17_500,-0_900, 0, 2, 1_500, // pt 2
  -16_000,-2_800,-100, 0, 0)

// Construct the landscape and runway
writef("// Runway*n")

{ MANIFEST { ns = 50_000
  ws = 5_000
  }
  FOR n = 0 TO 600_000-ns BY ns DO
    FOR w = -20_000 TO 20_000-ws BY ws DO
      { LET m = (17*n XOR 5*w)>>1
        LET r = 150 + m MOD 23
        LET g = 160 + m MOD 13
        LET b = 170 + m MOD 37
        quadland( n, w, 1_000, r, g, b,
          n+w+ws, 1_000, r, g, b,
          n+ns, w+w+ws, 1_000, r, g, b,
          n+ns, w, 1_000, r, g, b)
      }
    }
  }
writef("// The land*n")
// Plot a square region of land
plotland(-10_000_000, -10_000_000, 20_000_000)
}

AND strut(t1, w1, l1, t4, w4, l4) BE
{ LET t2 = (3*t1+t4)/4
  LET w2 = (3*w1+w4)/4
  LET l2 = (3*l1+l4)/4
  LET t3 = (3*t4+t1)/4
  LET w3 = (3*w4+w1)/4
  LET l3 = (3*l4+l1)/4
  LET ta, wa = 50, 30
  LET tb, wb = 110, 50

colour(80,80,80)
quad(t1-ta, w1, l1, t1, t1+ta, w1, l1, t2+tb, w2, l2)
colour(85, 75, 80)
quad(t1-ta, w1, l1, t1, w1-wa, l1, t2+tb, w2, l2)
colour(85, 80, 85)
quad(t1, w1+wa, l1, t1+ta, w1, l1, t2+tb, w2, l2)
colour(75, 80, 80)
quad(t1, t1-wa, l1, t1+ta, w1, l1, t2+tb, w2, l2)
colour(90, 80, 80)
quad(t2-tb, w2, l2, t2, w2+wb, l2, t3, w3+wb, l3)
colour(95, 75, 80)
quad(t2, w2+wb, l2, t2+tb, w2, l2, t3+tb, w3, l3)
colour(90, 85, 80)
quad(t2+tb, w2, l2, t2, w2-wb, l2, t3, w3-wb, l3)
colour(80, 80, 80)
quad(t2, w2-wb, l2, t2-tb, w2, l2, t3-tb, w3, l3)
colour(80, 80, 80)
quad(t4-ta, w4, l4, t4, w4+wa, l4, t3, w3+wb, l3)
colour(85, 75, 80)
quad(t4-ta, w4, l4, t4, w4-wa, l4, t3, w3-wb, l3)
colour(85, 80, 85)
quad(t4, w4+wa, l4, t4+ta, w4, l4, t3+tb, w3, l3)
colour(75, 80, 80)
quad(t4, w4-wa, l4, t4+ta, w4, l4, t3+tb, w3, l3)

AND height(n, w) = VALOF
{
    // Make it zero on or near the runway.
    // Make it small near the runway and typically larger
    // away from the runway.
    LET size = landsize
    LET halfsize = size/2
    LET h = randheight(n, w,
        -halfsize, +halfsize, // x coords
        -halfsize, +halfsize, // y coords
        0, 0, 0, 0) // corner heights
    LET dist = (ABS(n - runwaylength/2)) + (ABS(w))
    LET factor = ? // Will be in the range 0 to 1_000 depending on dist
    LET d1, d2 = 600_000, 3_000_000
    IF dist <= d1 DO factor := 0
    IF dist >= d2 DO factor := 1_000
    IF d1<dist<d2 DO factor := muldiv(1_000, dist-d1, d2-d1)
    // factor is a function of dist. Below d1 it is zero. Between
// d1 and d2 it grows linearly to 1_000. Above d2 it remains at 1_000.
//sawritef("dist=%9.3d factor=%6.3d h=%i9*n", dist, factor, h)
    h := muldiv(h, factor, 1_000) / 1000
//sawritef("h=%i9 h^2=%i9*n", h, h*h)
RESULTIS (h * h)
}

AND randvalue(x, y, max) = VALOF
{ LET a = 123*x >> 1
    LET b = 541*y >> 3
    LET hashval = ABS((a*b XOR b XOR #x1234567)/3)
    hashval := hashval MOD (max+1)
//sawritef("randvalue: (%i9 %i9 %i9) => %i4*n", x, y, max, hashval)
    RESULTIS hashval
}

AND randheight(x, y, x0, x1, y0, y1, h0, h1, h2, h3) = VALOF
{ // Return a random height depending on x and y only.
    // The result is in the range 0 to 1000
    LET k0, k1, k2, k3 = ?, ?, ?, ?
    LET size = x1-x0
    LET sz = size>1_000_000 -> 1_000_000, size/2
    LET sz2 = sz/2

    TEST sz < 100_000
    THEN { // Use linear interpolation based on the heights
        // of the corners.
        // The formula is
        // h = a + bp + cq + dpq
        // where a = h0
        // b = h1 - h0
        // c = h2 - h0
        // d = h3 - h2 - h1 + h0
        // p = (x-x0)/(x1-x0)
        // and q = (y-y0)/(y1-y0)
        // This formula agrees with the heights at four the vertices,
        // and for fixed x it is linear in y, and vice-versa.
        LET a = h0
        LET b = h1-h0
        LET c = h2-h0
        LET d = h3-h2-h1+h0
        b := muldiv(b, x-x0, x1-x0)
        c := muldiv(c, y-y0, y1-y0)
        d := muldiv(muldiv(d, x-x0, x1-x0), y-y0, y1-y0)
        RESULTIS a+b+c+d
    }
}
}  // Calculate the heights of the vertices of the 1/2 sized square
// containing x,y.
LET mx = (x0+x1)/2
LET my = (y0+y1)/2
LET mh = (h0+h1+h2+h3)/4 + randvalue(mx, my, sz) - sz2
TEST x<mx
THEN TEST y<my
THEN {  // Lower left
  LET k1 = (h0+h1)/2 + randvalue(mx, y0, sz) - sz2
  LET k2 = (h1+h2)/2 + randvalue(x0, my, sz) - sz2
  h1, h2, h3 := k1, k2, mh
  x1, y1 := mx, my
  LOOP
}
ELSE {  // Upper left
  LET k0 = (h0+h2)/2 + randvalue(x0, my, sz) - sz2
  LET k3 = (h2+h3)/2 + randvalue(mx, y1, sz) - sz2
  h0, h1, h3 := k0, mh, k3
  x1, y0 := mx, my
  LOOP
}
ELSE TEST y<my
THEN {  // Lower right
  LET k0 = (h0+h1)/2 + randvalue(mx, y0, sz) - sz2
  LET k3 = (h1+h3)/2 + randvalue(x1, my, sz) - sz2
  h0, h2, h3 := k0, mh, k3
  x0, y1 := mx, my
  LOOP
}
ELSE {  // Upper right
  LET k1 = (h1+h3)/2 + randvalue(x1, my, sz) - sz2
  LET k2 = (h0+h2)/2 + randvalue(mx, y1, sz) - sz2
  h0, h1, h2 := mh, k1, k2
  x0, y0 := mx, my
  LOOP
}
} REPEAT

AND plotland(n, w, size) BE
{ LET sz = size/80
  FOR i = 0 TO 79 DO
    { LET n0 = n + i*sz
      LET n1 = n0 + sz


FOR j = 0 TO 79 DO
{ LET w0 = w + j*sz
  LET w1 = w0 + sz
  LET h0 = height(n0, w0)
  LET h1 = height(n0, w1)
  LET h2 = height(n1, w1)
  LET h3 = height(n1, w0)
  LET r, g, b = redfn(n0,w0,h0), greenfn(n0,w0,h0), bluefn(n0,w0,h0)
  //sawritef("calling qualland(%n,%n,...)*n", n0, w0, h0)
  quadland(n0,w0,h0, r, g, b,
         n0,w1,h1, r, g, b,
         n1,w1,h2, r, g, b,
         n1,w0,h3, r, g, b)
}
}

AND plotland1(x0, y0, sx, sy, h0, h1, h2, h3) BE
{ // This construct a rectangle of land with its south western corner
  // at (x0,y0) using world coordinates. The east-west size of the
  // square is sx, and sy is the north-south size. The vertices are
  // numbered 0 to 3 anticlockwise starting ar (x0,y0).
  LET x2, y2 = x0+sx, y0+sy

  TEST sx > 1000_000
  THEN { FOR i = 0 TO 9 DO
    { LET xa = (x0 * (10-i) + x2 * i ) / 10
      LET xb = (x0 * ( 9-i) + x2 * (i+1)) / 10
      LET sx1 = xb-xa

      LET ha = (h0 * (10-i) + h1 * i ) / 10
      LET hb = (h0 * ( 9-i) + h1 * (i+1)) / 10
      LET hc = (h2 * ( 9-i) + h2 * (i+1)) / 10
      LET hd = (h2 * (10-i) + h3 * i ) / 10

      ha := ha + height(xa, y0, sx1)
      hb := hb + height(xa, y0, sx1)
      hc := hc + height(xb, y2, sx1)
      hd := hd + height(xb, y2, sx1)

      FOR j = 0 TO 9 DO
        { LET ya = (y0 * (10-j) + y2 * j ) / 10
          LET yb = (y0 * ( 9-j) + y2 * (j+1)) / 10
          LET sy1 = yb-ya

          quadland(xa, y0, xc, yc, sx1, sy1, r, g, b,
                   xa, y2, xc, yc, sx1, sy1, r, g, b,
                   xc, y2, xc, yc, sx1, sy1, r, g, b,
                   xc, y0, xc, yc, sx1, sy1, r, g, b)
        }
      }
    }
  }
}
LET ka = (ha * (10-j) + hd * j) / 10
LET kb = (hb * (9-j) + hc * (j+1)) / 10
LET kc = (hb * (9-j) + hc * (j+1)) / 10
LET kd = (ha * (10-j) + hd * j) / 10

ka := ka + height(xa, ya, sy1)
kb := kb + height(xb, ya, sy1)
kc := kc + height(xb, yb, sy1)
d := kd + height(xa, yb, sy1)

plotland(xa, ya, sx1, sy1, ka, kb, kc, kd)
}
}
}
ELSE {
LET r, g, b = redfn(x0,y0,h0), greenfn(x0,y0,h0), bluefn(x0,y0,h0)
sawritef("calling qualland(%n,%n,%n,...)*n", x0, y0, h0)
quadland(x0,y0,h0, r, g, b,
x0,y0,h1, r, g, b,
x2,y2,h2, r, g, b,
x2,y0,h3, r, g, b)
}
}

AND redfn(x,y,h) = 100 +
((x * 12345)>>1) MOD 17 +
((y * 23456)>>1) MOD 37 +
((h * 34567)>>1) MOD 53

AND greenfn(x,y,h) = 100 +
((x * 123456)>>1) MOD 17 +
((y * 234567)>>1) MOD 37 +
((h * 345678)>>1) MOD 53

AND bluefn(x,y,h) = 100 +
((x * 1234567)>>1) MOD 17 +
((y * 2345678)>>1) MOD 37 +
((h * 3456789)>>1) MOD 53

The flight simulator program is currently under development and is called gltiger.b, it is currently as follows. Currently you cannot fly the tigermoth but just move it and rotate it, and view it from various directions.

/ *
This program is a demonstration of the OpenGL interface.
### STILL UNDER DEVELOPMENT

It is soon going to be modified to make extensive use of the floating point facilities now available in BCPL. This modification involves changing the BCPL GL library to use floating point.

The BCPL GL library is in g/gl.b with header g/gl.h and is designed to work unchanged with either OpenGL using SDL or OpenGL ES using EGL and some SDL features.

Implemented by Martin Richards (c) April 2015

**History**

20/12/14
Modified the cube to be like a square missile with control surfaces. It will display a rotating tigermoth by default.

03/12/14
Began conversion to use floating point numbers.

**Command argument:**

- **-a/n** Aircraft number, default = 0 for the tigermoth = 1 for the cube-like missile used in gltst.b
- **OBJ** Use OpenGL Objects for vertex and index data
- **-d** Turn on debugging

**Controls:**

- **Q** causes quit
- **P** Output debugging info to the terminal
- **S** Stop/start the stepping the image

**Rotational controls**

- **Right/left arrow** Increase/decrease rotation rate about direction of thrust
- **Up/Down arrow** Increase/decrease rotation rate about direction of left wing
- **> <** Increase/decrease rotation rate about direction of lift
- **R L** Increase/decrease cgndot
- **U D** Increase/decrease cgvdot
- **F B** Increase/decrease cghdot
0,1,2,3,4,5,6,7  Set eye direction -- the eye is always looking at the CG of the aircraft.

8,9  Increase/decrease eye height
+,-  Increase/decrease eye distance

The transformations

Three coordinate systems are used in this program.

The first specifies point \((t,w,l)\) on the aircraft where \(t\) is the distance from the centre of gravity (CG) forward in the direction of thrust, \(w\) is the distance from the CG in the direction of the left wing, and \(l\) is the distance in the direction of lift. These three directions are at right angles to each other. Mathematicians describe them as orthogonal.

The second coordinate system \((n,w,h)\) describes points using real world coordinates. \(n\) is the distance north of the origin, \(w\) is the distance west of the origin and \(h\) is the distance (height) above the origin. The origin is chosen to be in the centre line of the runway at its southern most end. The runway is aligned from south to north.

The third coordinate system \((x,y,z)\) describes points as displayed on the screen. In this system the origin is the centre of the screen. \(x\) is the distance to the right of the origin and \(y\) is the distance above the origin, and \(z\) is the distance from the origin towards the viewer. Thus the further a point is from the viewer the more negative will be its \(z\) component. These \(z\) components are used by the graphics hardware to remove surfaces that are hidden behind other surfaces.

The orientation of the aircraft is specified by the following nine direction cosines.

\[
\begin{align*}
\text{ctn; ctw; cth} & \quad // \text{Direction cosines of direction } t \\
\text{cwn; cww; cwh} & \quad // \text{Direction cosines of direction } w \\
\text{cln; clw; clh} & \quad // \text{Direction cosines of direction } l \\
\text{cgn; cgw; cgh} & \quad // \text{Coordinates of the CG} \\
\text{eyedirection} & \quad // =0 \text{ means the eye is looking horizontally} \\
& \quad // \text{ in the direction of thrust.} \\
\text{eyerelh} & \quad // \text{Relative to cgh} \\
\text{eyedistance} & \quad \text{holds the distance between the eye and}
\end{align*}
\]
the CG of the aircraft.

eyepn, eyepw, eyeph specify the real world coordinates of the point (P) the eye is focussing on. P is often the CG of the aircraft.

eyen, eyew, eyeh specify real world coordinates of a point on the line of sight of the eye.

Since standard BCPL now supports floating point operations and the latest Raspberry Pi (Model B-2) has proper support for floating point this program will phase out scales fixed point arithmetic and use floating point instead. This is a simple but extensive change.

*/

GET "libhdr"
GET "gl.h"
GET "gl.b" // Insert the library source code
.
GET "libhdr"
GET "gl.h"

GLOBAL {
    done:ug
    aircraft // =0 or 1
    stepping
done:ug
    debug
glprog
Vshader
Fshader
VertexLoc // Attribute variable locations
ColorLoc
DataLoc // data[0]=ctrl data[1]=value
ModelMatrixLoc // Uniform variable locations
LandMatrixLoc
ControlLoc

CosElevator
SinElevator
CosRudder
SinRudder
CosAileron
SinAileron

modelfile

// The following variables are floating point number
ctn; ctw; cth  // Direction cosines of direction t
cwn; cww; cwh  // Direction cosines of direction w
cln; clw; clh  // Direction cosines of direction l
rtdot; rwdot; rldot  // Anti-clockwise rotation rates
    // about the t, w and l axes

cgn; cgw; cgh  // Coordinates of the CG of the aircraft
    // in feet as a floating point number
cgndot; cgwdot; cghdot  // CG velocity

eyedirection  // =0 to =7
eyerelh  // height of the eye relative to cgh

eyen; eyew; eyeh  // Coordinates of a point on the line of sight
    // from to eye to the origin (0.0,0.0,0.0).
eyedistance  // The distance between the eye and the CG of
    // the aircraft.

// The next four variables must be in consecutive locations
// since @VertexData is passed to loadmodel.
VertexData  // Vector of 32-bit floating point numbers
VertexDataSize  // = number of numbers in VertexData
IndexData  // Vector of 16-bit unsigned integers
IndexDataSize  // = number of 16-bit integers in IndexData

useObjects //= TRUE if using OpenGL Objects
VertexBuffer
IndexBuffer

LandMatrix  // The matrix used by the vertex shader
    // to transform the vertex coordinates of points
    // on the land to screen coordinates.
ModelMatrix  // The matrix used by the vertex shader
    // to transform the vertex coordinates of points
    // on the model to screen coordinates.
WorkMatrix  // is used when constructing the projection matrix.
}
LET start() = VALOF
{ LET m1 = VEC 15
  LET m2 = VEC 15
  LET m3 = VEC 15
  LET argv = VEC 50
  LET modelfile = "tigermothmodel.mdl"
  LET aircraft = 0

  ModelMatrix, LandMatrix, WorkMatrix := m1, m2, m3

  UNLESS rdargs("-a/n, obj/s, -d/s", argv, 50) DO
    { writef("Bad arguments for gltst*n")
      RETURN
    }
  
  IF argv!0 DO aircraft := !argv!0 // -a/n
  useObjects := argv!1 // obj/s
  debug := argv!2 // -d/s

  IF aircraft=1 DO modelfile := "gltst.mdl"

  // writef("start: calling glInit*n")
  UNLESS glInit() DO
    { writef("*nOpenGL not available*n")
      RESULTIS 0
    }
  
  writef("start: calling glMkScreen*n")
  // Create an OpenGL window
  screenxsize := glMkScreen("Tigermoth flight simulator", 800, 680)
  screenysize := result2
  UNLESS screenxsize DO
    { writef("*nUnable to create an OpenGL window*n")
      RESULTIS 0
    }
  
  writef("Screen Size is %n x %n*n", screenxsize, screenysize)

  writef("start: calling glMkProg ")
  glprog := glMkProg()
  writef("=> glprog=%n*n", glprog);

  IF glprog<0 DO
    { writef("*nUnable to create a GL program*n")
      RESULTIS 0
    }
}
// Read and Compile the vertex shader
writef("start: calling CompileV(\n,glprog) 
Vshader := Compileshader(glprog, TRUE, "gltigerVshader.sdr")
writef("=> Vshader=%n\n", Vshader)

// Read and Compile the fragment shader
writef("start: calling CompileF(\n,glprog) 
Fshader := Compileshader(glprog, FALSE, "gltigerFshader.sdr")
writef("=> Fshader=%n\n", Fshader)

// Link the program
writef("start: calling glLinkProg(\n), glprog) UNLESS glLinkProg(glprog) DO
{ writef("Unable to link a GL program\n")
  RESULT := 0
}
writef("start: calling glUseProgram(\n), glprog)
glUseProgram(glprog)

// Get attribute locations after linking
VertexLoc := glGetAttribLocation(glprog, "g_vVertex")
ColorLoc := glGetAttribLocation(glprog, "g_vColor")
DataLoc := glGetAttribLocation(glprog, "g_vData")
writef("VertexLoc=%n\n", VertexLoc)
writef("ColorLoc=%n\n", ColorLoc)
writef("DataLoc=%n\n", DataLoc)

// Get uniform locations after linking
LandMatrixLoc := glGetUniformLocation(glprog, "landmatrix")
ModelMatrixLoc := glGetUniformLocation(glprog, "modelmatrix")
ControlLoc := glGetUniformLocation(glprog, "control")
writef("LandMatrixLoc=%n\n", LandMatrixLoc)
writef("ModelMatrixLoc=%n\n", ModelMatrixLoc)
writef("ControlLoc=%n\n", ControlLoc)

//glDeleteShader(Vshader)
//glDeleteShader(Fshader)

// Load model
6.4. A FIRST OPENGL EXAMPLE

UNLESS loadmodel(modelfile, @VertexData) DO
  { writef("*nUnable to load model: %s*n", modelfile)
    RESULTIS 0
  }

IF debug DO
  { // Output the vertex and index data
    // as a debugging aid
    FOR i = 0 TO VertexDataSize-1 DO
      IF i MOD 8 = 0 DO newline()
      writef(" %8.3d", sc3(VertexData!i))
    }
    newline()
    FOR i = 0 TO (IndexDataSize-1)/2 DO
      LET w = IndexData!i
      IF i MOD 6 = 0 DO writef("*n%i6: ", 2*i)
      writef(" %i5 %i5", w & #xFFFF, w>>16)
    }
    newline()
  }

sys(Sys_gl, GL_Enable, GL_DEPTH_TEST) // This call is neccessary
sys(Sys_gl, GL_DepthFunc, GL_LESS) // This the default

// Pixel written if incoming depth < buffer depth
// This assumes positive Z is into the screen, but
// remember the depth test is performed after all other
// transformations have been done.

TEST useObjects
THEN {
  // Setup the model using OpenGL objects
  writef("start: VertexDataSize=%n*n", VertexDataSize)
  VertexBuffer := sys(Sys_gl, GL_GenVertexBuffer, VertexDataSize, VertexData)

  // Tell GL the positions in VertexData of the xyz fields,
  // ie the first 3 words of each 8 word item in VertexData
  sys(Sys_gl, GL_EnableVertexAttribArray, VertexLoc);
  sys(Sys_gl, GL_VertexData,
       VertexLoc, // Attribute number for xyz data
       3, // 3 floats for xyz
       8, // 8 floats per vertex item in vertexData
       0) // Offset in words of the xyz data
writef("start: VertexData xyz data copied to graphics object %n", VertexBuffer)

// Tell GL the positions in VertexData of the rgb fields,
// ie the second 3 words of each 8 word item in VertexData
sys(Sys_gl, GL_EnableVertexAttribArray, ColorLoc);
sys(Sys_gl, GL_VertexData,
    ColorLoc, // Attribute number rgb data
    3, // 3 floats for rgb data
    8, // 8 floats per vertex item in vertexData
    3) // Offset in words of the rgb data
writef("start: ColourData rgb data copied to graphics object %n", VertexBuffer)

// Tell GL the positions in VertexData of the kd fields,
// ie word 6 of each 8 word item in VertexData
sys(Sys_gl, GL_EnableVertexAttribArray, DataLoc);
sys(Sys_gl, GL_VertexData,
    DataLoc, // Attribute number rgb data
    2, // 2 floats for kd data
    8, // 8 floats per vertex item in vertexData
    6) // Offset in words of the kd data
writef("start: VertexData kd data copied to graphics object %n", VertexBuffer)

// VertexData can now be freed
//freevec(VertexData)

writef("start: IndexDataSize=%n", IndexDataSize)
IndexBuffer := sys(Sys_gl, GL_GenIndexBuffer, IndexData, IndexDataSize)
writef("start: IndexData copied to graphics memory object %n", IndexBuffer)

// IndexData can now be freed
//freevec(IndexData)
} ELSE {
    // Setup the model not using objects
    sys(Sys_gl, GL_EnableVertexAttribArray, VertexLoc);
    sys(Sys_gl, GL_EnableVertexAttribArray, ColorLoc);
    sys(Sys_gl, GL_EnableVertexAttribArray, DataLoc);

    // The next call tells GL where the xyz fields of
    // attribute VertexLoc appear in VertexData. It says
    // that each vertex is specified by items consisting
    // 8 words. The first 3 words of each item contains
    // the xyz values.
6.4. A FIRST OPENGL EXAMPLE

```c
VertexData(VertexLoc,
      3, // 3 Values x, y, z
     8, // Stride of 8 words (=32 bytes)
        // ie 8 values in VertexData per vertex
    VertexData) // Position of xyz value of vertex 0

// The next call tells GL where the rgb fields of
// attribute ColorLoc appear in VertexData. It says
// they are in 3 words at position 3 of each 8 word item.
glVertexData(ColorLoc,
      3, // 3 Values r, g, b
     8, // Stride in words (=32 bytes)
        // ie 8 values in VertexData per vertex
    VertexData+3) // Position of rgb values of vertex 0

// The next call tells GL where the kd fields of
// attribute ColorLoc appear in VertexData. It says
// they are in the last 2 words of each 8 word item.
glVertexData(DataLoc,
      2, // 2 Values k, d
     8, // Stride in words (=32 bytes)
        // ie 8 values in VertexData per vertex
    VertexData+6) // Position of kd values of vertex 0
}

// Initialise the state

done := FALSE
stepping := FALSE

cgn, cgw, cgh := 0.0, 0.0, 20.0
cgndot, cgwdot, cghdot := 0.0, 0.0, 0.0

// Set the initial direction cosines to orient t, w and l in
// directions -z, -x and y, ie viewing the aircraft from behind.

ctn, ctw, cth := 1.0, 0.0, 0.0
cwn, cww, cwh := 0.0, 1.0, 0.0
cln, clw, clh := 0.0, 0.0, 1.0

tndot, tndot, rldot := 0.0, 0.0, 0.0
//tndot, tndot, ldot := 0.002, 0.003, 0.001 // Rotate the model slowly
```
eyedirection := 0 // Direction of thrust
eyerelh := 0.0 // Relative to cgh
eyedistance := 50.000

eyen, eyew, eyeh := 1.0, 0.0, 0.0

IF debug DO
{ glSetvec( WorkMatrix, 16,
    2.0, 0.0, 0.0, 0.0,
    0.0, 1.0, 0.0, 0.0,
    0.0, 0.0, 1.0, 0.0,
    0.0, 0.0, 0.0, 10.0
    )
    glSetvec( LandMatrix, 16,
        1.0, 2.0, 3.0, 4.0,
        5.0, 6.0, 7.0, 8.0,
        9.0, 10.0, 11.0, 12.0,
        13.0, 14.0, 15.0, 16.0
    )
    newline()
    prmat(WorkMatrix)
    writef("times*n")
    prmat(LandMatrix)
    glMat4mul(WorkMatrix, LandMatrix, LandMatrix)
    writef("gives*n")
    prmat(LandMatrix)
    abort(1000)
}

//sawritef("Entering main loop*n")

UNTIL done DO
{ proscsevents()

    // Only rotate the object if not stepping
    UNLESS stepping DO
    { // If not stepping adjust the orientation of the model.
        rotate(rtdot, rwdot, rldot)

        // Move the centre of the model
        cgn := cgn #+ cgndot
        cgw := cgw #+ cgwdot
        cgh := cgh #+ cghdot
    }
}
// We now construct the matrix LandMatrix to transform
// points in real world coordinated to screen coordinates

// We assume the eye is looking directly towards the centre
// of gravity of the model.

// First rotate world coordinate (n,w,u) to
// screen coordinates (x,y,z)
// ie n -> -z
// w -> -x
// u -> y
// and translate the aircraft and land to place the aircraft CG
// to the origin

SWITCHON eyedirection INTO
{ DEFAULT:
  CASE 0: eyen, eyew := #-1.000, 0.000; ENDCASE
  CASE 1: eyen, eyew := #0.707, #0.707; ENDCASE
  CASE 2: eyen, eyew := 0.0, #1.000; ENDCASE
  CASE 3: eyen, eyew := 0.707, #0.707; ENDCASE
  CASE 4: eyen, eyew := 1.0, 0.000; ENDCASE
  CASE 5: eyen, eyew := 0.707, 0.707; ENDCASE
  CASE 6: eyen, eyew := 0.0, 1.000; ENDCASE
  CASE 7: eyen, eyew := #0.707, 0.707; ENDCASE
}

eyeh := eyerelh

// Matrix to move aircraft and land so that the CG of
// the aircraft is at the origin

glSetvec( LandMatrix, 16,
          1.0, 0.0, 0.0, 0.0, // column 1
          0.0, 1.0, 0.0, 0.0, // column 2
          0.0, 0.0, 1.0, 0.0, // column 3
          #-cgn, #-cgw, #-cgh, 1.0 // column 4
)

// Rotate the model and eye until the eye is on the z axis

{ LET en, ew, eh = eyen, eyew, eyeh
  LET oq = glRadius2(en, ew)
  LET op = glRadius3(en, ew, eh)
  LET cos_theta = # en #/ oq
}
LET sin_theta = #- ew #/ oq
LET cos_phi = oq #=> op
LET sin_phi = eh #/ op

// Rotate anti-clockwise about h axis by angle theta
// to move the eye onto the nh plane.
glSetvec( WorkMatrix, 16,
    cos_theta, #=-sin_theta, 0.0, 0.0, // column 1
    sin_theta, cos_theta, 0.0, 0.0, // column 2
    0.0, 0.0, 1.0, 0.0, // column 3
    0.0, 0.0, 0.0, 1.0 // column 4
)
//sawritef("Rotation matrix R1*n")
//prmat(LandMatrix)
//abort(1000)
glMat4mul(WorkMatrix, LandMatrix, LandMatrix)

//newline()
//writef("eyen=%6.3d eyew=%6.3d eyeh=%6.3d*n", eyen, eyew, eyeh)
//writef("cgn= %6.3d cgw= %6.3d cgh= %6.3d*n", cgn, cgw, cgh)
//writef("cos and sin of theta and phi: "); prv(@cos_theta); newline()
//writef("Matrix to rotate and translate the model*n")
//writef("and move the eye into the yz plane*n")
//dbmatrix(LandMatrix)

// Rotate clockwise about w axis by angle phi
// to move the eye onto the n axis.
glSetvec( WorkMatrix, 16,
    cos_phi, 0.0, #=-sin_phi, 0.0, // column 1
    0.0, 1.0, 0.0, 0.0, // column 2
    sin_phi, 0.0, cos_phi, 0.0, // column 3
    0.0, 0.0, 0.0, 1.0 // column 4
)
//sawritef("Rotation matrix R2*n")
//prmat(WorkMatrix)
//abort(1000)
glMat4mul(WorkMatrix, LandMatrix, LandMatrix)

//newline()
//writef("Matrix to rotate and translate the model*n")
//writef("and move the eye onto the z axis*n")
//dbmatrix(LandMatrix)
6.4. A FIRST OPENGL EXAMPLE

// Matrix to transform world coordinates (n,w,h) to
// to screen coordinated ((x,y,z)
// ie x = -w
// y = h
// z = -n

glSetvec(WorkMatrix, 16,
    0.0, 0.0, #-1.0, 0.0, // column 1
    #-1.0, 0.0, 0.0, 0.0, // column 2
    0.0, 1.0, 0.0, 0.0, // column 3
    0.0, 0.0, 0.0, 1.0 // column 4
)

glMat4mul(WorkMatrix, LandMatrix, LandMatrix)

//IF FALSE DO
{
    // Change the origin to the eye position on the z axis by
    // moving the model eyedistance in the negative z direction.
    glSetvec(WorkMatrix, 16,
        1.0, 0.0, 0.0, 0.0, // column 1
        0.0, 1.0, 0.0, 0.0, // column 2
        0.0, 0.0, 1.0, 0.0, // column 3
        0.0, 0.0, #-eyedistance, 1.0 // column 4
    )

    //sawritef("Change to eye origin matrix*n")
    //prmat(WorkMatrix)
    //abort(1000)
    glMat4mul(WorkMatrix, LandMatrix, LandMatrix)

    //newline()
    //writef("Matrix to rotate and translate the model*n")
    //writef("and move the eye onto the z axis*n")
    //writef("and move the eye a distance in the z direction*n")
    //dbmatrix(LandMatrix)
}

//IF FALSE DO
{
    // Define the truncated pyramid for the view projection
    // using the frustrum transformation.
    LET n, f = 0.1, 5000.0
    LET fan, fsn = f#+n, f#-n
    LET n2 = 2.0#*n
    LET l, r = #-0.5, 0.5
LET ral, rsl = r#+1, r#-1
LET b, t = #-0.5, 0.5
LET tab, tsb = t#+b, t#-b

//glSetvec( WorkMatrix, 16,
//         n2#/rsl, 0.0, 0.0, 0.0, // column 1
//         0.0, n2#/tsb, 0.0, 0.0, // column 2
//         ral#/rsl, tab#/tsb, #-fan#/fsn, #-1.0, // column 3
//         0.0, 0.0, #-(n2#*f)#/fsn, 0.0 // column 4
// )

// Alternatively use the perspective transformation explicitly.
{ LET aspect = FLOAT screenxsize #/ FLOAT screenysize
  LET fv = 2.0 // Half field of view at unit distance
  glSetvec( WorkMatrix, 16,
           fv #/ aspect, 0.0, 0.0, 0.0, // column 1
             0.0, fv, 0.0, 0.0, // column 2
             0.0, 0.0, (f #+ n) #/ (n #- f), #-1.0, // column 3
             0.0, 0.0, (2.0 ** f ** n) #/ (n #- f), 0.0 // column 4
          )

  // The perspective matrix could be set more conveniently using
  // glSetPerspective library function defined in g/gl.b
  //glSetPerspective(WorkMatrix,
  //                  aspect, // Aspect ratio
  //                  1.0, // Field of view at unit distance
  //                  0.1, // Distance to near limit
  //                  5000.0) // Distance to far limit
}

//sawritef("work matrix*n")
//prmat(WorkMatrix)
//sawritef("Projection matrix*n")
//prmat(LandMatrix)
//glMat4mul(WorkMatrix, LandMatrix, LandMatrix)
//sawritef("final Projection matrix*n")
//dbmatrix(LandMatrix)

/*
 newline()
 writef(" n="); prf8_3(n)
 writef(" f=\%3.3d", sc3(f))
 writef(" l=\%3.3d", sc3(l))
 writef(" r=\%3.3d", sc3(r))
*/
writef(" b=%8.3d", sc3(b))
writef(" t=%8.3d", sc3(t))
newline()
*/

//abort(1000)
}

// Send the LandMatrix to uniform variable "landmatrix" for
// use by the vertex shader transform land points.
glUniformMatrix4fv(LandMatrixLoc, glprog, LandMatrix)

// Set the model rotation matrix from model
// coordinates (t,w,l) to world coordinates (x,y,z)
glSetvec(ModelMatrix, 16,
    ctn, ctw, cth, 0.0, // column 1
    cwn, cww, cwh, 0.0, // column 2
    cln, clw, clh, 0.0, // column 3
    0.0, 0.0, 0.0, 1.0 // column 4
    )
newline()
writef("Matrix to rotate the model\n")
dbmatrix(LandMatrix)

// Set the model’s centre of gravity to (cgn,cgw,cgh)
glSetvec( WorkMatrix, 16,
    1.0, 0.0, 0.0, 0.0, // column 1
    0.0, 1.0, 0.0, 0.0, // column 2
    0.0, 0.0, 1.0, 0.0, // column 3
    cgn, cgw, cgh, 1.0 // column 4
    )
newline()
writef("Translation matrix\n")
prmat(WorkMatrix)
//abort(1000)

glMat4mul(WorkMatrix, ModelMatrix, ModelMatrix)
newline()
writef("Matrix to rotate and translate the model\n")
dbmatrix(ModelMatrix)
//abort(1000)

// Now apply the projection transformation to the model matrix
glMat4mul(LandMatrix, ModelMatrix, ModelMatrix)
// Send the ModelMatrix to uniform variable "modelmatrix" for
// use by the vertex shader transform points on the model.
glUniformMatrix4fv(ModelMatrixLoc, glprog, ModelMatrix)

// Calculate the cosines and sines of the control surfaces.
{ LET RudderAngle = # - rldot # * 100.0
  CosRudder := sys(Sys_flt, fl_cos, RudderAngle)
  SinRudder := sys(Sys_flt, fl_sin, RudderAngle)
  //writef("RudderAngle = %9.3d  cos=%5.3d  sin=%5.3d\n", 
    //    sc3(RudderAngle), sc3(CosRudder), sc3(SinRudder))
}

{ LET ElevatorAngle = rwdot # * 100.0
  CosElevator := sys(Sys_flt, fl_cos, ElevatorAngle)
  SinElevator := sys(Sys_flt, fl_sin, ElevatorAngle)
  //writef("ElevatorAngle = %9.3d  cos=%5.3d  sin=%5.3d\n", 
    //    sc3(ElevatorAngle), sc3(CosElevator), sc3(SinElevator))
}

{ LET AileronAngle = rtdot # * 100.0
  CosAileron := sys(Sys_flt, fl_cos, AileronAngle)
  SinAileron := sys(Sys_flt, fl_sin, AileronAngle)
}

// Send them to the graphics hardware as elements of the
// uniform matrix "control" for use by the vertex shader.
{ LET control = VEC 15
  FOR i = 0 TO 15 DO control!i := 0.0

    control!00 := CosRudder // 0 0
    control!01 := SinRudder // 0 1
    control!02 := CosElevator // 0 2
    control!03 := SinElevator // 0 3
    control!04 := CosAileron // 1 0
    control!05 := SinAileron // 1 1

  // Send the control values to the graphics hardware
  glUniformMatrix4fv(ControlLoc, glprog, control)
}

//writef("  %5.3d %5.3d %5.3d %5.3d %5.3d %5.3d\n", 
  //    sc3(CosRudder), sc3(CosElevator), sc3(CosAileron), 
  //    sc3(SinRudder), sc3(SinElevator), sc3(SinAileron))
6.4. A FIRST OPENGL EXAMPLE

```c
// Draw a new image
glClearColour(130, 130, 250, 255)
glClearBuffer() // Clear colour and depth buffers
drawmodel()

IF FALSE DO
  FOR i = -1 TO 1 BY 2 DO
    { // Draw half size images either side
      glSetvec(LandMatrix, 16,
               ctn#/100.0, ctw#/100.0, cth#/100.0, 0.0, // column 1
               cwn#/100.0, cww#/100.0, cwh#/100.0, 0.0, // column 2
               cln#/100.0, clw#/100.0, clh#/100.0, 0.0, // column 3
               cgn#+0.450#*(FLOAT i), cgw, cgh, 1.0 // column 4
      )
      glSetPerspective(WorkMatrix, 1.0, 0.5, 0.1, 5000.0)
      glMat4mul(WorkMatrix, LandMatrix, LandMatrix)
      // Send the matrix to uniform variable "matrix" for use
      // by the vertex shader.
      glUniformMatrix4fv(ModelMatrixLoc, glprog, LandMatrix)
      drawmodel()
    }
  glSwapBuffers()
  delay(0.020) // Delay for 1/50 sec
  //abort(1000)
}
sys(Sys_gl, GL_DisableVertexAttribArray, VertexLoc)
sys(Sys_gl, GL_DisableVertexAttribArray, ColorLoc)
sys(Sys_gl, GL_DisableVertexAttribArray, DataLoc)

delay(0.050)
glClose()

RESULTIS 0

AND Compileshader(prog, isVshader, filename) = VALOF
  { // Create and compile a shader whose source code is
    // in a given file.
```
// isVshader=TRUE if compiling a vertex shader
// isVshader=FALSE if compiling a fragment shader
LET oldin = input()
LET oldout = output()
LET buf = 0
LET shader = 0
LET ramstream = findinoutput("RAM:")
LET instream = findinput(filename)
UNLESS ramstream & instream DO
{ writef("Compileshader: Trouble with i/o streams\n")
  RESULTIS -1
}

//Copy shader program to RAM:
//writef("Compiling shader %s\n", filename)
selectoutput(ramstream)
selectinput(instream)

{ LET ch = rdch()
  IF ch=endstreamch BREAK
  wrch(ch)
} REPEAT

wrch(0) // Place the terminating byte

selectoutput(oldout)
endstream(instream)
selectinput(oldin)

buf := ramstream!scb_buf

shader := sys(Sys_gl,
  (isVshader -> GL_CompileVshader, GL_CompileFshader),
  prog,
  buf)

//writef("Compileshader: shader=%n\n", shader)
endstream(ramstream)
RESULTIS shader
}

AND drawmodel() BE
  TEST useObjects
  THEN { // Draw triangles using vertex and index data
    // held in graphics objects
glDrawTriangles(IndexDataSize, 0)
}
ELSE { // Draw triangles using vertex and index data
        // held in main memory
    glDrawTriangles(IndexDataSize, IndexData)
}

AND processesevents() BE WHILE getevent() SWITCHON eventtype INTO
{ DEFAULT:
    //writef("processesevents: Unknown event type = %n", eventtype)
    LOOP

CASE sdle_keydown:
    SWITCHON capitalch(eventa2) INTO
    { DEFAULT: LOOP

    CASE 'Q': done := TRUE
        LOOP

    CASE 'A': abort(5555)
        LOOP

    CASE 'P': // Print direction cosines and other data
        newline()
        writef("xyz= %9.3d %9.3d %9.3d", sc3(cgn),sc3(cgw),sc3(cgh))
        writef("ct %9.6d %9.6d %9.6d rtdot=%9.6d", sc6(ctn),sc6(ctw),sc6(cth), sc6(rtdot))
        writef("cw %9.6d %9.6d %9.6d rwdot=%9.6d", sc6(cwn),sc6(cww),sc6(cwh), sc6(rwdot))
        writef("cl %9.6d %9.6d %9.6d rldot=%9.6d", sc6(cln),sc6(clw),sc6(clh), sc6(rldot))
        newline()
        writef("eyedirection %n", eyedirection)
        writef("eyepos %9.3d %9.3d %9.3d", sc3(eyen), sc3(eyew), sc3(eyeh))
        writef("eyedistance = %9.3d", sc3(eyedistance))
        LOOP

    CASE 'S': stepping := ~stepping
        LOOP

    CASE 'L': // Increase cgwdot
        cgwdot := cgwdot #+ 0.05
        LOOP

CASE 'R': // Decrease cgwdot
    cgwdot := cgwdot #- 0.05
    LOOP

CASE 'U': // Increase cghdot
    cghdot := cghdot #+ 0.05
    LOOP

CASE 'D': // Decrease cghdot
    cghdot := cghdot #- 0.05
    LOOP

CASE 'F': // Increase cgndot
    cgndot := cgndot #+ 0.05
    LOOP

CASE 'B': // Decrease cgndot
    cgndot := cgndot #- 0.05
    LOOP

CASE '0':
    CASE '1':
        CASE '2':
            CASE '3':
                CASE '4':
                    CASE '5':
                        CASE '6':
                            CASE '7':
                                eyedirection := eventa2 - '0'
                                LOOP
                            CASE '8':
                                eyerelh := eyerelh #+ 0.1; LOOP
                            CASE '9':
                                eyerelh := eyerelh #+ # - 0.1; LOOP
                            CASE '=':
                                eyerelh := eyerelh #* 1.1; LOOP
                            CASE '+':
                                eyerelh := eyerelh #* 1.1; LOOP
                            CASE '-':
                                IF eyedistance# >= 1.0 DO
                                    eyedistance := eyedistance #/ 1.1
                                    LOOP
                                END
                            CASE '<', '>', CASE '.':
                                rldot := rldot #+ 0.0005; LOOP
                            CASE '<':
                                rldot := rldot # - 0.0005; LOOP
CASE sdle_arrowdown:  rwdot := rwdot #+ 0.0005; LOOP
CASE sdle_arrowup:    rwdot := rwdot #- 0.0005; LOOP
CASE sdle_arrowleft:  rtdot := rtdot #+ 0.0005; LOOP
CASE sdlearrowright:  rtdot := rtdot #- 0.0005; LOOP
}
LOOP

CASE sdle_quit:       // 12
  writef("QUIT\n");
  sys(Sys_gl, GL_Quit)
  LOOP

CASE sdle_videosize:  // 14
  //writef("videosize\n", eventa1, eventa2, eventa3)
  LOOP
}

// Conversion functions between floating point and scaled values.
AND sc3(x) = glF2N( 1_000, x)
AND sc6(x) = glF2N(1_000_000, x)

AND inprod(a,b,c, x,y,z) =
  // Return the cosine of the angle between two unit vectors.
  a ** x + b ** y + c ** z

AND rotate(t, w, l) BE
{
  // Rotate the orientation of the aircraft
  // t, w and l are assumed to be small and cause
  // rotation about axis t, w, l. Positive values cause
  // anti-clockwise rotations about their axes.

  LET tx = inprod(1.0, #-l, w, ctn,cwn,cln)
  LET wx = inprod( 1.0, #t, ctn,cwn,cln)
  LET lx = inprod(#-w, t, 1.0, ctn,cwn,cln)

  LET ty = inprod(1.0, #-l, w, ctw,cww,clw)
  LET wy = inprod( 1.0, #t, ctw,cww,clw)
  LET ly = inprod(#-w, t, 1.0, ctw,cww,clw)

  LET tz = inprod(1.0, #-l, w, cth,cwh,clh)
  LET wz = inprod( 1.0, #t, cth,cwh,clh)
  LET lz = inprod(#-w, t, 1.0, cth,cwh,clh)

  ctn, ctw, cth := tx, ty, tz
cwn, cww, cwh := wx, wy, wz
cln, clw, clh := lx, ly, lz

adjustlength(@ctn); adjustlength(@cwn); adjustlength(@cln)
adjustortho(@ctn, @cwn); adjustortho(@ctn, @cln); adjustortho(@cwn, @cln)
}

AND adjustlength(v) BE
{ // This helps to keep vector v of unit length
  LET r = glRadius3(v!0, v!1, v!2)
v!0 := v!0 #/ r
v!1 := v!1 #/ r
v!2 := v!2 #/ r
}

AND adjustortho(a, b) BE
{ // This helps to keep the unit vector b orthogonal to a
  LET a0, a1, a2 = a!0, a!1, a!2
  LET b0, b1, b2 = b!0, b!1, b!2
  LET corr = inprod(a0,a1,a2, b0,b1,b2)
b!0 := b0 #- a0 #* corr
b!1 := b1 #- a1 #* corr
b!2 := b2 #- a2 #* corr
}

AND prmat(m) BE
{ prf8_3(m! 0)
  prf8_3(m! 4)
  prf8_3(m! 8)
  prf8_3(m!12)
newline()
  prf8_3(m! 1)
  prf8_3(m! 5)
  prf8_3(m! 9)
  prf8_3(m!13)
newline()
  prf8_3(m! 2)
  prf8_3(m! 6)
  prf8_3(m!10)
  prf8_3(m!14)
newline()
  prf8_3(m! 3)
  prf8_3(m! 7)
  prf8_3(m!11)
  prf8_3(m!15)
6.4. A FIRST OPENGL EXAMPLE

newline()
}

AND prv(v) BE
{ prf8_3(v!0)
  prf8_3(v!1)
  prf8_3(v!2)
  prf8_3(v!3)
}

AND prf8_3(x) BE writef(" %8.3d", sc3(x))

AND dbmatrix(m) BE //IF FALSE DO
{ LET x,y,z,w = ?,?,?,,?
  LET v = 0x
  LET n, p, one = #-0.5, #+0.5, 1.0
  prmat(m); newline()
  x,y,z,w := 1.0,0.0,0.0,1.0
  prv(v); glMat4mulV(m, v, v); writef(" => "); prv(v); newline()
  x,y,z,w := 0.0,1.0,0.0,1.0
  prv(v); glMat4mulV(m, v, v); writef(" => "); prv(v); newline()
  x,y,z,w := 0.0,0.0,1.0,1.0
  prv(v); glMat4mulV(m, v, v); writef(" => "); prv(v); newline()
  x,y,z,w := n,n,p,one
  prv(v); glMat4mulV(m, v, v); writef(" => "); prv(v); newline()
  x,y,z,w := p,n,n,one
  prv(v); glMat4mulV(m, v, v); writef(" => "); prv(v); newline()
  x,y,z,w := n,n,n,one
  prv(v); glMat4mulV(m, v, v); writef(" => "); prv(v); newline()
  x,y,z,w := n,p,p,one
  prv(v); glMat4mulV(m, v, v); writef(" => "); prv(v); newline()
  x,y,z,w := p,p,p,one
  prv(v); glMat4mulV(m, v, v); writef(" => "); prv(v); newline()
  x,y,z,w := p,p,n,one
  prv(v); glMat4mulV(m, v, v); writef(" => "); prv(v); newline()
  x,y,z,w := n,p,n,one
  prv(v); glMat4mulV(m, v, v); writef(" => "); prv(v); newline()
  newline()
}
Appendix A

sdl.h

This appendix give the source of the SDL header file cintcode/g/sdl.h. It is mainly here so I can proof read it on my iPad.

/*
######## UNDER DEVELOPMENT ###############

This is the header file for the SDL features

Implemented by Martin Richards (c) Sept 2012

History:
12/12/12
Added drawtriangle(3d) and drawquad(3d)

28/08/12
Started a major modification of the library.

30/05/12
Initial implementation

g_sdlbase is set in libhdr to be the first global used in the sdl library
It can be overridden by re-defining g_sdlbase after GETting libhdr.

A program wishing to use the SDL library should contain the following lines.

GET "libhdr"
MANIFEST { g_sdlbase=nnn } // Only used if the default setting of 450 in
// libhdr is not suitable.
GET "sdl.h"
GET "sdl.b" // Insert the library source code
GET "libhdr"
MANIFEST { g_sdlbase=nnn } // Only used if the default setting of 450 in
// libhdr is not suitable.
GET "sdl.h"
Rest of the program
*/

GLOBAL {
// More functions will be included in due course
initsdl: g_sdlbase
mkscreen // (title, xsize, ysize)
setcaption // (title)
closesdl // ()

screen // Handle to the screen surface
format // Handle to the screen format, used by eg setcolour

lefts // Used by drawtriangle and drawquad
leftds // Used by drawtriangle3d and drawquad3d
rights // Used by drawtriangle and drawquad
rightds // Used by drawtriangle3d and drawquad3d
depthscreen // Used by drawtriangle3d and drawquad3d
                // holding the depth of a drawn pixel

miny // Used by drawtriangle(3d) and drawquad(3d)
maxy // Used by drawtriangle(3d) and drawquad(3d)

joystick

screenxsize
screenysize
colour // Current colour for screen
maprgb // (r, g, b) create colour for current screen format

resizescreen // (xsize, ysize)
setcolour // (colour) sets colour

currx // Coords of latest point drawn, possibly off screen
curry
currz

prevdrawn // = TRUE if actually drawn

mousex // Mouse state set by getmousestate
mousey
mousebuttons

eventtype // Event type set by getevent()
evento1
evento2
evento3
evento4
evento5

mksurface // (width, height, key)
freesurface // (surf)
selectsurface // (surf, xsize, ysize)
currsurf // Currently selected surface for drawing
currxsize // its width
currysize // its height
setcolourkey // (col)

drawpoint // (x, y) equivalent to drawfillrect(x,y,1,1)
drawpoint3d // (x, y, z)
moveto // (x, y) set (currx, curry) to (x,y)
moveby // (dx, dy) set (currx, curry) to (currx+dx, curry+dy)
drawto // (x, y) in colour from (currx, curry) to (x,y)
drawby // (dx, dy) in colour from (currx, curry) to (currx+dxx,curry+dy)

moveto3d // (x,y,z) set (currx,currency,currz) to (x,y,z)
moveby3d // (dx,dy,dz) set (currx,currency,currz) to (currx+dx,currency+dy,curry+dz)
drawto3d // (x,y,z) draw (currx,currency,currz) to (x,y,z)
drawby3d // (dx,dy,dz) draw (currx,currency,currz) to (currx+dx,currency+dy)

drawquad // (x1,y1,x2,y2,x3,y3,x4,y4) draw a filled quadrilateral
drawtriangle // (x1,y1,x2,y2,x3,y3) draw a filled triangle
setlims // used by drawtriangle and drawquad (sets lefts and rights)
drawquad3d // (x1,y1,z1,x2,y2,z2,x3,y3,z3,x4,y4,z4) draw a filled 3D quadrilateral
drawtriangle3d // (x1,y1,z1,x2,y2,z2,x3,y3,z3) draw a filled 3D triangle
setlims3d // used by drawtriangle3d and drawquad3d (sets lefts, rights, leftds,

drawstring // (str)
drawcircle // (ox, oy, r)
drawrect // (x,y,w,h)
drawellipce // (ox, oy, rx, ry)
drawellipse // (ox, oy, rx, ry)
drawroundrect // (x,y,w,h,r) rect with rounded corners
drawfillroundrect // (x,y,w,h,r) rect with rounded corners
drawfillcircle // (ox, oy, r)
drawfillrect // (x,y,w,h)
fillsurf // (surf)

movesurf // (surf, dx, dy) move entire surface filling vacated pixels with colour
// eg movesurf(screen, -1, 0) move the screen left by one pixel

blitsurf // (src, dsr, x, y)
blitsurfrect // (src, sx, sy, sw, sh, dsr, dx, dy)

getmousestate // set (mousex, mousey, buttons)
getevent // sets event state

sdldelay // (msecs) using the SDL delay mechanism
sdlmsecs // () returns msecs since start of run

hidecursor // ()
showcursor // ()
updatescreen // () display the current screen

plotf // (x, y, format, args...)
plotfstr // Used by plotf

MANIFEST {
// ops used in calls of the form: sys(Sys_sdl, op,...)
// These should work when using a properly configured BCPL Cintcode system
// running under Linux, Windows or or OSX provided the SDL libraries have been
// installed.

sdl_avail=0
sdl_init // initialise SDL with everything
sdl_setvideomode // width, height, bbp, flags
sdl_quit // Shut down SDL
sdl_locksurface // surf
sdl_unlocksurface // surf
sdl_getsurfaceinfo // surf, and a pointer to [flag, format, w, h, pitch, pixels]
sdl_getfmtinfo // fmt, and a pointer to [palette, bitspp, bytespp,
// rloss, rshift, gloss, gshift, bloss, bshift, aloss, ashift,
// colorkey, alpha]
sdl_geterror // str -- fill str with BCPL string for the latest SDL error
sdl_updaterect // surf, left, top, right, bottom
sdl_loadbmp // filename of a .bmp image
sdl_blitsurface // src, srcrect, dest, destrect
sdl_setcolourkey // surf, flags, colorkey
sdl_freesurface // surf
sdl_setalpha // surf, flags, alpha
sdl_imgload // filename -- using the SDL_image library
sdl_delay    // msecs -- the SDL delay function
sdl_flip     // surf -- Double buffered update of the screen
sdl_displayformat  // surf -- convert surf to display format
sdl_waitevent // pointer to [type, args, ... ] to hold details of the next event
                 // return 0 if no events available
sdl_pollevent  // pointer to [type, args, ... ] to hold details of the next event
                 // return 0 if no events available
sdl_getmousestate // pointer to [x, y] returns bit pattern of buttons currently pressed
sdl_loadwav   // file, spec, buff, len
sdl_freewav   // buffer

sdl_wm_setcaption  // string
sdl_videoinfo     // v => [ flags, blit_fill, video_mem, vfmt]
sdl_maprgb       // format, r, g, b
sdl_drawline      //27
sdl_drawhline     //28
sdl_dawvline      //29
sdl_drawcircle    //30
sdl_drawrect      //31
sdl_drawpixel     //32
sdl_drawellipse   //33
sdl_drawfillellipse //34
sdl_drawround     //35
sdl_drawfillround //36
sdl_drawfillcircle //37
sdl_drawfillrect  //38

sdl_fillrect     //39
sdl_fillsurf     //40

// Joystick functions
sdl_numjoysticks   // 41 (index)
sdl_joystickopen   // 42 (index) => joy
sdl_joystickclose  // 43 (index)
sdl_joystickname   // 44 (index)
sdl_joysticknumaxes // 45 (joy)
sdl_joysticknumbuttons // 46 (joy)
sdl_joysticknumballs // 47 (joy)
sdl_joysticknumhats // 48 (joy)

sdl_joystickeventstate //49  sdl_enable=1 or sdl_ignore=0
sdl_getticks         //50 () => msecs since initialisation

sdl_showcursor       //51
APPENDIX A. SDL.H

sdl_hidecursor    //52
sdl_mksurface    //53
sdl_setcolourkey //54

sdl_joystickgetbutton    //55
sdl_joystickgetaxis    //56
sdl_joystickgetball    //57
sdl_joystickgethat    //58

// more to come ...

// SDL events
sdl_ignore = 0
sdl_enable = 1    // eg enable joystick events

sdle_active = 1    // window gaining or losing focus
sdle_keydown = 2    // => mod ch
sdle_keyup = 3    // => mod ch
sdle_mousedown = 4    // => x y
sdle_mousebuttondown = 5    // => buttonbits
sdle_mousebuttonup = 6    // => buttonbits
sdle_joyaxismotion = 7
sdle_joyballmotion = 8
sdle_joyhatmotion = 9
sdle_joybuttondown = 10
sdle_joybuttonup = 11
sdle_quit = 12
sdle_syswmevent = 13
sdle_videoresize = 14
sdle_userevent = 15

sdle_arrowup = 273
sdlearrowdown = 274
sdlearrowright = 275
sdlearrowleft = 276

sdl_init_everything = #xFFFF

sdl_SWSURFACE = #x00000000    // Surface is in system memory
sdl_HWSURFACE = #x00000001    // Surface is in video memory

sdl_ANYFORMAT = #x10000000    // Allow any video depth/pixel-format
sdl_HWPALETTE = #x20000000    // Surface has exclusive palette
sdl_DOUBLEBUF = #x40000000    // Set up double-buffered video mode
sdl_FULLSCREEN = #x80000000    // Surface is a full screen display
sdl_OPENGL = #x00000002 // Create an OpenGL rendering context
sdl_OPENGLBLIT = #x0000000A // Create an OpenGL rendering context and use it for blitting
sdl_RESIZABLE = #x00000010 // This video mode may be resized
sdl_NOFRAME = #x00000020 // No window caption or edge frame
}
Appendix B

sdl.b

This appendix give the BCPL source of the SDL library cintcode/g/sdl.b. It is mainly here so I can proof read it on my iPad.

/*
#UNDER DEVELOPMENT

This library provides some functions that interface with the SDL Graphics library.

Implemented by Martin Richards (c) September 2012

Change history:

26/08/12
Initial implementation.

It should typically be included as a separate section for programs that need it. Such programs typically have the following structure.

GET "libhdr"
MANIFEST { g_sdlbase=nnn } // Only used if the default setting of 450 in
// libhdr is not suitable.
GET "sdl.h"
GET "sdl.b" // Insert the library source code
.
GET "libhdr"
MANIFEST { g_sdlbase=nnn } // Only used if the default setting of 450 in
// libhdr is not suitable.
GET "sdl.h"
Rest of the program
LET initsdl() = VALOF
{ LET mes = VEC 256/bytesperword

IF sys(Sys_sdl, sdl_init, sdl_init_everything) DO
{ sys(Sys_sdl, sdl_geterror, mes)
  writeln("Unable to initialise SDL: %s", mes)
  RESULTIS FALSE
}

"Number of joysticks %2i", sys(Sys_sdl, sdl_numjoysticks))
  joystick := sys(Sys_sdl, sdl_joystickopen, 0)
// writeln("Number of axis %2i", sys(Sys_sdl, sdl_joysticknumaxes, joystick))
// writeln("Number of buttons %2i", sys(Sys_sdl, sdl_joysticknumbuttons, joystick))

lefts, rights := 0, 0
leftds, rightds := 0, 0
depthscreen := 0

// Successful
RESULTIS TRUE
}

AND mkscreen(title, xsize, ysize) = VALOF
{ // Create a screen surface with given title and size
  LET mes = VEC 256/bytesperword

  screenxsize, screenysize := xsize, ysize

  screen := sys(Sys_sdl, sdl_setvideomode, screenxsize, screenysize, 32, sdl_SWSURFACE)

  UNLESS screen DO
  { sys(Sys_sdl, sdl_geterror, mes)
    writeln("Unable to set video mode: %s", mes)
    RESULTIS 0
  }

  // Surface info structure
  LET flags, fmt, w, h, pitch, pixels, cliprect, refcount = 0, 0, 0, 0, 0, 0, 0
  sys(Sys_sdl, sdl_getsurfaceinfo, screen, @flags)

  format := fmt
}
APPENDIX B. SDL.B

setcaption(title)
selectsurface(screen, xsize, ysize)
}

AND maprgb(r, g, b) = sys(Sys SDL, SDL_maprgb, format, r, g, b)

AND setcaption(title) BE sys(Sys SDL, SDL_WM_setcaption, title, 0)

AND closesdl() BE
{ IF lefts DO freevec(lefts)
  IF rights DO freevec(rights)
  IF leftds DO freevec(leftds)
  IF rightds DO freevec(rightds)
  IF depthscreen DO freevec(depthscreen)
  sys(Sys SDL, SDL_quit)
}

AND setcolour(col) BE colour, prevdrawn := col, FALSE

AND setcolourkey(surf, col) BE sys(Sys SDL, SDL_setcolourkey, surf, col)

AND selectsurface(surf, xsize, ysize) BE
  currsurf, currxsize, currysize := surf, xsize, ysize

AND moveto(x, y) BE
  currx, curry, prevdrawn := x, y, FALSE

AND moveto3d(x, y, z) BE
  currx, curry, currz, prevdrawn := x, y, z, FALSE

AND drawto1(x, y) BE
{ LET mx, my = ?, ?
  IF x<0 & currx<0 | y<0 & curry<0 | x>=currxsize & currx>=currxsize |
    y>=currysize & curry>=currysize DO
    { currx, curry, prevdrawn := x, y, FALSE
      RETURN
    }
  UNLESS prevdrawn DO drawpoint(currx, curry)

  mx := (x+currx)/2
  my := (y+curry)/2
TEST (mx=currx | mx=x) & (my=curry | my=y) 
THEN drawpoint(x, y) 
ELSE { drawto(mx, my) 
   drawto(x, y) 
}

AND drawpoint(x, y) BE 
{ // (0, 0) is the bottom left point on the surface 
   prevdrawn := FALSE 
   IF 0<=x<currxsize & 0<=y<currysize DO 
   { sys(Sys_sdl, sdl_fillrect, currsurf, x, currysize-y, 1, 1, colour) 
     prevdrawn := TRUE 
   } 
   currx, curry := x, y 
}

AND drawpoint3d(x, y, z) BE 
{ // (0, 0) is the bottom left point on the surface 
   prevdrawn := FALSE 
   //IF y<2 DO writef("drawpoint3d: (%i3,%i3,%i3)*n", x,y,z) 
   //IF y<0 DO abort(1234) 
   IF 0<=x<currxsize & 0<=y<currysize DO 
   { LET p = @(depthscreen!(x+y*currxsize)) 
     IF z<!p DO 
     { !p := z 
       sys(Sys_sdl, sdl_fillrect, currsurf, x, currysize-y, 1, 1, colour) 
       prevdrawn := TRUE 
     } 
   } 
   currx, curry, currz := x, y, z 
}

AND moveby(dx, dy) BE moveto(currx+dx, curry+dy) 
AND drawby(dx, dy) BE drawto(currx+dx, curry+dy) 

AND moveby3d(dx, dy, dz) BE moveto3d(currx+dx, curry+dy, currz+dz) 
AND drawby3d(dx, dy, dz) BE drawto3d(currx+dx, curry+dy, currz+dz) 

AND getevent() = VALOF 
{ //writef("Calling pollevent*n") 
   RESULTIS sys(Sys_sdl, sdl_pollevent, @eventtype) 
}
APPENDIX B. SDL.B

AND sdl_delay(msecs) BE // Delay using the SDL delay mechanism
    sys(Sys_sdl, sdl_delay, msecs)

AND sdl_msecs() = // returns msecs since start of run
    sys(Sys_sdl, sdl_getticks)

AND hidecursor() = sys(Sys_sdl, sdl_hidecursor)

AND showcursor() = sys(Sys_sdl, sdl_showcursor)

AND update_screen() BE // Display the screen
    sys(Sys_sdl, sdl_flip, screen)

AND mksurface(w, h) = VALOF
    { // writef("mksurface: w=%n h=%n", w, h)
        RESULTIS sys(Sys_sdl, sdl_mksurface, format, w, h)
    }

AND freesurface(surf) BE sys(Sys_sdl, sdl_freesurface, surf)

AND blitsurf(src, dst, x, y) BE
    { // Blit the source surface to the specified position
        // in the destination surface
        LET dx, dy, dw, dh = x, currysize-y-1, 0, 0
        sys(Sys_sdl, sdl_blitsurface, src, 0, dst, @dx)
    }

AND blitsurf_rect(src, srcrect, dst, x, y) BE
    { // Blit the specified rectangle from the source surface to
        // the specified position in the destination surface
        LET dx, dy, dw, dh = x, currysize-y-1, 0, 0
        sys(Sys_sdl, sdl_blitsurface, src, srcrect, dst, @dx)
    }

AND fillsurf(col) BE
    sys(Sys_sdl, sdl_fillsurf, currsurf, col)

AND drawch(ch) BE TEST ch='*n'
THEN { currx, curry := 10, curry-14
}
ELSE { FOR line = 0 TO 11 DO
    write_ch_slice(currx, curry+11-line, ch, line)
    currx := currx+9
}
AND write_ch_slice(x, y, ch, line) BE
{
    // Writes the horizontal slice of the given character.
    // Character are 8x12
    LET cx, cy = currx, curry
    LET i = (ch&#x7F) - 's'
    LET charbase = TABLE // Still under development !!!
    #X00000000, #X00000000, #X00000000, // space
    #X18181818, #X18180018, #X18000000, // !
    #X66666600, #X00000000, #X00000000, // "
    #X6666FFFF, #X66FFFF66, #X66000000, // #
    #X7EFF8FE, #X7F1B1BFF, #X7E000000, // $
    #X06666C0C, #X18303666, #X60000000, // %
    #X3078C8C8, #X7276DCCC, #X76000000, // &
    #X18181800, #X00000000, #X00000000, // '
    #X18306060, #X60606030, #X18000000, // ( 
    #X180C0606, #X0606060C, #X18000000, // )
    #X00009254, #X38FE3854, #X92000000, // *
    #X00000018, #X187E7E18, #X18000000, // +
    #X00000000, #X00001818, #X08100000, // ,
    #X00000000, #X007E7E00, #X00000000, // -
    #X00000000, #X0000018, #X18000000, // .
    #X06060C0C, #X18130303, #X60600000, // /
    #X386C8C6C, #X6C6C6C6C, #X38000000, // 0
    #X18387818, #X18181818, #X18000000, // 1
    #X3C7E6206, #X0C18307E, #X7E000000, // 2
    #X3C6E4606, #X1C064666, #X3C000000, // 3
    #X1C3C3C6C, #XCCFFFF0C, #X0C000000, // 4
    #X7E7E6E60, #X7C0E6666, #X3C000000, // 5
    #X3C7E6060, #X7C66667E, #X3C000000, // 6
    #X7E7E6060, #X0C183060, #X40000000, // 7
    #X3C666666, #X3C66666E, #X3C000000, // 8
    #X3C666666, #X3E060666, #X3C000000, // 9
    #X00001818, #X00001818, #X00000000, // : 
    #X00001818, #X00001818, #X08100000, // ;
    #X000060C18, #X30603018, #X0C060000, // <
    #X00000000, #X7C07C00, #X00000000, // =
    #X00603018, #X0C606C18, #X30600000, // >
    #X3C7E0060, #X0C181800, #X18180000, // ?
    #X7E819DA5, #XA5A59F80, #X7F000000, // @
    #X3C7EC3C3, #XFFFFC3C3, #XC3000000, // A
    #XFFEFC3FE, #XFE3C3C3FF, #XFE000000, // B
    #X3E7FC3CO, #X0C0C37F, #X3E000000, // C
    #XFCFEC3C3, #XC3C3C3FE, #XFC000000, // D
#XFFFFC0FC, #XFCC0C0FF, #XFF000000, // E
#XFFFFC0FC, #XFCC0C0C0, #XFC000000, // F
#X3E7FE1C0, #XCFEFE3FF, #X7E000000, // G
#XC3C3C3FF, #XFC3C3C3, #X3C000000, // H
#X18181818, #X18181818, #X18000000, // I
#X7F7F00C0, #X0C00CCFC, #X78000000, // J
#XC26C2CD8, #XFOF8CCC6, #XC2000000, // K
#XC0C0C0C0, #XC0C0C0FE, #XFE000000, // L
#X81C3E7FF, #XDBC3C3C3, #X3C000000, // M
#X83C3E3F3, #XDBCFC7C3, #X10000000, // N
#X7EFC3C3, #X3C3C3C3FF, #X7E000000, // O
#XFEFFC3C3, #XFFEC0C0, #X0C000000, // P
#X7EFFC3C3, #XDBCFC7FE, #X7D000000, // Q
#XFEFFC3C3, #XFFEECC6, #X30000000, // R
#X7EC3C0C0, #X7E0303C3, #X7E000000, // S
#XF0FF1818, #X18181818, #X18000000, // T
#XC3C3C3C3, #X3C3C3C37E, #X3C000000, // U
#X81C3C366, #X663C3C18, #X18000000, // V
#XC3C3C3C3, #XDBFF07C3, #X81000000, // W
#XC3C3663C, #X83C66C3, #X3C000000, // X
#XC3C36666, #X3C3C181B, #X18000000, // Y
#XFFFF060C, #X183060FF, #XFF000000, // Z
#X78786060, #X60606060, #X78780000, // [  
#X60600030, #X181800C0, #X06060000, // \  
#X1E1E6060, #X06060606, #X1E1E0000, // ]  
#X10284400, #X00000000, #X00000000, // ^  
#X00000000, #X00000000, #X00000000, // _  
#X00000000, #X00000000, #X00000000, // `  
#X000007AFE, #X6C6C6C6FE, #X7B000000, // a  
#X0000DCFE, #X6C6C6C6FE, #XDC000000, // b  
#X00007CFE, #X6C6C6C6FE, #X7C000000, // c  
#X00006666, #X6C6C6C6FE, #X76000000, // d  
#X00007CFE, #X6C6C6C6FE, #X7C000000, // e  
#X000078FC, #X0F0F0FC0, #X00000000, // f  
#X000076FE, #X6C6C6C6FE, #X7606FE7C, // g  
#X00000C0C, #X6C6C6C6FE, #X60000000, // h  
#X1818018, #X18181818, #X18000000, // i  
#X0000C0C0, #X0000C0C07C, #X38000000, // j  
#X0000C6CC, #X0880F08CC, #XC6000000, // k  
#X00006060, #X60606067C, #X38000000, // l  
#X000006C6FE, #X6D6D6D6D, #X6D000000, // m  
#X0000DCFE, #X6C6C6C6C6, #X60000000, // n  
#X00007CFE, #X6C6C6C6FE, #X7C000000, // o  
#X00007CFE, #X6C6FEC0C0, #X00000000, // p  
#X00007CFE, #X6C6F7E06, #X06000000, // q
IF i>=0 DO charbase := charbase + 3*i

{ LET col = colour
  LET w = VALOF SWITCHON line INTO
  { CASE 0: RESULTIS charbase!0>>24
  CASE 1: RESULTIS charbase!0>>16
  CASE 2: RESULTIS charbase!0>> 8
  CASE 3: RESULTIS charbase!0
  CASE 4: RESULTIS charbase!1>>24
  CASE 5: RESULTIS charbase!1>>16
  CASE 6: RESULTIS charbase!1>> 8
  CASE 7: RESULTIS charbase!1
  CASE 8: RESULTIS charbase!2>>24
  CASE 9: RESULTIS charbase!2>>16
  CASE 10: RESULTIS charbase!2>> 8
  CASE 11: RESULTIS charbase!2
  }

  IF ((w >> 7) & 1) = 1 DO drawpoint(x, y)
  IF ((w >> 6) & 1) = 1 DO drawpoint(x+1, y)
  IF ((w >> 5) & 1) = 1 DO drawpoint(x+2, y)
  IF ((w >> 4) & 1) = 1 DO drawpoint(x+3, y)
  IF ((w >> 3) & 1) = 1 DO drawpoint(x+4, y)
  IF ((w >> 2) & 1) = 1 DO drawpoint(x+5, y)
  IF ((w >> 1) & 1) = 1 DO drawpoint(x+6, y)
  IF (w & 1) = 1 DO drawpoint(x+7, y)
}

currx, curry := cx, cy
APPENDIX B. SDL.B

AND drawstring(x, y, s) BE
{ moveto(x, y)
    FOR i = 1 TO s%0 DO drawch(s%i)
}

AND plotf(x, y, form, a, b, c, d, e, f, g, h) BE
{ LET oldwrch = wrch
    LET s = VEC 256/bytesperword
    plotfstr := s
    plotfstr%0 := 0
    wrch := plotwrch
    writef(form, a, b, c, d, e, f, g, h)
    wrch := oldwrch
    drawstring(x, y, plotfstr)
}

AND plotwrch(ch) BE
{ LET strlen = plotfstr%0 + 1
    plotfstr%strlen := ch
    plotfstr%0 := strlen
}

AND drawto(x, y) BE
{ // This is Bresenham’s algorithm
    LET dx = ABS(x-currx)
    LET dy = ABS(y-curry)
    LET sx = currx<x -> 1, -1
    LET sy = curry<y -> 1, -1
    LET err = dx-dy
    LET e2 = ?

    { drawpoint(currx, curry)
        IF currx=x & curry=y RETURN
        e2 := 2*err
        IF e2 > -dy DO
            { err := err - dy
                currx := currx+sx
            }
        IF e2 < dx DO
            { err := err + dx
                curry := curry + sy
            }
    } REPEAT
}
AND drawto3d(x, y, z) BE
{ // This is Bresenham’s algorithm
  LET dx = ABS(x-currx)
  AND dy = ABS(y-curry)
  LET sx = currx<x -> 1, -1
  LET sy = curry<y -> 1, -1
  LET py = curry<y -> currxsize, -currxsize
  LET x0, y0, z0 = currx, curry, currz
  LET err = dx-dy
  LET e2 = ?

  //IF y<0 DO
  //{ writef("drawto3d: x=%n y=%n z=%n*n", x,y,z)
  // abort(1237)
  //}
  { drawpoint3d(currx,curry,currz)
    IF currx=x & curry=y RETURN
    e2 := 2*err
    IF e2 > -dy DO
      { err := err - dy
        currx := currx+sx
      }
    IF e2 < dx DO
      { err := err + dx
        curry := curry + sy
      }
    TEST dx>=dy
    THEN currz := z0 + muldiv(z-z0, currx-x0, x-x0)
    ELSE currz := z0 + muldiv(z-z0, curry-y0, y-y0)
  } REPEAT
}

AND setlims(x, y) BE
{ // This is used by drawtriangle and is based on Bresenham’s algorithm
  LET dx = ABS(x-currx)
  AND dy = ABS(y-curry)
  LET sx = currx<x -> 1, -1
  LET sy = curry<y -> 1, -1
  LET err = dx-dy

  IF curry<miny DO miny := curry
  IF curry>maxy DO maxy := curry

  { LET e2 = 2*err
IF currx< lefts!curry DO lefts!curry := currx
IF currx>rights!curry DO rights!curry := currx
IF currx=x & curry=y RETURN
IF e2 > -dy DO
{ err := err - dy
  currx := currx + sx
}
IF e2 < dx DO
{ err := err + dx
  curry := curry + sy
}
} REPEAT

AND alloc2dvecs() BE UNLESS lefts DO
{ lefts := getvec(currysize-1)
  rights := getvec(currysize-1)
  FOR i = 0 TO currysize-1 DO
    lefts!i, rights!i := maxint, minint
}

AND drawquad(x1,y1,x2,y2,x3,y3,x4,y4) BE
{ alloc2dvecs()
  miny, maxy := maxint, minint
  moveto(x1,y1)
  setlims(x2,y2)
  setlims(x3,y3)
  setlims(x4,y4)
  setlims(x1,y1)
  FOR y = miny TO maxy DO
    { moveto(lefts!y, y)
      drawto(rights!y, y)
      lefts!y, rights!y := maxint, minint
    }
    moveto(x1,y1)
}

AND drawtriangle(x1,y1,x2,y2,x3,y3) BE
{ alloc2dvecs()
miny, maxy := maxint, minint

moveto(x1, y1)
setlims(x2, y2)
setlims(x3, y3)
setlims(x1, y1)

FOR y = miny TO maxy DO
{ moveto(lefts!y, y)
  drawto(rights!y, y)
  lefts!y, rights!y := maxint, minint
}
moveto(x1, y1)
}

AND setlims3d(x, y, z) BE
{ // This is used by drawtriangle3d and drawquad3d
  // It is based on Bresenham’s algorithm
  LET dx = ABS(x-currx)
  AND dy = ABS(y-curry)
  LET x0, y0, z0 = currx, curry, currz
  LET sx = currx<x -> 1, -1
  LET sy = curry<y -> 1, -1
  LET err = dx-dy

  { LET e2 = 2*err

    IF 0<=curry<currysize DO
    { IF curry<miny DO miny := curry
      IF curry>maxy DO maxy := curry

      IF currx <= lefts!curry DO
      { lefts!curry := currx
        //IF leftds!curry > currz DO // Bug???
        leftds!curry := currz
      }

      IF currx >= rights!curry DO
      { rights!curry := currx
        //IF rightds!curry > currz DO // Bug???
        rightds!curry := currz
      }
    }
    IF currx=x & curry=y RETURN
  }
IF \( e2 > -dy \) DO
\[
\{ \quad \text{err} := \text{err} - dy \\
\text{currx} := \text{currx} + sx \\
\text{IF } dx \geq dy \text{ DO} \\
\quad \{ \quad \text{currz} := z0 + \text{muldiv}(z-z0, \text{currx-x0}, x-x0) \\
\quad \} \\
\} \\
\text{IF } e2 < dx \text{ DO} \\
\{ \quad \text{err} := \text{err} + dx \\
\quad \text{curry} := \text{curry} + sy \\
\text{IF } dy \geq dx \text{ DO} \\
\quad \{ \quad \text{currz} := z0 + \text{muldiv}(z-z0, \text{curry-y0}, y-y0) \\
\quad \} \\
\} \\
\text{REPEAT} \}
\]

\text{AND alloc3dvecs() BE} \\
\{ \ \text{UNLESS lefts DO} \\
\quad \{ \quad \text{lefts} := \text{getvec(currsyze-1)} \\
\quad \text{right} := \text{getvec(currsyze-1)} \\
\quad \text{FOR } y = 0 \text{ TO currsyze-1 DO} \\
\quad \text{lefts}!y, \text{right}!y := \text{maxint, minint} \\
\quad \} \\
\text{UNLESS leftds DO} \\
\quad \{ \quad \text{leftds} := \text{getvec(currsyze-1)} \\
\quad \text{rightds} := \text{getvec(currsyze-1)} \\
\quad \text{FOR } y = 0 \text{ TO currsyze-1 DO} \\
\quad \text{leftds}!y, \text{rightds}!y := \text{maxint, maxint} \\
\quad \} \\
\text{UNLESS depthscreen DO} \\
\quad \{ \quad \text{depthscreen} := \text{getvec(currxsize*currsyze-1)} \\
\quad \text{FOR } i = 0 \text{ TO currxsize*currsyze-1 DO} \\
\quad \text{depthscreen}!i := \text{maxint} \\
\quad \} \\
\}

\text{AND drawquad3d(x1,y1,z1, x2,y2,z2, x3,y3,z3, x4,y4,z4) BE} \\
\{ \ // \text{Draw a filled convex quadrilateral} \\
\quad \// \text{The points are assumed to be coplanar} \\
\quad \text{alloc3dvecs()} \}
//IF x1=400 & y1=7 DO
//{ write("drawquad3d: x1=%i5 y1=%i5 z1=%i5*n", x1,y1,z1)
// write("drawquad3d: x2=%i5 y2=%i5 z2=%i5*n", x2,y2,z2)
// write("drawquad3d: x3=%i5 y3=%i5 z3=%i5*n", x3,y3,z3)
// write("drawquad3d: x4=%i5 y4=%i5 z4=%i5*n", x4,y4,z4)
// abort(1235)
//}
miny, maxy := maxint, minint
moveto3d (x1,y1,z1)
setlims3d(x2,y2,z2)
setlims3d(x3,y3,z3)
setlims3d(x4,y4,z4)
setlims3d(x1,y1,z1)
//IF miny<0 DO
//{ write("drawquad3d: miny=%n maxy=%n*n", miny, maxy)
// abort(1236)
//}
FOR y = miny TO maxy DO
{ moveto3d( lefts!y, y, leftds!y)
drawto3d(rights!y, y, rightds!y)

lefts!y, rights!y := maxint, minint
leftds!y, rightds!y := maxint, maxint
}
moveto3d(x1,y1,z1)
AND drawtriangle3d(x1,y1,z1, x2,y2,z2, x3,y3,z3) BE
{ alloc3dvecs()

miny, maxy := maxint, minint
moveto3d (x1,y1,z1)
setlims3d(x2,y2,z2)
setlims3d(x3,y3,z3)
setlims3d(x1,y1,z1)
FOR y = miny TO maxy DO
{ moveto3d( lefts!y, y, leftds!y)
drawto3d(rights!y, y, rightds!y)

lefts!y, rights!y := maxint, minint
leftds!y, rightds!y := maxint, maxint
}
} moveto3d(x1,y1,z1)
}

AND drawrect(x0, y0, x1, y1) BE
{ LET xmin, xmax = x0, x1
  LET ymin, ymax = y0, y1
  IF xmin>xmax DO xmin, xmax := x1, x0
  IF ymin>ymax DO ymin, ymax := y1, y0
  FOR x = xmin TO xmax DO
    { drawpoint(x, ymin)
      drawpoint(x, ymax)
    }
  FOR y = ymin+1 TO ymax-1 DO
    { drawpoint(xmin, y)
      drawpoint(xmax, y)
    }
  currx, curry := x0, y0
}

AND drawfillrect(x0, y0, x1, y1) BE
{ LET xmin, xmax = x0, x1
  LET ymin, ymax = y0, y1
  IF xmin>xmax DO xmin, xmax := x1, x0
  IF ymin>ymax DO ymin, ymax := y1, y0
  sys(Sys_sdl, sdl_fillrect, currsurf,
      xmin, currsysize-ymax, xmax-xmin+1, ymax-ymin+1, colour)
  /*
   FOR x = xmin TO xmax FOR y = ymin TO ymax DO
     { drawpoint(x, y)
   }
  */
  currx, curry := x0, y0
}

AND drawroundrect(x0,y0,x1,y1,radius) BE
{ LET xmin, xmax = x0, x1
  LET ymin, ymax = y0, y1
  LET r = radius
  LET f, ddf_x, ddf_y, x, y = ?, ?, ?, ?, ?
  IF xmin>xmax DO xmin, xmax := x1, x0
  IF ymin>ymax DO ymin, ymax := y1, y0
IF r<0 DO r := 0
IF r+r>xmax-xmin DO r := (xmax-xmin)/2
IF r+r>ymax-ymin DO r := (ymax-ymin)/2

FOR x = xmin+r TO xmax-r DO
    { drawpoint(x, ymin)
      drawpoint(x, ymax)
    }
FOR y = ymin+r+1 TO ymax-r-1 DO
    { drawpoint(xmin, y)
      drawpoint(xmax, y)
    }
// Now draw the rounded corners
// This is commonly called Bresenham’s circle algorithm since it
// is derived from Bresenham’s line algorithm.
f := 1 - r
ddf_x := 1
ddf_y := -2 * r
x := 0
y := r
drawpoint(xmax, ymin+r)
drawpoint(xmin, ymin+r)
drawpoint(xmax, ymax-r)
drawpoint(xmin, ymax-r)

WHILE x<y DO
    { // ddf_x = 2*x + 1
      // ddf_y = -2 * y
      // f = x*x + y*y - radius*radius + 2*x - y + 1
      IF f>=0 DO
        { y := y-1
          ddf_y := ddf_y + 2
          f := f + ddf_y
        }
      x := x+1
      ddf_x := ddf_x + 2
      f := f + ddf_x
      drawpoint(xmax-r+x, ymax-r+y) // octant 2
      drawpoint(xmin+r-x, ymax-r+y) // Octant 3
      drawpoint(xmax-r+x, ymin+r-y) // Octant 7
      drawpoint(xmin+r-x, ymin+r-y) // Octant 6
      drawpoint(xmax+r+y, ymax-r+x) // Octant 1
      drawpoint(xmin+r+y, ymax-r+x) // Octant 4
      drawpoint(xmax+r-y, ymin+r-x) // Octant 8
    }
drawpoint(xmin+r-y, ymin+r-x) // Octant 5

currx, curry := x0, y0

AND drawfillroundrect(x0, y0, x1, y1, radius) BE
{ LET xmin, xmax = x0, x1
  LET ymin, ymax = y0, y1
  LET r = radius
  LET f, ddf_x, ddf_y, x, y = ?, ?, ?, ?, ?
  LET lastx, lasty = 0, 0
  IF xmin>xmax DO xmin, xmax := x1, x0
  IF ymin>ymax DO ymin, ymax := y1, y0
  IF r<0 DO r := 0
  IF r+r>xmax-xmin DO r := (xmax-xmin)/2
  IF r+r>ymax-ymin DO r := (ymax-ymin)/2

  FOR x = xmin TO xmax FOR y = ymin+r TO ymax-r DO
    { drawpoint(x, y)
    drawpoint(x, y)
    }

    // Now draw the rounded corners
    // This is commonly called Bresenham’s circle algorithm since it
    // is derived from Bresenham’s line algorithm.
    f := 1 - r
    ddf_x := 1
    ddf_y := -2 * r
    x := 0
    y := r

drawpoint(xmax, ymin+r)
drawpoint(xmin, ymin+r)
drawpoint(xmax, ymax-r)
drawpoint(xmin, ymax-r)

  WHILE x<y DO
    { // ddf_x = 2*x + 1
      // ddf_y = -2 * y
      // f = x*x + y*y - radius*radius + 2*x - y + 1
      IF f>=0 DO
        { y := y-1
          ddf_y := ddf_y + 2
          x := x+1
        }
      }
\[ f := f + ddf_y \]
\}
x := x + 1
ddf_x := ddf_x + 2
f := f + ddf_x
drawpoint(xmax - r + x, ymax - r + y) // octant 2
drawpoint(xmin + r - x, ymax - r + y) // Octant 3
drawpoint(xmax - r + x, ymin + r - y) // Octant 7
drawpoint(xmin + r - x, ymin + r - y) // Octant 6
drawpoint(xmax - r + y, ymax - r + x) // Octant 1
drawpoint(xmin + r - y, ymax - r + x) // Octant 4
drawpoint(xmax - r + y, ymin + r - x) // Octant 8
drawpoint(xmin + r - y, ymin + r - x) // Octant 5

UNLESS x = lastx DO
{ FOR fx = xmin + r - y + 1 TO xmax - r + y - 1 DO
  { drawpoint(fx, ymax - r + x)
    drawpoint(fx, ymin + r - x)
  }
lastx := x
}
UNLESS y = lasty DO
{ FOR fx = xmin + r - x + 1 TO xmax - r + x - 1 DO
  { drawpoint(fx, ymax - r + y)
    drawpoint(fx, ymin + r - y)
  }
}
}

\[ \text{currx, curry} := x_0, y_0 \]

AND drawcircle(x_0, y_0, radius) BE
{ // This is commonly called Bresenham’s circle algorithm since it
  // is derived from Bresenham’s line algorithm.
  LET f = 1 - radius
  LET ddf_x = 1
  LET ddf_y = -2 * radius
  LET x = 0
  LET y = radius
drawpoint(x_0, y_0 + radius)
drawpoint(x_0, y_0 - radius)
drawpoint(x_0 + radius, y_0)
drawpoint(x_0 - radius, y_0)
WHILE x<y DO
  { // ddf_x = 2*x + 1
    // ddf_y = -2 * y
    // f = x*x + y*y - radius*radius + 2*x - y + 1
    IF f>=0 DO
      { y := y-1
        ddf_y := ddf_y + 2
        f := f + ddf_y
      }
      x := x+1
      ddf_x := ddf_x + 2
      f := f + ddf_x
      drawpoint(x0+x, y0+y)
      drawpoint(x0-x, y0+y)
      drawpoint(x0+x, y0-y)
      drawpoint(x0-x, y0-y)
      drawpoint(x0+y, y0+x)
      drawpoint(x0-y, y0+x)
      drawpoint(x0+y, y0-x)
      drawpoint(x0-y, y0-x)
    }
  }

AND drawfillcircle1(x0, y0, radius) BE
  { IF y0<radius DO y0 := radius
    IF y0>=currysize-radius DO y0 := currysize-radius
    sys(Sys_sdl, sdl_drawfillcircle, currsurf, x0, currysize-y0, radius, colour)
  }

AND drawfillcircle(x0, y0, radius) BE
  { // This is commonly called Bresenham’s circle algorithm since it
    // is derived from Bresenham’s line algorithm.
    LET f = 1 - radius
    LET ddf_x = 1
    LET ddf_y = -2 * radius
    LET x = 0
    LET y = radius
    LET lastx, lasty = 0, 0
    drawpoint(x0, y0+radius)
    drawpoint(x0, y0-radius)
    FOR x = x0-radius TO x0+radius DO drawpoint(x, y0)
    WHILE x<y DO
      { // ddf_x = 2*x + 1
        // ddf_y = -2 * y
        // f = x*x + y*y - radius*radius + 2*x - y + 1
        IF f>=0 DO
          { y := y-1
            ddf_y := ddf_y + 2
            f := f + ddf_y
          }
          x := x+1
          ddf_x := ddf_x + 2
          f := f + ddf_x
          drawpoint(x0+x, y0+y)
          drawpoint(x0-x, y0+y)
          drawpoint(x0+x, y0-y)
          drawpoint(x0-x, y0-y)
          drawpoint(x0+y, y0+x)
          drawpoint(x0-y, y0+x)
          drawpoint(x0+y, y0-x)
          drawpoint(x0-y, y0-x)
        }
      }
    }
  }
// ddf_y = -2 * y
// f = x*x + y*y - radius*radius + 2*x - y + 1
IF f>=0 DO
{ y := y-1
  ddf_y := ddf_y + 2
  f := f + ddf_y
}
x := x+1
ddf_x := ddf_x + 2
f := f + ddf_x
drawpoint(x0+x, y0+y)
drawpoint(x0-x, y0+y)
drawpoint(x0+x, y0-y)
drawpoint(x0-x, y0-y)
drawpoint(x0+y, y0+x)
drawpoint(x0-y, y0+x)
drawpoint(x0+y, y0-x)
drawpoint(x0-y, y0-x)
UNLESS x=lastx DO
{ FOR fx = x0-y+1 TO x0+y-1 DO
  { drawpoint(fx, y0+x)
    drawpoint(fx, y0-x)
  }
  lastx := x
}
UNLESS y=lasty DO
{ FOR fx = x0-x+1 TO x0+x-1 DO
  { drawpoint(fx, y0+y)
    drawpoint(fx, y0-y)
  }
  lasty := y
}

AND getmousestate() = VALOF
{ writef("*ngetmousestate: not available*n")
  abort(999)
}
Appendix C

Package Installation Details

All the programs described in this document are designed to run on the Raspberry Pi, but they can also run on almost any other machine including those running Linux, Windows or Mac OSX. The annoying problem is that you will have to install the relevant packages unless they are already present. This can be a daunting and error prone task unless you are already an experienced systems programmer. This appendix has been written, mainly for my benefit, to remind me of the packages I have used and how to install them on the various machines I have access to, namely, the Raspberry Pi, a laptop running either Ubuntu Linux or Windows and a Mac Mini running Mac OSX.

The documentation here is typically rather terse, consisting mainly of sequences of commands to install and check each package. Details of how install the packages under Windows will be added in due course.

C.0.1 Installing BCPL under Linux, the Raspberry Pi and Mac OSX

First obtain bcpl.tgz from my home page (www.cl.cam.ac.uk/~mr10) and place it in a directory called ~/Downloads. Then type the following commands.

cd
mkdir distribution
cd distribution
tar zxvf ~/Downloads/bcpl.tgz
cd BCPL/cintcode
cp -r Elisp $HOME
cp .emacs $HOME

For Linux on my laptop I then type:

.o/s/linux/setbcplev
make clean
make
c compall

For the Raspberry Pi, the BCPL system can be built by typing:

.os/linux/setbcplenv
make clean
make -f MakefileRaspi
c compall

For Mac OSX type:

.os/MacOSX/setbcplenv
make clean
make -f MakefileMacOSX
c compall

You might like to put . $HOME/distribution/BCPL/cintcode/os/linux/setbcplenv as a line in .bashrc so that the BCPL environment variables are properly set whenever you login. For the OSX replace linux by MacOSX.

C.0.2 Installing Emacs under Linux, the Raspberry Pi and Mac OSX

On these systems the apt-get command should be available. Before installing anything it is a good idea to type:

sudo apt-get update

Emacs can then be installed by typing:

sudo apt-get update
sudo apt-get install emacs

Note the file ~/.emacs and directory Elisp have already been setup when BCPL was installed.
C.0.3 Installing SDL under Linux and the Raspberry Pi

This document originally used the SDL graphics library but since SDL2 is now available, I plan to use it instead since it has many advantages over the original SDL. Until this happens you may still need SDL and this can be installed under Linux or the Raspberry Pi by typing:

```
sudo apt-get update
sudo apt-get install libsdl1.2-dev libsdl-image1.2-dev
sudo apt-get install libsdl-mixer1.2-dev libsdl-ttf2.0-dev
```

To check that is now installed type the following:

```
ls -l /usr/local/bin/sdl-config
sdl-config --cflags --libs
ls /usr/local/include/SDL
ls /usr/local/lib/SDL
```

Having installed SDL you will need to build a version of the BCPL system that uses it. For Linux, this is done my typing:

```
cd $BCPLROOT
make -f MakefileSDL clean
make -f MakefileSDL
```

For the Raspberry Pi, type:

```
cd $BCPLROOT
make -f MakefileRaspiSDL clean
make -f MakefileRaspiSDL
```

You should now be able to run graphics programs such as `bucket` by typing:

```
cd
cd distribution/BCPL/bcplprogs/raspi
cintsys
c b bucket
bucket
```

Under Mac OSX, I only use SDL2.
C.0.4 Installing SDL2 under Linux and the Raspberry Pi

SDL2 is fairly new and is currently not installable using `apt-get` however its source code can be downloaded from [www.libsdl.org](http://www.libsdl.org). Obtain a file with a name such as `sdl2-2.0.3.tar.gz` and place it in `~/Downloads`. Then type:

```bash
cd ~/Downloads
tar zxvf SDL2-2.0.3.tar.gz
cd SDL2-2.0.3
./configure
```

A really useful document describing how to setup SDL2 under Linux can be found using a web search with keywords `SDL2 download for linux`. This document points out that the `./configure` step probably finds that some dependent packages are missing and it recommends running the following before attempting to compile SDL2.

```bash
sudo apt-get install build-essential xorg-dev libudev-dev
sudo apt-get install libfts-dev libgl1-mesa-dev libglu1-mesa-dev
sudo apt-get install libasound2-dev libpulse-dev libopenal-dev
sudo apt-get install libogg-dev libvorbis-dev libaudiofile-dev
sudo apt-get install libpng12-dev libfreetype6-dev libusb-dev
sudo apt-get install libdbus-1-dev zlib1g-dev libdirectfb-dev
```

Type the following should now successfully compile SDL2.

```bash
./configure
make
```

Note the `./configure` creates the file `Makefile` used by `make`. Assuming the `make` step worked, SDL2 can now be installed in its proper place by typing:

```bash
sudo make install
```

To check that it worked, try typing:

```bash
sdl2-config --cflags --libs
ls /usr/local/include/SDL2
ls /usr/local/lib
```

The same approach should work on the Raspberry Pi, but I have not yet tried it. Apparently the compilation of SDL2 takes about 50 minutes so be patient.