Computer Graphics & Image Processing

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Computer Science Tripos Part IB

Neil Dodgson & Peter Robinson

Lent 2012

William Gates Building
15 JJ Thomson Avenue
Cambridge
CB3 0FD

http://www.cl.cam.ac.uk/

This handout includes copies of the slides that will be used in lectures together with some suggested exercises for supervisions. These notes do not constitute a complete transcript of all the lectures and they are not a substitute for text books. They are intended to give a reasonable synopsis of the subjects discussed, but they give neither complete descriptions nor all the background material.

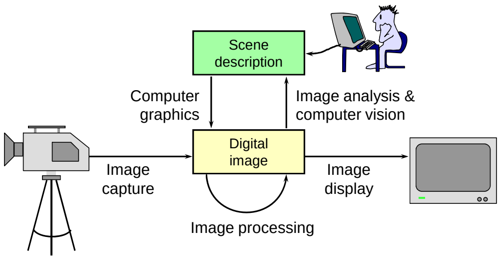
Computer Graphics & Image Processing

1

- Sixteen lectures for Part IB CST
- Neil Dodgson
  - Introduction
  - Colour and displays
  - Image processing
- Peter Robinson
  - 2D computer graphics
  - 3D computer graphics
- Two exam questions on Paper 4

What are Computer Graphics & Image Processing?

2



Why bother with CG & IP?

3

- *All* visual computer output depends on CG
  - printed output (laser/ink jet/phototypesetter)
  - monitor (CRT/LCD/plasma/DMD)
  - all visual computer output consists of real images generated by the computer from some internal digital image
- Much other visual imagery depends on CG & IP
  - TV & movie special effects & post-production
  - most books, magazines, catalogues, flyers, brochures, junk mail, newspapers, packaging, posters

What are CG & IP used for?

4

- 2D computer graphics
  - graphical user interfaces: Mac, Windows, X,…
  - graphic design: posters, cereal packets,…
  - typesetting: book publishing, report writing,…
- Image processing
  - photograph retouching: publishing, posters,…
  - photocollaging: satellite imagery,…
  - art: new forms of artwork based on digitised images
- 3D computer graphics
  - visualisation: scientific, medical, architectural,…
  - Computer Aided Design (CAD)
  - entertainment: special effect, games, movies,…

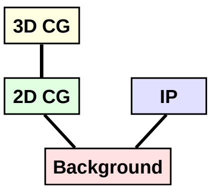
430 thousand printing companies worldwide £250 billion annual turnover

20 million users worldwide

Course Structure

5

- Background [3L]
  - images, human vision, displays
- 2D computer graphics [4L]
  - lines, curves, clipping, polygon filling, transformations
- 3D computer graphics [6L]
  - projection (3D→2D), surfaces, clipping, transformations, lighting, filling, ray tracing, texture mapping
- Image processing [3L]
  - filtering, compositing, half-toning, dithering, encoding, compression



Course books

6

- *Computer Graphics: Principles & Practice*
  - Foley, van Dam, Feiner & Hughes,Addison-Wesley, 1990
    - Older version: *Fundamentals of Interactive Computer Graphics*
      - Foley & van Dam, Addison-Wesley, 1982
- *Computer Graphics & Virtual Environments*
  - Slater, Steed, & Chrysanthou, Addison-Wesley, 2002
- *Digital Image Processing*
  - Gonzalez & Woods, Addison-Wesley , 2008 (5th ed.) , 2003 (4th ed.) or 1992 (3rd ed.)
    - Alternatives:
      - *Digital Image Processing*, Gonzalez & Wintz
      - *Digital Picture Processing*, Rosenfeld & Kak

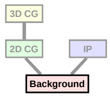
7

Past exam questions

- Prof. Dodgson has been lecturing the course since 1996
  - the course changed considerably between 1996 and 1997
  - all questions from 1997 onwards are good examples of his question setting style
  - do not worry about the last 5 marks of 97/5/2
    - this is now part of Advanced Graphics syllabus
- do not attempt exam questions from 1994 or earlier
  - the course was so different back then that they are not helpful

8

Background



- what is a digital image?
  - what are the constraints on digital images?
- what hardware do we use?

Later on in the course we will ask:

- how does human vision work?
  - what are the limits of human vision?
  - what can we get away with given these constraints & limits?
- how do we represent colour?
- how do displays & printers work?
  - how do we fool the human eye into seeing what we want it to see?

9

What is an image?

- two dimensional function
- value at any point is an intensity or colour
- not digital!

10

What is a *digital* image?

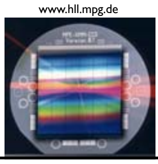
- a contradiction in terms
  - if you can see it, it's not digital
  - if it's digital, it's just a collection of numbers
- a sampled and quantised version of a real image
- a rectangular array of intensity or colour values

11

Image capture

- a variety of devices can be used
  - scanners
    - line CCD (charge coupled device) in a flatbed scanner
    - spot detector in a drum scanner
  - cameras
    - area CCD

area CCD
www.hll.mpg.de



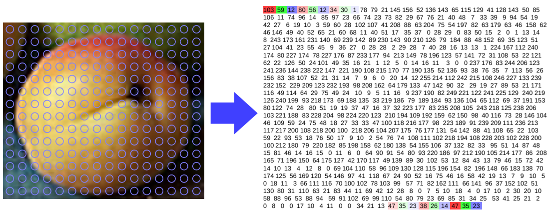
flatbed scanner
www.nuggetlab.com

Heidelberg drum scanner

The image of the Heidelberg drum scanner and many other images in this section come from "Handbook of Print Media", by Helmutt Kipphan, Springer-Verlag, 2001

12

Image capture example



A real image A digital image

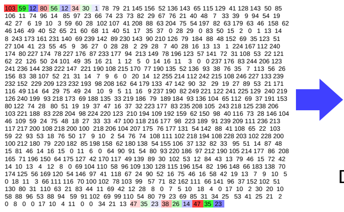
13

Image display

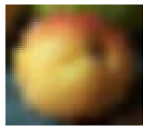
- a digital image is an array of integers, how do you display it?
- reconstruct a real image on some sort of display device
  - CRT — computer monitor, TV set
  - LCD — portable computer, video projector
  - DMD — video projector
  - printer — ink jet, laser printer, dot matrix, dye sublimation, commercial typesetter

14

Image display example



The image data

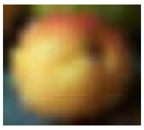


Displayed on a CRT

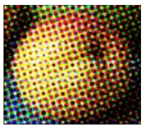
15

Different ways of displaying the same digital image

Nearest-neighbour
e.g. LCD



Gaussian
e.g. CRT



Half-toning
e.g. laser printer

- the display device has a significant effect on the appearance of the displayed image

16

Sampling

- a digital image is a rectangular array of intensity values
- each value is called a *pixel*
  - "picture element"
- sampling resolution is normally measured in pixels per inch (ppi) or dots per inch (dpi)
  - computer monitors have a resolution around 100 ppi
  - laser and ink jet printers have resolutions between 300 and 1200 ppi
  - typesetters have resolutions between 1000 and 3000 ppi

17

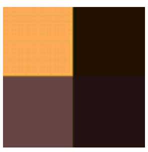
Sampling resolution

256×256

128×128

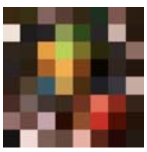
64×64

32×32



2×2

4×4



8×8

16×16

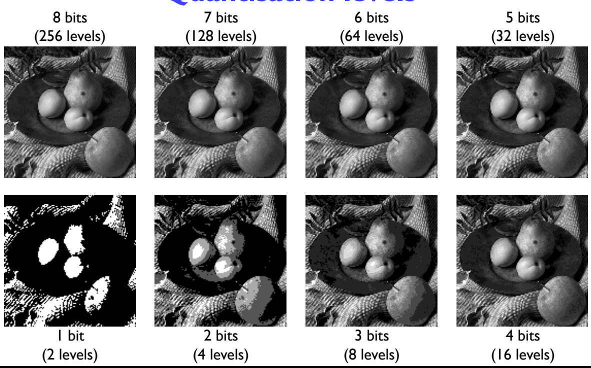
18

Quantisation

- each intensity value is a number
- for digital storage the intensity values must be quantised
  - limits the number of different intensities that can be stored
  - limits the brightest intensity that can be stored
- how many intensity levels are needed for human consumption
  - 8 bits often sufficient
  - some applications use 10 or 12 or 16 bits
  - more detail later in the course
- colour is stored as a set of numbers
  - usually as 3 numbers of 5–16 bits each
  - more detail later in the course

19

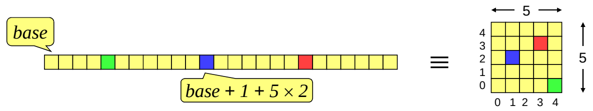
Quantisation levels



20

Storing images in memory

- 8 bits became a *de facto* standard for greyscale images
  - 8 bits = 1 byte
  - 16 bits is now being used more widely, 16 bits = 2 bytes
  - an 8 bit image of size $W \times H$ can be stored in a block of $W \times H$ bytes
  - one way to do this is to store `pixel[x][y]` at memory location $base + x + W \times y$
    - memory is 1D, images are 2D



21

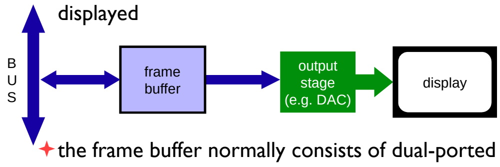
Colour images

- tend to be 24 bits per pixel
  - 3 bytes: one red, one green, one blue
  - increasing use of 48 bits per pixel, 2 bytes per colour plane
- can be stored as a contiguous block of memory
  - of size $W \times H \times 3$
- more common to store each colour in a separate "plane"
  - each plane contains just $W \times H$ values
- the idea of planes can be extended to other attributes associated with each pixel
  - alpha plane (*transparency*), z-buffer (*depth value*), A-buffer (*pointer to a data structure containing depth and coverage information*), overlay planes (*e.g. for displaying pop-up menus*) — see later in the course for details

22

The frame buffer

- most computers have a special piece of memory reserved for storage of the current image being displayed

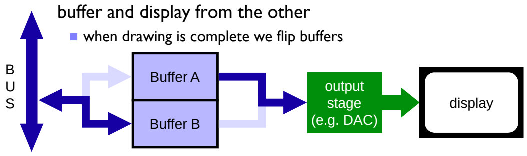


- the frame buffer normally consists of dual-ported Dynamic RAM (DRAM)
  - sometimes referred to as Video RAM (VRAM)

23

Double buffering

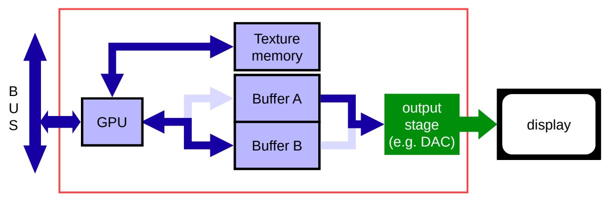
- if we allow the currently displayed image to be updated then we may see bits of the image being displayed halfway through the update
  - this can be visually disturbing, especially if we want the illusion of smooth animation
- double buffering solves this problem: we draw into one frame buffer and display from the other
  - when drawing is complete we flip buffers



24

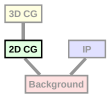
Modern graphics cards

- most graphics processing is now done on a separate graphics card
- the CPU communicates primitive data over the bus to the special purpose Geometry Processing Unit (GPU)
- there is additional video memory on the graphics card, mostly used for storing textures, which are mostly used in 3D games



25

2D Computer Graphics



- lines
  - how do I draw a straight line?
- curves
  - how do I specify curved lines?
- clipping
  - what about lines that go off the edge of the screen?
- filled areas
- transformations
  - scaling, rotation, translation, shearing
- applications

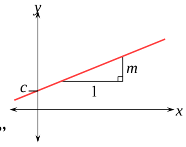
26

Drawing a straight line

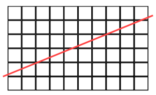
- a straight line can be defined by:

$$y = mx + c$$

the slope of the line



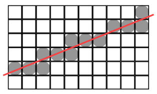
- a mathematical line is "length without breadth"
- a computer graphics line is a set of pixels
- which pixels do we need to turn on to draw a given line?



27

Which pixels do we use?

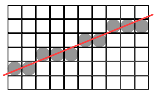
- there are two reasonably sensible alternatives:



every pixel through which the line passes

for lines of slope less than 45° we can have either one or two pixels in each column

✗



the "closest" pixel to the line in each column

for lines of slope less than 45° we always have just one pixel in every column

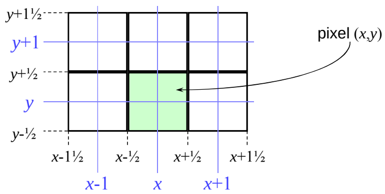
✓

- **in general, use this**

28

A line drawing algorithm — preparation 1

- pixel (x,y) has its centre at real co-ordinate (x,y)
  - it thus stretches from $(x-½, y-½)$ to $(x+½, y+½)$

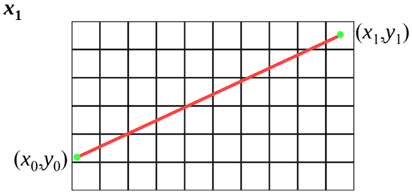


Beware: not every graphics system uses this convention. Some put real co-ordinate (x,y) at the bottom left hand corner of the pixel.

29

A line drawing algorithm — preparation 2

- the line goes from (x_0,y_0) to (x_1,y_1)
- the line lies in the first octant $(0 \le m \le 1)$
- $x_0 < x_1$



30

Bresenham's line drawing algorithm 1

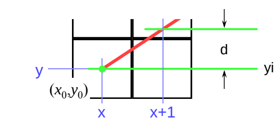
Initialisation

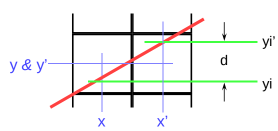
```
d = (y1 - y0) / (x1 - x0)
x = x0
yi = y0
y = y0
DRAW(x,y)
```

Iteration

```
WHILE x < x1 DO
    x = x + 1
    yi = yi + d
    y = ROUND(yi)
    DRAW(x,y)
END WHILE
```

assumes integer end points





J. E. Bresenham, "Algorithm for Computer Control of a Digital Plotter", *IBM Systems Journal*, **4**(1), 1965

Bresenham's line drawing algorithm 2

31

- this slide and the next show how we can optimise Bresenham's algorithm for use on an architecture where integer operations are much faster than floating point
- naïve algorithm involves floating point arithmetic & rounding inside the loop
  $\Rightarrow$ slow
- Speed up A:
  - separate integer and fractional parts of yi (into y and yf)
  - replace rounding by an IF
    - removes need to do rounding

```
d = (y1 - y0) / (x1 - x0)
x = x0
yf = 0
y = y0
DRAW(x,y)
WHILE x < x1 DO
    x = x + 1
    yf = yf + d
    IF ( yf > ½ ) THEN
        y = y + 1
        yf = yf - 1
    END IF
    DRAW(x,y)
END WHILE
```

Bresenham's line drawing algorithm 3

32

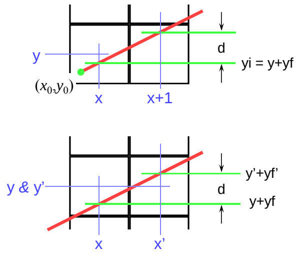
- Speed up B:
  - multiply all operations involving yf by $2(x_1 - x_0)$
    - yf = yf + dy/dx → yf = yf + 2dy
    - yf > ½ → yf > dx
    - yf = yf - 1 → yf = yf - 2dx
  - removes need to do floating point arithmetic *if end-points have integer co-ordinates*

```
dy = (y1 - y0)
dx = (x1 - x0)
x = x0
yf = 0
y = y0
DRAW(x,y)
WHILE x < x1 DO
    x = x + 1
    yf = yf + 2dy
    IF ( yf > dx ) THEN
        y = y + 1
        yf = yf - 2dx
    END IF
    DRAW(x,y)
END WHILE
```

Bresenham's algorithm for floating point end points

33

```
d = (y1 - y0) / (x1 - x0)
x = ROUND(x0)
yi = y0 + d * (x-x0)
y = ROUND(yi)
yf = yi - y
DRAW(x,y)
WHILE x < (x1 - ½) DO
    x = x + 1
    yf = yf + d
    IF ( yf > ½ ) THEN
        y = y + 1
        yf = yf - 1
    END IF
    DRAW(x,y)
END WHILE
```

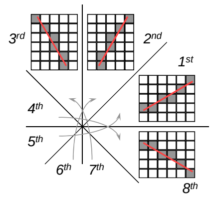


If your end-points are not integers then these kinds of optimisations may not be appropriate. In this case, it is still useful to incorporate speed up A, but not speed up B.

Bresenham's algorithm — more details

34

- we assumed that the line is in the first octant
  - can do fifth octant by swapping end points
- therefore need four versions of the algorithm

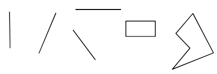


Exercise: work out what changes need to be made to the algorithm for it to work in each of the other three octants

Uses of the line drawing algorithm

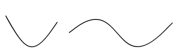
35

- to draw lines

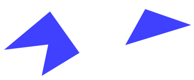


- as the basis for a curve-drawing algorithm

- to draw curves as a sequence of lines



- as the basis for iterating on the edges of polygons in the polygon filling algorithms



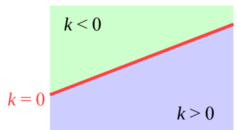
A second line drawing algorithm

36

- a line can be specified using an equation of the form:

$$k = ax + by + c$$

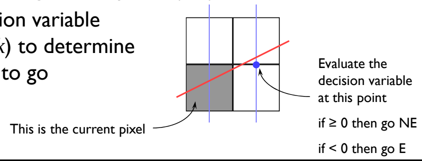
- this divides the plane into three regions:
  - above the line $k < 0$
  - below the line $k > 0$
  - on the line $k = 0$



37

Midpoint line drawing algorithm 1

- first work out the iterative step
  - it is often easier to work out what should be done on each iteration and only later work out how to initialise and terminate the iteration
- given that a particular pixel is on the line, the next pixel must be either immediately to the right (E) or to the right and up one (NE)
- use a decision variable (based on k) to determine which way to go



38

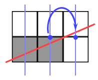
Midpoint line drawing algorithm 2

- decision variable needs to make a decision at point **(x+1, y+½)**

$$d = a(x+1) + b(y+\tfrac{1}{2}) + c$$

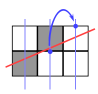
- if go E then the new decision variable is at **(x+2, y+½)**

$$d' = a(x+2) + b(y+\tfrac{1}{2}) + c$$
$$= d + a$$



- if go NE then the new decision variable is at **(x+2, y+1½)**

$$d' = a(x+2) + b(y+1\tfrac{1}{2}) + c$$
$$= d + a + b$$



39

Midpoint line drawing algorithm 3

Initialisation

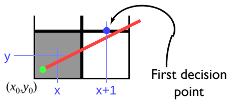
```
a = (y1 - y0)
b = -(x1 - x0)
c = x1 y0 - x0 y1
x = ROUND(x0)
y = ROUND(y0-(x- x0)(a / b))
d = a * (x+1) + b * (y+½) + c
DRAW(x,y)
```

Iteration

```
WHILE x < (x1 - ½) DO
      x = x + 1
      IF d < 0 THEN
            d = d + a
      ELSE
            d = d + a + b
            y = y + 1
      END IF
      DRAW(x,y)
END WHILE
```

E case
just increment x

NE case
increment x & y



If end-points have integer co-ordinates then all operations can be in integer arithmetic

40

Midpoint — comments

- this version only works for lines in the first octant
  - extend to other octants as for Bresenham
- it is not immediately obvious that Bresenham and Midpoint give identical results, but it can be proven that they do
- Midpoint algorithm can be generalised to draw arbitrary circles & ellipses
  - Bresenham can only be generalised to draw circles with integer radii

41

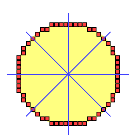
Curves

- circles & ellipses
- Bezier cubics
  - Pierre Bézier, worked in CAD for Renault
  - de Casteljau invented them five years earlier at Citroën
    - but Citroën would not let him publish the results
  - widely used in graphic design & typography
- Overhauser cubics
  - Overhauser, worked in CAD for Ford
- NURBS
  - Non-Uniform Rational B-Splines
  - more powerful than Bezier & now more widely used
  - consider these in Part II

42

Midpoint circle algorithm 1

- equation of a circle is $x^2 + y^2 = r^2$
  - centred at the origin
- decision variable can be $d = x^2 + y^2 - r^2$
  - $d = 0$ on the circle, $d > 0$ outside, $d < 0$ inside
- divide circle into eight octants
  - on the next slide we consider only the second octant, the others are similar



43

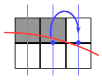
Midpoint circle algorithm 2

- decision variable needed to make a decision at point $(x+1, y-\frac{1}{2})$

$$d=(x+1)^2+(y-\tfrac{1}{2})^2-r^2$$

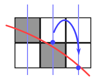
- if go E then the new decision variable is at $(x+2, y-\frac{1}{2})$

$$d'=(x+2)^2+(y-\tfrac{1}{2})^2-r^2$$
$$=d+2x+3$$



- if go SE then the new decision variable is at $(x+2, y-1\frac{1}{2})$

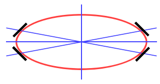
$$d'=(x+2)^2+(y-1\tfrac{1}{2})^2-r^2$$
$$=d+2x-2y+5$$



44

Taking circles further

- the algorithm can be easily extended to circles not centred at the origin
- a similar method can be derived for ovals
  - but: cannot naively use octants
    - use points of 45° slope to divide oval into eight sections
  - and: ovals must be axis-aligned
    - there is a more complex algorithm which can be used for non-axis aligned ovals



Exercise 1: complete the circle algorithm for the second octant

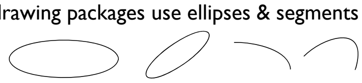
Exercise 2: complete the circle algorithm for the entire circle

Exercise 3: explain how to handle a circle not centred at the origin

45

Are circles & ellipses enough?

- simple drawing packages use ellipses & segments of ellipses



- for graphic design & CAD need something with more flexibility
  - use cubic polynomials
    - lower orders (linear, quadratic) cannot:
      - have a point of inflection
      - match both position and slope at both ends of a segment
      - be non-planar in 3D
    - higher orders (quartic, quintic,…):
      - can wiggle too much
      - take longer to compute

46

Hermite cubic

- the Hermite form of the cubic is defined by its two end-points and by the tangent vectors at these end-points:

$$\begin{aligned}P(t) &= (2t^3-3t^2+1)P_0\\ &+(-2t^3+3t^2)P_1\\ &+(t^3-2t^2+t)T_0\\ &+(t^3-t^2)T_1\end{aligned}$$

- two Hermite cubics can be smoothly joined by matching both position and tangent at an end point of each cubic

Charles Hermite, mathematician, 1822–1901

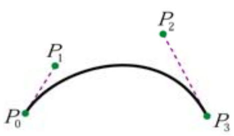
47

Bezier cubic

- difficult to think in terms of tangent vectors
- Bezier defined by two end points and two other control points

$$\begin{aligned}P(t) &= (1-t)^3P_0\\ &+3t(1-t)^2P_1\\ &+3t^2(1-t)P_2\\ &+t^3P_3\end{aligned}$$

where: $P_i \equiv (x_i, y_i)$



Pierre Bézier worked for Renault in the 1960s

48

Bezier properties

- Bezier is equivalent to Hermite

$$T_0=3(P_1-P_0) \quad T_1=3(P_3-P_2)$$

- Weighting functions are Bernstein polynomials

$$b_0(t)=(1-t)^3 \quad b_1(t)=3t(1-t)^2 \quad b_2(t)=3t^2(1-t) \quad b_3(t)=t^3$$

- Weighting functions sum to one

$$\sum_{i=0}^{3} b_i(t)=1$$

- Bezier curve lies within convex hull of its control points

49

Types of curve join

- each curve is smooth within itself
- joins at endpoints can be:
  - C_1 – continuous in both position and tangent vector
    - smooth join in a mathematical sense
  - G_1 – continuous in position, tangent vector in same direction
    - smooth join in a geometric sense
  - C_0 – continuous in position only
    - "corner"
  - discontinuous in position

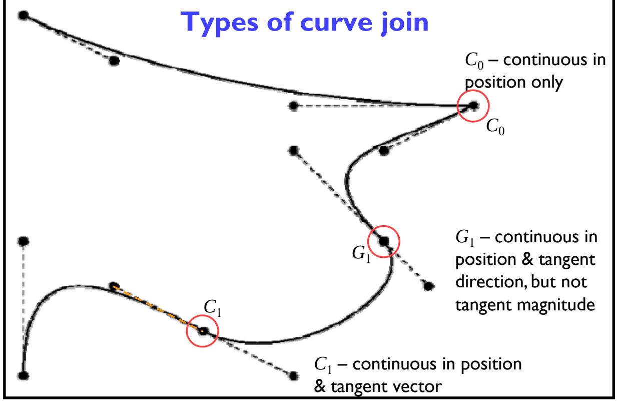
C_n (mathematical continuity): continuous in all derivatives up to the n^{th} derivative

G_n (geometric continuity): each derivative up to the n^{th} has the same "direction" to its vector on either side of the join

$C_n \Rightarrow G_n$

50

Types of curve join



51

Drawing a Bezier cubic – naïve method

- draw as a set of short line segments equispaced in parameter space, t

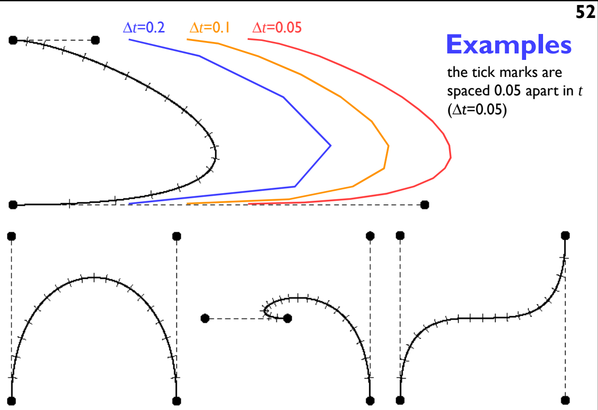
```
(x0,y0) = Bezier(0)
FOR t = 0.05 TO 1 STEP 0.05 DO
        (x1,y1) = Bezier(t)
        DrawLine( (x0,y0), (x1,y1) )
        (x0,y0) = (x1,y1)
END FOR
```

- problems:
  - cannot fix a number of segments that is appropriate for all possible Beziers: too many or too few segments
  - distance in real space, (x,y), is not linearly related to distance in parameter space, t

52

Examples

the tick marks are spaced 0.05 apart in t (Δt=0.05)



53

Drawing a Bezier cubic – sensible method

- adaptive subdivision
  - check if a straight line between P_0 and P_3 is an adequate approximation to the Bezier
  - if so: draw the straight line
  - if not: divide the Bezier into two halves, each a Bezier, and repeat for the two new Beziers
- need to specify some tolerance for when a straight line is an adequate approximation
  - when the Bezier lies within half a pixel width of the straight line along its entire length

54

Drawing a Bezier cubic (continued)

```
Procedure DrawCurve( Bezier curve )
VAR Bezier left, right
BEGIN DrawCurve
        IF Flat( curve ) THEN
                DrawLine( curve )
        ELSE
                SubdivideCurve( curve, left, right )
                DrawCurve( left )
                DrawCurve( right )
        END IF
END DrawCurve
```

e.g. if P_1 and P_2 both lie within half a pixel width of the line joining P_0 to P_3

draw a line between P_0 and P_3: we already know how to do this

this requires some straightforward calculations

55

Checking for flatness

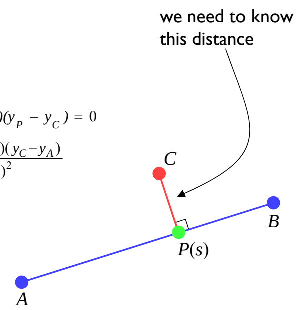
$$P(s) = (1-s)A + sB$$

$$\overline{AB} \cdot \overline{CP(s)} = 0$$

$$\Rightarrow (x_B - x_A)(x_P - x_C) + (y_B - y_A)(y_P - y_C) = 0$$

$$\Rightarrow s = \frac{(x_B - x_A)(x_C - x_A) + (y_B - y_A)(y_C - y_A)}{(x_B - x_A)^2 + (y_B - y_A)^2}$$

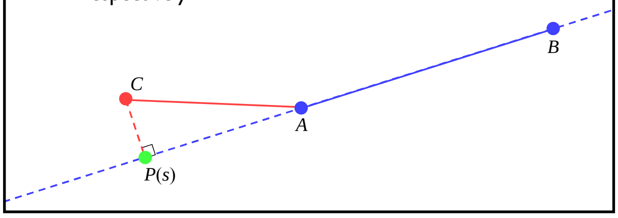
$$\Rightarrow s = \frac{\overline{AB} \cdot \overline{AC}}{|\overline{AB}|^2}$$



56

Special cases

- if $s<0$ or $s>1$ then the distance from point C to the line segment $\overline{AB}$ is not the same as the distance from point C to the infinite line $\overleftrightarrow{AB}$
- in these cases the distance is $|AC|$ or $|BC|$ respectively



57

Subdividing a Bezier cubic into two halves

- a Bezier cubic can be easily subdivided into two smaller Bezier cubics

$$Q_0 = P_0 \qquad R_0 = \tfrac{1}{8}P_0 + \tfrac{3}{8}P_1 + \tfrac{3}{8}P_2 + \tfrac{1}{8}P_3$$

$$Q_1 = \tfrac{1}{2}P_0 + \tfrac{1}{2}P_1 \qquad R_1 = \tfrac{1}{4}P_1 + \tfrac{1}{2}P_2 + \tfrac{1}{4}P_3$$

$$Q_2 = \tfrac{1}{4}P_0 + \tfrac{1}{2}P_1 + \tfrac{1}{4}P_2 \qquad R_2 = \tfrac{1}{2}P_2 + \tfrac{1}{2}P_3$$

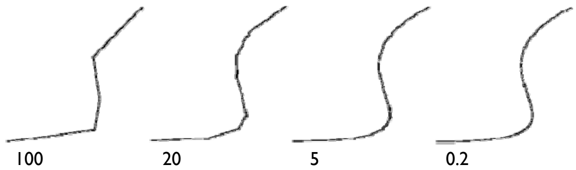
$$Q_3 = \tfrac{1}{8}P_0 + \tfrac{3}{8}P_1 + \tfrac{3}{8}P_2 + \tfrac{1}{8}P_3 \qquad R_3 = P_3$$

Exercise: prove that the Bezier cubic curves defined by Q_0, Q_1, Q_2, Q_3 and R_0, R_1, R_2, R_3 match the Bezier cubic curve defined by P_0, P_1, P_2, P_3 over the ranges $t\in[0,½]$ and $t\in[½,1]$ respectively

58

The effect of different tolerances

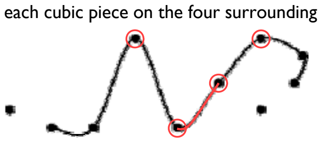
- this is the same Bezier curve drawn with four different tolerances



59

What if we have no tangent vectors?

- base each cubic piece on the four surrounding data points



- at each data point the curve must depend solely on the three surrounding data points   Why?
  - define the tangent at each point as the direction from the preceding point to the succeeding point
    - tangent at P_1 is $½(P_2 - P_0)$, at P_2 is $½(P_3 - P_1)$
- this is the basis of Overhauser's cubic

60

Overhauser's cubic

- method for generating Bezier curves which match Overhauser's model
  - simply calculate the appropriate Bezier control point locations from the given points
    - e.g. given points A, B, C, D, the Bezier control points are:
      P_0=B P_1=B+(C-A)/6
      P_3=C P_2=C-(D-B)/6
- Overhauser's cubic *interpolates* its controlling data points
  - good for control of movement in animation
  - not so good for industrial design because moving a single point modifies the surrounding four curve segments
    - compare with Bezier where moving a single point modifies just the two segments connected to that point

Overhauser worked for the Ford motor company in the 1960s

61

Simplifying line chains

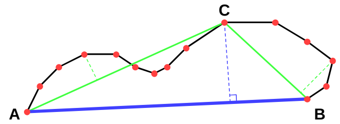
- this can be thought of as an inverse problem to the one of drawing Bezier curves
- problem specification: you are given a chain of line segments at a very high resolution, how can you reduce the number of line segments without compromising quality
  - e.g. given the coastline of Britain defined as a chain of line segments at one metre resolution, draw the entire outline on a 1280×1024 pixel screen
- the solution: Douglas & Pücker's line chain simplification algorithm

This can also be applied to chains of Bezier curves at high resolution: most of the curves will each be approximated (by the previous algorithm) as a single line segment, Douglas & Pücker's algorithm can then be used to further simplify the line chain

62

Douglas & Pücker's algorithm

- find point, C, at greatest distance from line segment AB
- if distance from C to AB is more than some specified tolerance then subdivide into AC and CB, repeat for each of the two subdivisions
- otherwise approximate entire chain from A to B by the single line segment AB



Exercises:
(1) How do you calculate the distance from C to AB?
(2) What special cases need to be considered? How should they be handled?

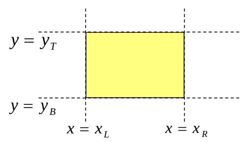
see slides 135 and 136

Douglas & Pücker, *Canadian Cartographer*, **10**(2), 1973

63

Clipping

- what about lines that go off the edge of the screen?
  - need to clip them so that we only draw the part of the line that is actually on the screen
- clipping points against a rectangle



need to check against four edges:

$$x = x_L$$
$$x = x_R$$
$$y = y_B$$
$$y = y_T$$

64

Clipping lines against a rectangle — naïvely

P_1 to $P_2 = (x_1, y_1)$ to (x_2, y_2)
$P(t) = (1-t)P_1 + tP_2$
$x(t) = (1-t)x_1 + tx_2$
$y(t) = (1-t)y_1 + ty_2$

- do this operation for each of the four edges

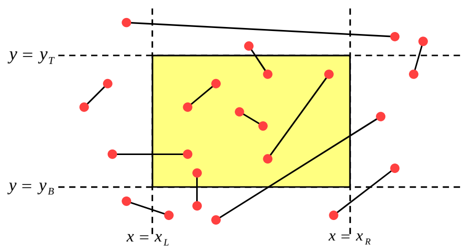
to intersect with $x = x_L$
if $(x_1 = x_2)$ then no intersection
else
$x_L = (1-t_L)x_1 + t_L x_2$
$\Rightarrow t_L = \dfrac{x_L - x_1}{x_2 - x_1}$
if $(0 \le t_L \le 1)$
then line segment intersects
$x = x_L$ at $(x(t_L), y(t_L))$
else line segment does not intersect edge

Exercise:

once you have the four intersection calculations, work out how to determine which bit of the line is actually inside the rectangle

65

Clipping lines against a rectangle — examples

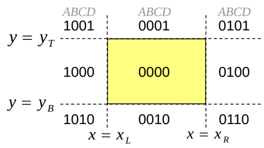


- you can naïvely check every line against each of the four edges
  - this works but is obviously inefficient
- adding a little cleverness improves efficiency enormously
  - Cohen-Sutherland clipping algorithm

66

Cohen-Sutherland clipper 1

- make a four bit code, one bit for each inequality
  $A \equiv x < x_L \quad B \equiv x > x_R \quad C \equiv y < y_B \quad D \equiv y > y_T$



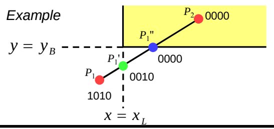
- evaluate this for both endpoints of the line
  $Q_1 = A_1B_1C_1D_1 \quad Q_2 = A_2B_2C_2D_2$

Ivan Sutherland is one of the founders of Evans & Sutherland, manufacturers of flight simulator systems

67

Cohen-Sutherland clipper 2

- $Q_1 = Q_2 = 0$
  - both ends in rectangle *ACCEPT*
- $Q_1 \wedge Q_2 \neq 0$
  - both ends outside and in same half-plane *REJECT*
- *otherwise*
  - need to intersect line with one of the edges and start again
    - you must always re-evaluate Q and recheck the above tests after doing a single clip
  - the 1 bits tell you which edge to clip against



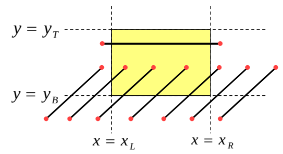
$x_1' = x_L \quad y_1' = y_1 + (y_2 - y_1)\frac{x_L - x_1}{x_2 - x_1}$

$y_1'' = y_B \quad x_1'' = x_1' + (x_2 - x_1')\frac{y_B - y_1'}{y_2 - y_1'}$

68

Cohen-Sutherland clipper 3

- if code has more than a single 1 then you cannot tell which is the best: simply select one and loop again
- horizontal and vertical lines are not a problem — Why not?
- need a line drawing algorithm that can cope with floating-point endpoint co-ordinates — Why?



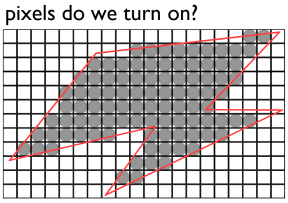
Exercise: what happens in each of the cases at left?
[Assume that, where there is a choice, the algorithm always tries to intersect with x_L or x_R before y_B or y_T.]

Try some other cases of your own devising.

69

Polygon filling

- which pixels do we turn on?



- those whose *centres* lie inside the polygon
  - this is a naïve assumption, but is sufficient for now

70

Scanline polygon fill algorithm

❶ take all polygon edges and place in an *edge list (EL)* , sorted on lowest *y* value

❷ start with the first scanline that intersects the polygon, get all edges which intersect that scan line and move them to an *active edge list (AEL)*

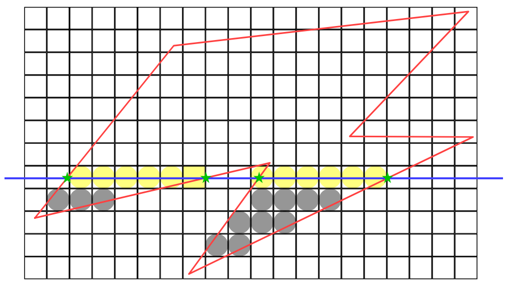
❸ for each edge in the AEL: find the intersection point with the current scanline; sort these into ascending order on the *x* value

❹ fill between pairs of intersection points

❺ move to the next scanline (increment *y*); remove edges from the AEL if endpoint < *y* ; move new edges from EL to AEL if start point ≤ *y*; if any edges remain in the AEL go back to step ❸

71

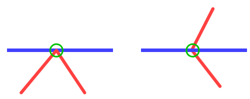
Scanline polygon fill example



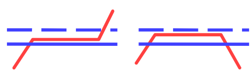
72

Scanline polygon fill details

- how do we efficiently calculate the intersection points?
  - use a line drawing algorithm to do incremental calculation
  - store current x value, increment value dx, starting and ending y values
  - on increment do a single addition x=x+dx
- what if endpoints exactly intersect scanlines?
  - need to ensure that the algorithm handles this properly

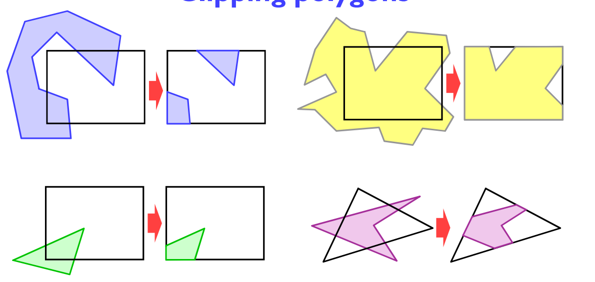


- what about horizontal edges?
  - can throw them out of the *edge list*, they contribute nothing



73

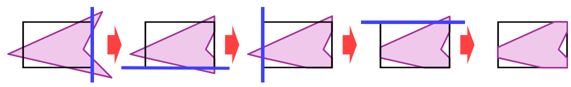
Clipping polygons



74

Sutherland-Hodgman polygon clipping 1

- clips an arbitrary polygon against an arbitrary *convex* polygon
  - basic algorithm clips an arbitrary polygon against a single infinite clip edge
    - so we reduce a complex algorithm to a simpler one which we call recursively
  - the polygon is clipped against one edge at a time, passing the result on to the next stage

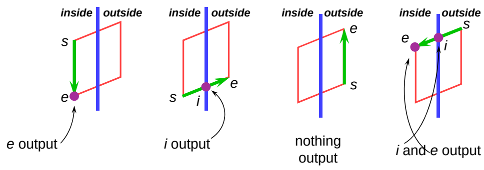


Sutherland & Hodgman, "Reentrant Polygon Clipping," *Comm. ACM*, **17**(1), 1974

75

Sutherland-Hodgman polygon clipping 2

- the algorithm progresses around the polygon checking if each edge crosses the clipping line and outputting the appropriate points

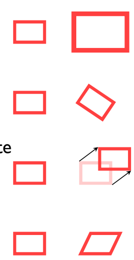


Exercise: the Sutherland-Hodgman algorithm may introduce new edges along the edge of the clipping polygon — when does this happen and why?

76

2D transformations

- scale
- rotate
- translate
- (shear)



- why?
  - it is extremely useful to be able to transform predefined objects to an arbitrary location, orientation, and size
  - any reasonable graphics package will include transforms
    - 2D ➔ Postscript
    - 3D ➔ OpenGL

77

Basic 2D transformations

- scale
  - about origin
  - by factor ***m***

$$x' = mx \qquad y' = my$$

- rotate
  - about origin
  - by angle θ

$$x' = x\cos\theta - y\sin\theta \qquad y' = x\sin\theta + y\cos\theta$$

- translate
  - along vector (x_o, y_o)

$$x' = x + x_o \qquad y' = y + y_o$$

- shear
  - parallel to *x* axis
  - by factor ***a***

$$x' = x + ay \qquad y' = y$$

78

Matrix representation of transformations

- scale
  - about origin, factor ***m***

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} m & 0 \\ 0 & m \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

- rotate
  - about origin, angle θ

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

- do nothing
  - identity

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

- shear
  - parallel to *x* axis, factor ***a***

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & a \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

79

Homogeneous 2D co-ordinates

- translations cannot be represented using simple 2D matrix multiplication on 2D vectors, so we switch to homogeneous co-ordinates

$$(x, y, w) \equiv \left(\tfrac{x}{w}, \tfrac{y}{w}\right)$$

- an infinite number of homogeneous co-ordinates map to every 2D point
- w=0 represents a point at infinity
- usually take the inverse transform to be:

$$(x, y) \equiv (x, y, 1)$$

80

Matrices in homogeneous co-ordinates

- scale
  - about origin, factor m

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} m & 0 & 0 \\ 0 & m & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

- rotate
  - about origin, angle θ

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

- do nothing
  - identity

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

- shear
  - parallel to x axis, factor a

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} 1 & a & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

81

Translation by matrix algebra

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} 1 & 0 & x_o \\ 0 & 1 & y_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

In homogeneous coordinates

$$x' = x + wx_o \qquad y' = y + wy_o \qquad w' = w$$

In conventional coordinates

$$\frac{x'}{w'} = \frac{x}{w} + x_0 \qquad \frac{y'}{w'} = \frac{y}{w} + y_0$$

82

Concatenating transformations

- often necessary to perform more than one transformation on the same object
- can concatenate transformations by multiplying their matrices e.g. a shear followed by a scaling:

scale

$$\begin{bmatrix} x'' \\ y'' \\ w'' \end{bmatrix} = \begin{bmatrix} m & 0 & 0 \\ 0 & m & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x' \\ y' \\ w' \end{bmatrix}$$

shear

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} 1 & a & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

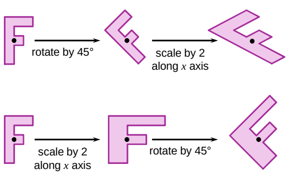
scale, shear, both

$$\begin{bmatrix} x'' \\ y'' \\ w'' \end{bmatrix} = \begin{bmatrix} m & 0 & 0 \\ 0 & m & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & a & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix} = \begin{bmatrix} m & ma & 0 \\ 0 & m & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

83

Concatenation is not commutative

- be careful of the order in which you concatenate transformations



rotate then scale

$$\begin{bmatrix} 2/\sqrt{2} & -2/\sqrt{2} & 0 \\ 1/\sqrt{2} & 1/\sqrt{2} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

scale

$$\begin{bmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 2/\sqrt{2} & -1/\sqrt{2} & 0 \\ 2/\sqrt{2} & 1/\sqrt{2} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

scale then rotate

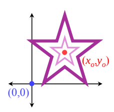
$$\begin{bmatrix} 1/\sqrt{2} & -1/\sqrt{2} & 0 \\ 1/\sqrt{2} & 1/\sqrt{2} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

rotate

84

Scaling about an arbitrary point

- scale by a factor m about point (x_o, y_o)
  1. translate point (x_o, y_o) to the origin
  2. scale by a factor m about the origin
  3. translate the origin to (x_o, y_o)



1. $$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} 1 & 0 & -x_o \\ 0 & 1 & -y_o \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

2. $$\begin{bmatrix} x'' \\ y'' \\ w'' \end{bmatrix} = \begin{bmatrix} m & 0 & 0 \\ 0 & m & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x' \\ y' \\ w' \end{bmatrix}$$

3. $$\begin{bmatrix} x''' \\ y''' \\ w''' \end{bmatrix} = \begin{bmatrix} 1 & 0 & x_o \\ 0 & 1 & y_o \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x'' \\ y'' \\ w'' \end{bmatrix}$$

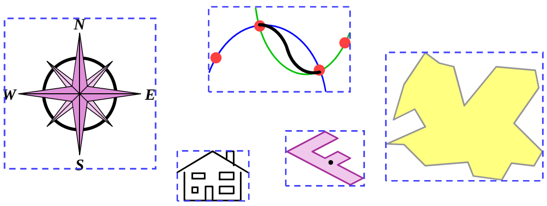
$$\begin{bmatrix} x''' \\ y''' \\ w''' \end{bmatrix} = \begin{bmatrix} 1 & 0 & x_o \\ 0 & 1 & y_o \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} m & 0 & 0 \\ 0 & m & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -x_o \\ 0 & 1 & -y_o \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

Exercise: show how to perform rotation about an arbitrary point

85

Bounding boxes

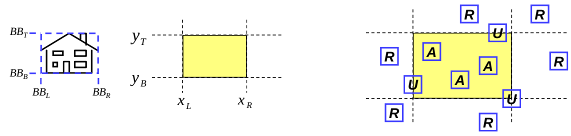
- when working with complex objects, bounding boxes can be used to speed up some operations



86

Clipping with bounding boxes

- do a quick *accept/reject/unsure* test to the bounding box then apply clipping to only the *unsure* objects



$BB_L > x_R \vee BB_R < x_L \vee BB_B > x_T \vee BB_T < x_B \Rightarrow$ REJECT

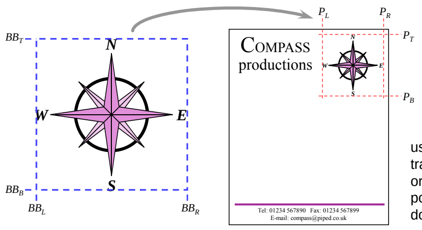
$BB_L \ge x_L \wedge BB_R \le x_R \wedge BB_B \ge x_B \wedge BB_T \le x_T \Rightarrow$ ACCEPT

otherwise ⇒ clip at next higher level of detail

87

Object inclusion with bounding boxes

- including one object (e.g. a graphics) file inside another can be easily done if bounding boxes are known and used



use the eight values to translate and scale the original to the appropriate position in the destination document

88

Bit block transfer (*BitBlT*)

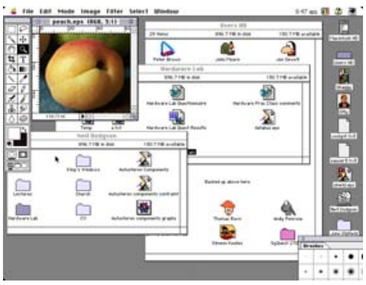
- it is sometimes preferable to predraw something and then copy the image to the correct position on the screen as and when required
  - e.g. icons
  - e.g. games

- copying an image from place to place is essentially a memory operation
  - can be made very fast
  - e.g. 32×32 pixel icon can be copied, say, 8 adjacent pixels at a time, if there is an appropriate memory copy operation

89

Application 1: user interface

- tend to use objects that are quick to draw
  - straight lines
  - filled rectangles
- complicated bits done using predrawn icons
- typefaces also tend to be predrawn



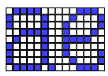
90

Application 2: typography

- typeface: a family of letters designed to look good together
  - usually has upright (roman/regular), italic (oblique), bold and bold-italic members

abcd *efgh* **ijkl** ***mnop*** – Gill Sans abcd *efgh* **ijkl** ***mnop*** – Times
abcd *efgh* **ijkl** ***mnop*** – Arial abcd *efgh* **ijkl** ***mnop*** – Garamond

- two forms of typeface used in computer graphics
  - pre-rendered bitmaps
    - single resolution (don't scale well)
    - use BitBlT to put into frame buffer
  - outline definitions
    - multi-resolution (can scale)
    - need to render (fill) to put into frame buffer



These notes are mainly set in Gill Sans, a lineale (sans-serif) typeface designed by Eric Gill for Monotype, 1928–30. The lowercase italic *p* is particularly interesting. Mathematics is mainly set in Times New Roman, a roman typeface commissioned by *The Times* in 1931, the design supervised by Stanley Morison.

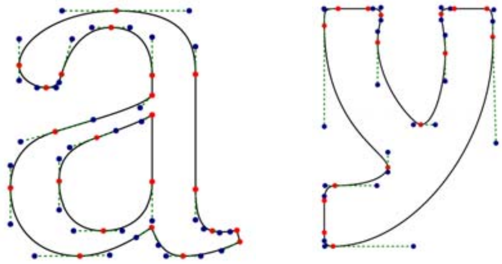
91

Application 3: Postscript

- industry standard rendering language for printers
- developed by Adobe Systems
- stack-based interpreted language
- basic features
  - object outlines made up of *lines*, *arcs* & *Bezier curves*
  - objects can be *filled* or *stroked*
  - whole range of 2D transformations can be applied to objects
  - typeface handling built in
    - typefaces are defined using Bezier curves
  - halftoning
  - can define your own functions in the language

92

Examples which are Bezier-friendly

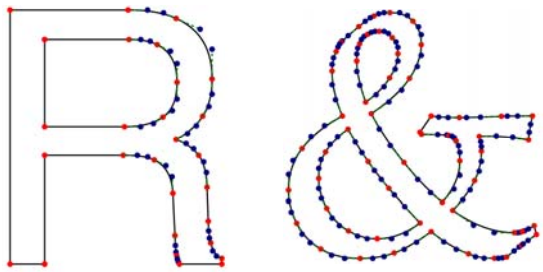


typeface: Utopia (1989)
designed as a Postscript typeface by Robert Slimbach at Adobe

typeface: Hobo (1910)
this typeface can be easily approximated by Beziers

93

Examples which are more fussy

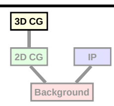


typeface: Helvetica (1957)
abcdQRST2345&

typeface: Palatino (1950)
abcdQRST2345&

94

3D Computer Graphics

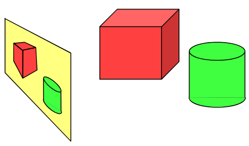


- 3D ⇨ 2D projection
- 3D versions of 2D operations
  - clipping, transforms, matrices, curves & surfaces
- 3D scan conversion
  - depth-sort, BSP tree, *z*-Buffer, A-buffer
- sampling
- lighting
- ray tracing

95

3D ⇨ 2D projection

- to make a picture
  - 3D world is projected to a 2D image
    - like a camera taking a photograph
    - the three dimensional world is projected onto a plane



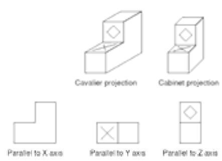
The 3D world is described as a set of (mathematical) objects

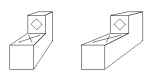
e.g. sphere	radius (3.4) centre (0,2,9)
e.g. box	size (2,4,3) centre (7, 2, 9) orientation (27º, 156º)

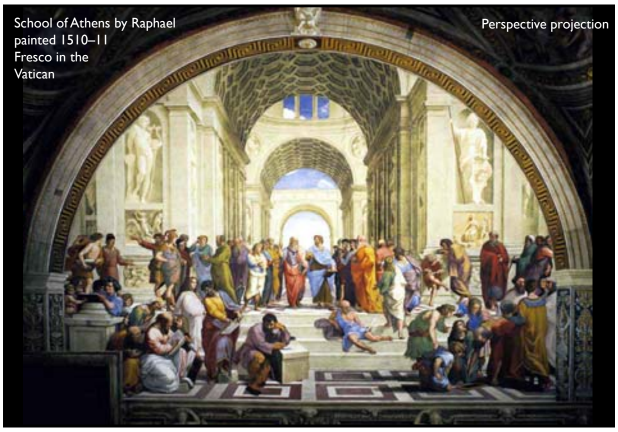
96

Types of projection

- parallel
  - e.g. $(x, y, z) \rightarrow (x, y)$
  - useful in CAD, architecture, etc
  - looks unrealistic
- perspective
  - e.g. $(x, y, z) \rightarrow (\frac{x}{z}, \frac{y}{z})$
  - things get smaller as they get farther away
  - looks realistic
    - this is how cameras work





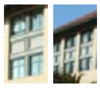


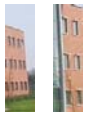
98

Perspective projection examples

Gates Building – the rounded version (Stanford)

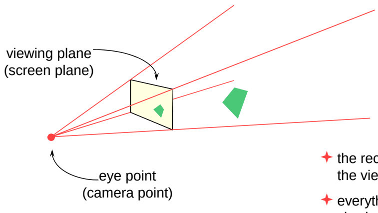
Gates Building – the rectilinear version (Cambridge)





99

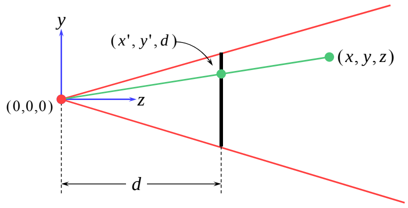
Viewing volume



- the rectangular pyramid is the viewing volume
- everything within the viewing volume is projected onto the viewing plane

100

Geometry of perspective projection



$$x' = x\frac{d}{z}$$

$$y' = y\frac{d}{z}$$

101

Perspective projection with an arbitrary camera

- we have assumed that:
  - screen centre at **(0,0,d)**
  - screen parallel to *xy*-plane
  - *z*-axis into screen
  - *y*-axis up and *x*-axis to the right
  - eye (camera) at origin **(0,0,0)**
- for an arbitrary camera we can either:
  - work out equations for projecting objects about an arbitrary point onto an arbitrary plane
  - transform all objects into our standard co-ordinate system (viewing co-ordinates) and use the above assumptions

102

3D transformations

- 3D homogeneous co-ordinates
  $(x, y, z, w) \rightarrow (\frac{x}{w}, \frac{y}{w}, \frac{z}{w})$
- 3D transformation matrices

translation
$$\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

identity
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

rotation about *x*-axis
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

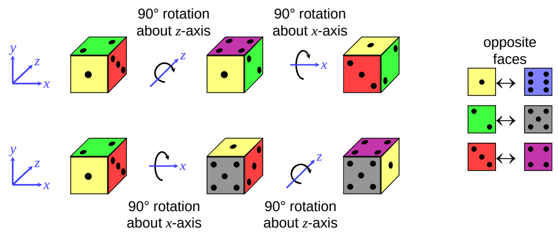
scale
$$\begin{bmatrix} m_x & 0 & 0 & 0 \\ 0 & m_y & 0 & 0 \\ 0 & 0 & m_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

rotation about *z*-axis
$$\begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

rotation about *y*-axis
$$\begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

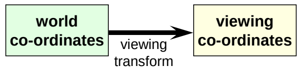
103

3D transformations are not commutative

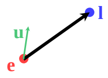


104

Viewing transform 1



- the problem:
  - to transform an arbitrary co-ordinate system to the default viewing co-ordinate system
- camera specification in world co-ordinates
  - eye (camera) at (e_x,e_y,e_z)
  - look point (centre of screen) at (l_x,l_y,l_z)
  - up along vector (u_x,u_y,u_z)
    - perpendicular to $\overline{\mathbf{el}}$



105

Viewing transform 2

- translate eye point, (e_x,e_y,e_z), to origin, $(0,0,0)$

$$\mathbf{T}=\begin{bmatrix}1 & 0 & 0 & -e_x\\ 0 & 1 & 0 & -e_y\\ 0 & 0 & 1 & -e_z\\ 0 & 0 & 0 & 1\end{bmatrix}$$

- scale so that eye point to look point distance, $|\overline{\mathbf{el}}|$, is distance from origin to screen centre, d

$$|\overline{\mathbf{el}}|=\sqrt{(l_x-e_x)^2+(l_y-e_y)^2+(l_z-e_z)^2} \qquad \mathbf{S}=\begin{bmatrix}d/|\overline{\mathbf{el}}| & 0 & 0 & 0\\ 0 & d/|\overline{\mathbf{el}}| & 0 & 0\\ 0 & 0 & d/|\overline{\mathbf{el}}| & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$$

106

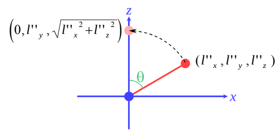
Viewing transform 3

- need to align line $\overline{\mathbf{el}}$ with z-axis
  - first transform e and l into new co-ordinate system

    $\mathbf{e}''=\mathbf{S}\times\mathbf{T}\times\mathbf{e}=\mathbf{0} \qquad \mathbf{l}''=\mathbf{S}\times\mathbf{T}\times\mathbf{l}$
  - then rotate e''l'' into yz-plane, rotating about y-axis

$$\mathbf{R}_1=\begin{bmatrix}\cos\theta & 0 & -\sin\theta & 0\\ 0 & 1 & 0 & 0\\ \sin\theta & 0 & \cos\theta & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$$

$$\theta=\arccos\frac{l''_z}{\sqrt{l''^2_x+l''^2_z}}$$



107

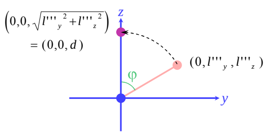
Viewing transform 4

- having rotated the viewing vector onto the yz plane, rotate it about the x-axis so that it aligns with the z-axis

$$\mathbf{l}'''=\mathbf{R}_1\times\mathbf{l}''$$

$$\mathbf{R}_2=\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & \cos\varphi & -\sin\varphi & 0\\ 0 & \sin\varphi & \cos\varphi & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$$

$$\varphi=\arccos\frac{l'''_z}{\sqrt{l'''^2_y+l'''^2_z}}$$



108

Viewing transform 5

- the final step is to ensure that the up vector actually points up, i.e. along the positive y-axis
  - actually need to rotate the up vector about the z-axis so that it lies in the positive y half of the yz plane

$$\mathbf{u}''''=\mathbf{R}_2\times\mathbf{R}_1\times\mathbf{u}$$

$$\mathbf{R}_3=\begin{bmatrix}\cos\psi & -\sin\psi & 0 & 0\\ \sin\psi & \cos\psi & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$$

$$\psi=\arccos\frac{u''''_y}{\sqrt{u''''^2_x+u''''^2_y}}$$

why don't we need to multiply **u** by **S** or **T**?

u is a vector rather than a point, vectors do not get translated

scaling **u** by a uniform scaling matrix would make no difference to the direction in which it points

109

Viewing transform 6

world co-ordinates → (viewing transform) → viewing co-ordinates

- we can now transform any point in world co-ordinates to the equivalent point in viewing co-ordinate

$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \mathbf{R}_3 \times \mathbf{R}_2 \times \mathbf{R}_1 \times \mathbf{S} \times \mathbf{T} \times \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

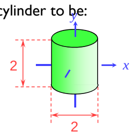
- in particular: $\mathbf{e} \rightarrow (0,0,0) \quad \mathbf{l} \rightarrow (0,0,d)$
- the matrices depend only on e, l, and u, so they can be pre-multiplied together

$$\mathbf{M} = \mathbf{R}_3 \times \mathbf{R}_2 \times \mathbf{R}_1 \times \mathbf{S} \times \mathbf{T}$$

110

Another transformation example

- a well known graphics package (Open Inventor) defines a cylinder to be:
  - **centre at the origin, (0,0,0)**
  - **radius 1 unit**
  - **height 2 units, aligned along the *y*-axis**



- this is the only cylinder that can be drawn, *but* the package has a complete set of 3D transformations
- we want to draw a cylinder of:
  - **radius 2 units**
  - **the centres of its two ends located at (1,2,3) and (2,4,5)**
    - its length is thus 3 units
- what transforms are required? and in what order should they be applied?

111

A variety of transformations

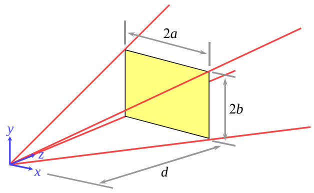
object in **object** co-ordinates → (modelling transform) → object in **world** co-ordinates → (viewing transform) → object in **viewing** co-ordinates → (projection) → object in ***2D screen*** co-ordinates

- the modelling transform and viewing transform can be multiplied together to produce a single matrix taking an object directly from object co-ordinates into viewing co-ordinates
- either or both of the modelling transform and viewing transform matrices can be the identity matrix
  - e.g. objects can be specified directly in viewing co-ordinates, or directly in world co-ordinates
- this is a useful set of transforms, not a hard and fast model of how things should be done

112

Clipping in 3D

- clipping against a volume in viewing co-ordinates



a point (x,y,z) can be clipped against the pyramid by checking it against four planes:

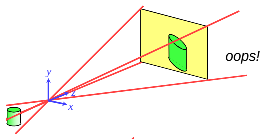
$$x > -z\frac{a}{d} \qquad x < z\frac{a}{d}$$

$$y > -z\frac{b}{d} \qquad y < z\frac{b}{d}$$

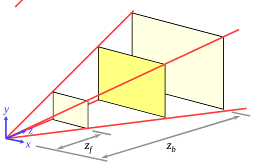
113

What about clipping in *z*?

- need to at least check for $z < 0$ to stop things behind the camera from projecting onto the screen



- can also have front and back clipping planes: $z > z_f$ and $z < z_b$
  - resulting clipping volume is called the *viewing frustum*



114

Clipping in 3D — two methods

which is best?

- clip against the viewing frustum
  - need to clip against six planes

$$x = -z\frac{a}{d} \quad x = z\frac{a}{d} \quad y = -z\frac{b}{d} \quad y = z\frac{b}{d} \quad z = z_f \quad z = z_b$$

- project to 2D (retaining *z*) and clip against the axis-aligned cuboid
  - still need to clip against six planes

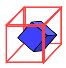
$$x = -a \quad x = a \quad y = -b \quad y = b \quad z = z_f \quad z = z_b$$

    - these are simpler planes against which to clip
    - this is equivalent to clipping in 2D with two extra clips for *z*

115

Bounding volumes & clipping

- can be very useful for reducing the amount of work involved in clipping
- what kind of bounding volume?
  - axis aligned box



  - sphere

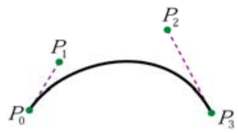
- can have multiple levels of bounding volume

116

Curves in 3D

- same as curves in 2D, with an extra co-ordinate for each point
- e.g. Bezier cubic in 3D:

$$P(t) = (1-t)^3 P_0 + 3t(1-t)^2 P_1 + 3t^2(1-t)P_2 + t^3 P_3$$

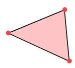


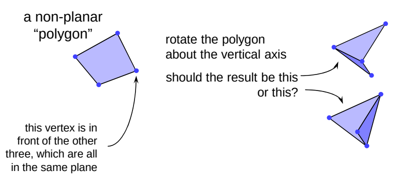
where: $P_i \equiv (x_i, y_i, z_i)$

117

Surfaces in 3D: polygons

- lines generalise to planar polygons
  - 3 vertices (triangle) must be planar
  - > 3 vertices, not necessarily planar

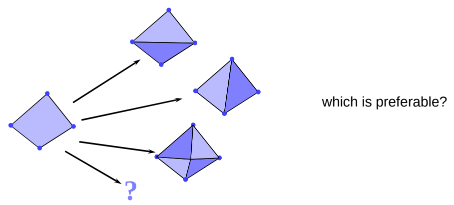




118

Splitting polygons into triangles

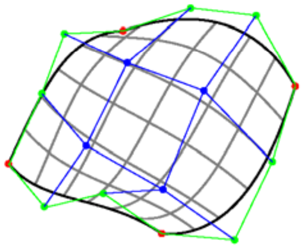
- some graphics processors accept only triangles
- an arbitrary polygon with more than three vertices isn't guaranteed to be planar; a triangle is



119

Surfaces in 3D: patches

- curves generalise to patches
  - a Bezier patch has a Bezier curve running along each of its four edges and four extra internal control points



120

Bezier patch definition

- the Bezier patch defined by the sixteen control points, $P_{0,0}, P_{0,1}, \ldots, P_{3,3}$, is:

$$P(s,t) = \sum_{i=0}^{3} \sum_{j=0}^{3} b_i(s) b_j(t) P_{i,j}$$

where: $b_0(t) = (1-t)^3 \quad b_1(t) = 3t(1-t)^2 \quad b_2(t) = 3t^2(1-t) \quad b_3(t) = t^3$

- compare this with the 2D version:

$$P(t) = \sum_{i=0}^{3} b_i(t) P_i$$

121

Continuity between Bezier patches

- each patch is smooth within itself
- ensuring continuity in 3D:
  - C_0 – continuous in position
    - the four edge control points must match
  - C_1 – continuous in both position and tangent vector
    - the four edge control points must match
    - the two control points on either side of each of the four edge control points must be co-linear with both the edge point and each another and be equidistant from the edge point

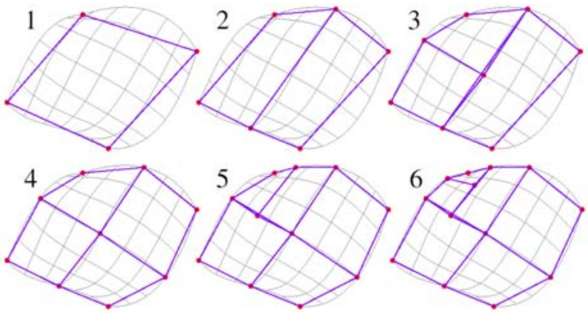
122

Drawing Bezier patches

- in a similar fashion to Bezier curves, Bezier patches can be drawn by approximating them with planar polygons
- simple method
  - select appropriate increments in s and t and render the resulting quadrilaterals
- tolerance-based method
  - check if the Bezier patch is sufficiently well approximated by a quadrilateral, if so use that quadrilateral
  - if not then subdivide it into two smaller Bezier patches and repeat on each
    - subdivide in different dimensions on alternate calls to the subdivision function
  - having approximated the whole Bezier patch as a set of (non-planar) quadrilaterals, further subdivide these into (planar) triangles
    - be careful to not leave any gaps in the resulting surface!

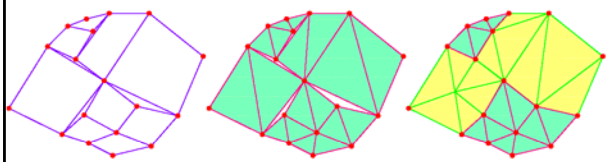
123

Subdividing a Bezier patch — example



124

Triangulating the subdivided patch



Final quadrilateral mesh
Naïve triangulation
More intelligent triangulation

- need to be careful not to generate holes
- need to be equally careful when subdividing connected patches

125

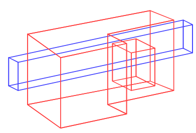
3D scan conversion

- lines
- polygons
  - depth sort
  - Binary Space-Partitioning tree
  - z-buffer
  - A-buffer
- ray tracing

126

3D line drawing

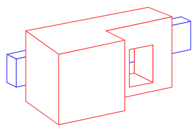
- given a list of 3D lines we draw them by:
  - projecting end points onto the 2D screen
  - using a line drawing algorithm on the resulting 2D lines
- this produces a wireframe version of whatever objects are represented by the lines



127

Hidden line removal

- by careful use of cunning algorithms, lines that are hidden by surfaces can be carefully removed from the projected version of the objects
  - still just a line drawing
  - will not be covered further in this course



128

3D polygon drawing

- given a list of 3D polygons we draw them by:
  - projecting vertices onto the 2D screen
    - but also keep the *z* information
  - using a 2D polygon scan conversion algorithm on the resulting 2D polygons
- in what order do we draw the polygons?
  - some sort of order on *z*
    - depth sort
    - Binary Space-Partitioning tree
- is there a method in which order does not matter?
    - z-buffer

129

Depth sort algorithm

❶ transform all polygon vertices into viewing co-ordinates and project these into 2D, keeping *z* information

❷ calculate a depth ordering for polygons, based on the most distant *z* co-ordinate in each polygon

❸ resolve any ambiguities caused by polygons overlapping in *z*

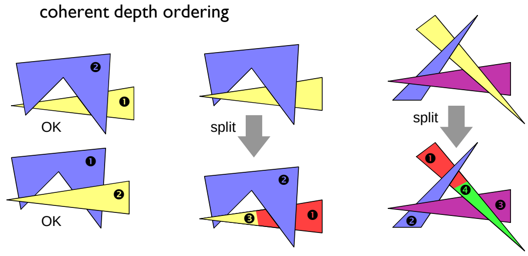
❹ draw the polygons in depth order from back to front
  - "painter's algorithm": later polygons draw on top of earlier polygons

- steps ❶ and ❷ are simple, step ❹ is 2D polygon scan conversion, step ❸ requires more thought

130

Resolving ambiguities in depth sort

- may need to split polygons into smaller polygons to make a coherent depth ordering



131

Resolving ambiguities: algorithm

- for the rearmost polygon, *P*, in the list, need to compare *each* polygon, *Q*, which overlaps *P* in *z*
  - the question is: can I draw *P* before *Q*?
    - ❶ do the polygons *y* extents not overlap?
    - ❷ do the polygons *x* extents not overlap?
    - ❸ is *P* entirely on the opposite side of *Q*'s plane from the viewpoint?
    - ❹ is *Q* entirely on the same side of *P*'s plane as the viewpoint?

    (tests get more expensive)
  - if all 4 tests fail, repeat ❸ and ❹ with *P* and *Q* swapped (i.e. can I draw *Q* before *P*?), if true swap *P* and *Q*
  - otherwise split either *P* or *Q* by the plane of the other, throw away the original polygon and insert the two pieces into the list
- draw rearmost polygon once it has been completely checked

132

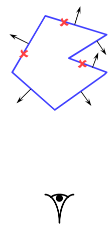
Depth sort: comments

- the depth sort algorithm produces a list of polygons which can be scan-converted in 2D, backmost to frontmost, to produce the correct image
- reasonably cheap for small number of polygons, becomes expensive for large numbers of polygons
- the ordering is only valid from one particular viewpoint

133

Back face culling: a time-saving trick

- if a polygon is a face of a closed polyhedron *and* faces backwards with respect to the viewpoint *then* it need not be drawn at all because front facing faces would later obscure it anyway
  - saves drawing time at the the cost of one extra test per polygon
  - assumes that we know which way a polygon is oriented
- back face culling can be used in combination with any 3D scan-conversion algorithm



134

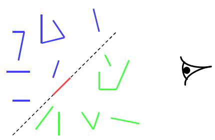
Binary Space-Partitioning trees

- BSP trees provide a way of quickly calculating the correct depth order:
  - for a collection of static polygons
  - from an arbitrary viewpoint
- the BSP tree trades off an initial time- and space-intensive pre-processing step against a linear display algorithm ($O(N)$) which is executed whenever a new viewpoint is specified
- the BSP tree allows you to easily determine the correct order in which to draw polygons by traversing the tree in a simple way

135

BSP tree: basic idea

- a given polygon will be correctly scan-converted if:
  - all polygons on the far side of it from the viewer are scan-converted first
  - then it is scan-converted
  - then all the polygons on the near side of it are scan-converted



136

Making a BSP tree

- given a set of polygons
  - select an arbitrary polygon as the root of the tree
  - divide all remaining polygons into two subsets:
    - those in front of the selected polygon's plane
    - those behind the selected polygon's plane
    - any polygons through which the plane passes are split into two polygons and the two parts put into the appropriate subsets
  - make two BSP trees, one from each of the two subsets
    - these become the front and back subtrees of the root
- may be advisable to make, say, 20 trees with different random roots to be sure of getting a tree that is reasonably well balanced

137

Drawing a BSP tree

- if the viewpoint is in front of the root's polygon's plane then:
  - draw the BSP tree for the *back* child of the root
  - draw the root's polygon
  - draw the BSP tree for the *front* child of the root
- otherwise:
  - draw the BSP tree for the *front* child of the root
  - draw the root's polygon
  - draw the BSP tree for the *back* child of the root

138

Scan-line algorithms

- instead of drawing one polygon at a time:
  modify the 2D polygon scan-conversion algorithm to handle all of the polygons at once
- the algorithm keeps a list of the active edges in all polygons and proceeds one scan-line at a time
  - there is thus one large *active edge list* and one (even larger) *edge list*
    - *enormous memory requirements*
- still fill in pixels between adjacent pairs of edges on the scan-line but:
  - need to be intelligent about which polygon is in front
    and therefore what colours to put in the pixels
  - every edge is used in two pairs:
    one to the left and one to the right of it

139

z-buffer polygon scan conversion

- depth sort & BSP-tree methods involve clever sorting algorithms followed by the invocation of the standard 2D polygon scan conversion algorithm
- by modifying the 2D scan conversion algorithm we can remove the need to sort the polygons
  - makes hardware implementation easier

140

z-buffer basics

- store both *colour* and *depth* at each pixel
- when scan converting a polygon:
  - calculate the polygon's depth at each pixel
  - if the polygon is closer than the current depth stored at that pixel
    - then store both the polygon's colour and depth at that pixel
    - otherwise do nothing

141

z-buffer algorithm

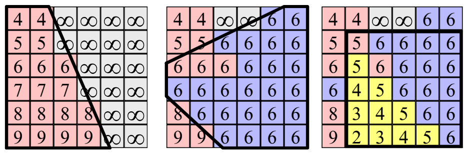
```
FOR every pixel (x,y)
    Colour[x,y] = background colour ;
    Depth[x,y] = infinity ;
END FOR ;

FOR each polygon
    FOR every pixel (x,y) in the polygon's projection
        z = polygon's z-value at pixel (x,y) ;
        IF z < Depth[x,y] THEN
            Depth[x,y] = z ;
            Colour[x,y] = polygon's colour at (x,y) ;
        END IF ;
    END FOR ;
END FOR ;
```

This is essentially the 2D polygon scan conversion algorithm with depth calculation and depth comparison added.

142

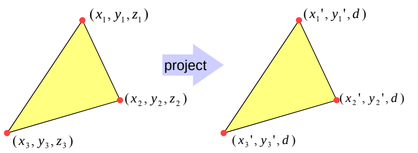
z-buffer example



143

Interpolating depth values 1

- just as we incrementally interpolate *x* as we move down the edges of the polygon, we can incrementally interpolate *z*:
  - as we move down the edges of the polygon
  - as we move across the polygon's projection



$$x_a' = x_a \frac{d}{z_a}$$

$$y_a' = y_a \frac{d}{z_a}$$

144

Interpolating depth values 2

- we thus have 2D vertices, with added depth information

$$[(x_a', y_a'), z_a]$$

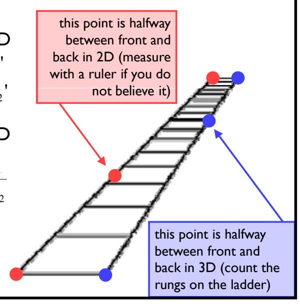
- we can interpolate *x* and *y* in 2D

$$x' = (1-t)x_1' + (t)x_2'$$

$$y' = (1-t)y_1' + (t)y_2'$$

- but *z* must be interpolated in 3D

$$\frac{1}{z} = (1-t)\frac{1}{z_1} + (t)\frac{1}{z_2}$$



145

Comparison of methods

Algorithm	Complexity	Notes
Depth sort	O(N log N)	Need to resolve ambiguities
Scan line	O(N log N)	Memory intensive
BSP tree	O(N)	O(N log N) pre-processing step
z-buffer	O(N)	Easy to implement in hardware

- BSP is only useful for scenes which do not change
- as number of polygons increases, average size of polygon decreases, so time to draw a single polygon decreases
- z-buffer easy to implement in hardware: simply give it polygons in any order you like
- other algorithms need to know about all the polygons before drawing a single one, so that they can sort them into order

146

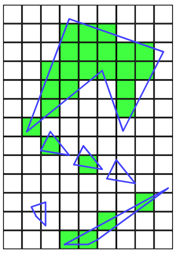
Putting it all together - a summary

- a 3D polygon scan conversion algorithm needs to include:
  - a 2D polygon scan conversion algorithm
  - 2D or 3D polygon clipping
  - projection from 3D to 2D
  - some method of ordering the polygons so that they are drawn in the correct order

147

Sampling

- all of the methods so far take a single sample for each pixel at the precise centre of the pixel
  - i.e. the value for each pixel is the colour of the polygon which happens to lie exactly under the centre of the pixel
- this leads to:
  - stair step (jagged) edges to polygons
  - small polygons being missed completely
  - thin polygons being missed completely or split into small pieces



148

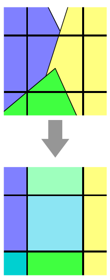
Anti-aliasing

- these artefacts (and others) are jointly known as aliasing
- methods of ameliorating the effects of aliasing are known as *anti-aliasing*
  - in signal processing *aliasing* is a precisely defined technical term for a particular kind of artefact
  - in computer graphics its meaning has expanded to include most undesirable effects that can occur in the image
    - this is because the same anti-aliasing techniques which ameliorate true aliasing artefacts also ameliorate most of the other artefacts

149

Anti-aliasing method 1: area averaging

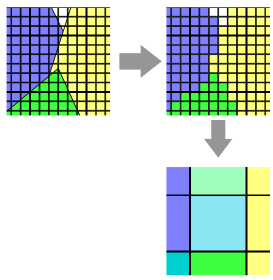
- average the contributions of all polygons to each pixel
  - e.g. assume pixels are square and we just want the average colour in the square
  - Ed Catmull developed an algorithm which does this:
    - works a scan-line at a time
    - clips all polygons to the scan-line
    - determines the fragment of each polygon which projects to each pixel
    - determines the amount of the pixel covered by the visible part of each fragment
    - pixel's colour is a weighted sum of the visible parts
  - expensive algorithm!



150

Anti-aliasing method 2: super-sampling

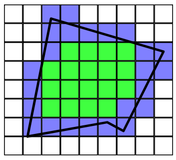
- sample on a finer grid, then average the samples in each pixel to produce the final colour
  - for an $n \times n$ sub-pixel grid, the algorithm would take roughly n^2 times as long as just taking one sample per pixel
- can simply average all of the sub-pixels in a pixel or can do some sort of weighted average



151

The A-buffer

- a significant modification of the z-buffer, which allows for sub-pixel sampling without as high an overhead as straightforward super-sampling
- basic observation:
  - a given polygon will cover a pixel:
    - totally
    - partially
    - not at all
  - sub-pixel sampling is only required in the case of pixels which are partially covered by the polygon



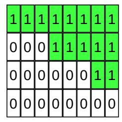
L. Carpenter, "The A-buffer: an antialiased hidden surface method", SIGGRAPH 84, 103–8

152

A-buffer: details

- for each pixel, a list of masks is stored
- each mask shows how much of a polygon covers the pixel
- the masks are sorted in depth order
- a mask is a 4×8 array of bits:

{ need to store both colour and depth in addition to the mask



1 = polygon covers this sub-pixel

0 = polygon doesn't cover this sub-pixel

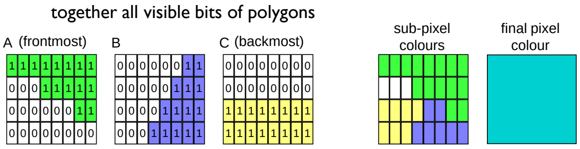
sampling is done at the centre of each of the sub-pixels

The use of 4×8 bits is because of the original architecture on which this was implemented. You could use any number of sub-pixels: a power of 2 is obviously sensible.

153

A-buffer: example

- to get the final colour of the pixel you need to average together all visible bits of polygons



A=11111111 00011111 00000011 00000000
B=00000011 00000111 00001111 00011111
C=00000000 00000000 11111111 11111111

¬A∧B =00000000 00000000 00001100 00011111
¬A∧¬B∧C =00000000 00000000 11110000 11100000

A covers 15/32 of the pixel
¬A∧B covers 7/32 of the pixel
¬A∧¬B∧C covers 7/32 of the pixel

154

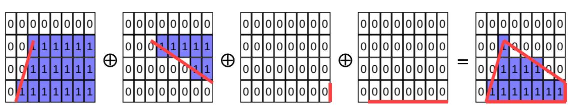
Making the A-buffer more efficient

- if a polygon *totally* covers a pixel then:
  - do not need to calculate a mask, because the mask is all 1s
  - all masks currently in the list which are *behind* this polygon can be discarded
  - any subsequent polygons which are behind this polygon can be immediately discounted (without calculating a mask)
- in most scenes, therefore, the majority of pixels will have only a single entry in their list of masks
- the polygon scan-conversion algorithm can be structured so that it is immediately obvious whether a pixel is *totally* or *partially* within a polygon

155

A-buffer: calculating masks

- clip polygon to pixel
- calculate the mask for each edge bounded by the right hand side of the pixel
  - there are few enough of these that they can be stored in a look-up table
- XOR all masks together



156

A-buffer: comments

- the A-buffer algorithm essentially adds anti-aliasing to the *z*-buffer algorithm in an efficient way
- most operations on masks are AND, OR, NOT, XOR
  - very efficient boolean operations
- why 4×8?
  - algorithm originally implemented on a machine with 32-bit registers (VAX 11/780)
  - on a 64-bit register machine, 8×8 is more sensible
- what does the A stand for in A-buffer?
  - anti-aliased, area averaged, accumulator

157

A-buffer: extensions

- as presented the algorithm assumes that a mask has a constant depth (*z* value)
  - can modify the algorithm and perform approximate intersection between polygons
- can save memory by combining fragments which start life in the same primitive
  - e.g. two triangles that are part of the decomposition of a Bezier patch
- can extend to allow transparent objects

158

Illumination & shading

- until now we have assumed that each polygon is a uniform colour and have not thought about how that colour is determined
- things look more realistic if there is some sort of illumination in the scene
- we therefore need a mechanism of determining the colour of a polygon based on its surface properties and the positions of the lights
- we will, as a consequence, need to find ways to shade polygons which do not have a uniform colour

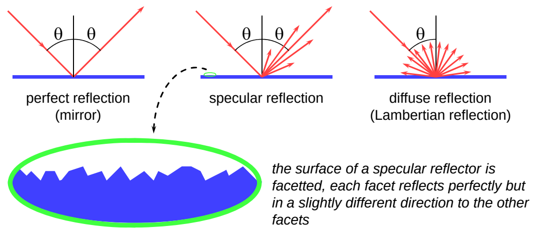
159

Illumination & shading (continued)

- in the real world every light source emits millions of photons every second
- these photons bounce off objects, pass through objects, and are absorbed by objects
- a tiny proportion of these photons enter your eyes (or the camera) allowing you to see the objects
- tracing the paths of all these photons is not an efficient way of calculating the shading on the polygons in your scene

160

How do surfaces reflect light?

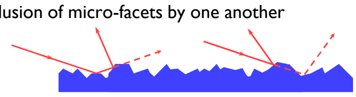


Johann Lambert, 18th century German mathematician

161

Comments on reflection

- the surface can absorb some wavelengths of light
  - e.g. shiny gold or shiny copper
- specular reflection has "interesting" properties at glancing angles owing to occlusion of micro-facets by one another



- plastics are good examples of surfaces with:
  - specular reflection in the light's colour
  - diffuse reflection in the plastic's colour

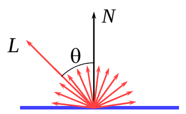
162

Calculating the shading of a polygon

- gross assumptions:
  - there is only diffuse (Lambertian) reflection
  - all light falling on a polygon comes directly from a light source
    - there is no interaction between polygons
  - no polygon casts shadows on any other
    - so can treat each polygon as if it were the only polygon in the scene
  - light sources are considered to be infinitely distant from the polygon
    - the vector to the light is the same across the whole polygon
- observation:
  - the colour of a flat polygon will be uniform across its surface, dependent only on the colour & position of the polygon and the colour & position of the light sources

163

Diffuse shading calculation



$$I = I_l k_d \cos\theta$$
$$= I_l k_d (N \cdot L)$$

L is a normalised vector pointing in the direction of the light source

N is the normal to the polygon

I_l is the intensity of the light source

k_d is the proportion of light which is diffusely reflected by the surface

I is the intensity of the light reflected by the surface

use this equation to set the colour of the whole polygon and draw the polygon using a standard polygon scan-conversion routine

164

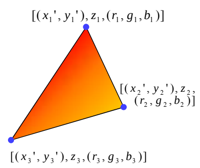
Diffuse shading: comments

- can have different I_l and different k_d for different wavelengths (colours)
- watch out for $\cos\theta < 0$
  - implies that the light is behind the polygon and so it cannot illuminate this side of the polygon
- do you use one-sided or two-sided polygons?
  - one sided: only the side in the direction of the normal vector can be illuminated
    - if $\cos\theta < 0$ then both sides are black
  - two sided: the sign of $\cos\theta$ determines which side of the polygon is illuminated
    - need to invert the sign of the intensity for the back side

165

Gouraud shading

- for a polygonal model, calculate the diffuse illumination at each vertex rather than for each polygon
  - calculate the normal at the vertex, and use this to calculate the diffuse illumination at that point
  - normal can be calculated directly if the polygonal model was derived from a curved surface
- interpolate the colour across the polygon, in a similar manner to that used to interpolate *z*
- surface will look smoothly curved
  - rather than looking like a set of polygons
  - surface outline will still look polygonal

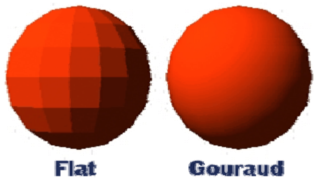


Henri Gouraud, "Continuous Shading of Curved Surfaces", *IEEE Trans Computers*, **20**(6), 1971

166

Flat *vs* Gouraud shading

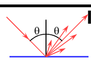
- note how the interior is smoothly shaded but the outline remains polygonal



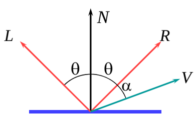
http://computer.howstuffworks.com/question484.htm

167

Specular reflection



- Phong developed an easy-to-calculate *approximation* to specular reflection



$$I = I_l k_s \cos^n \alpha$$
$$= I_l k_s (R \cdot V)^n$$

L is a normalised vector pointing in the direction of the light source

R is the vector of perfect reflection

N is the normal to the polygon

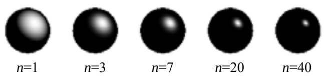
V is a normalised vector pointing at the viewer

I_l is the intensity of the light source

k_s is the proportion of light which is specularly reflected by the surface

n is Phong's *ad hoc* "roughness" coefficient

I is the intensity of the specularly reflected light

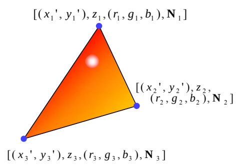


Phong Bui-Tuong, "Illumination for computer generated pictures", *CACM*, **18**(6), 1975, 311–7

168

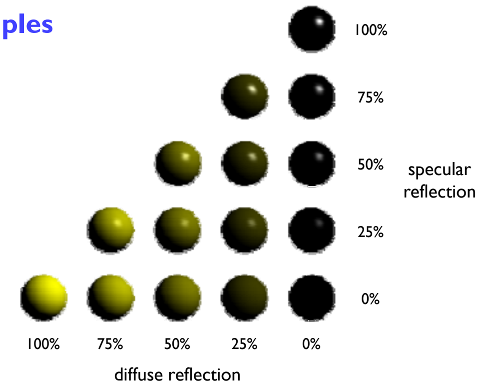
Phong shading

- similar to Gouraud shading, but calculate the specular component in addition to the diffuse component
- therefore need to interpolate the *normal* across the polygon in order to be able to calculate the reflection vector
- N.B. Phong's approximation to specular reflection ignores (amongst other things) the effects of glancing incidence



169

Examples



170

The gross assumptions revisited

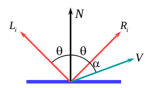
- only diffuse reflection
  - now have a method of approximating specular reflection
- no shadows
  - need to do ray tracing to get shadows
- lights at infinity
  - can add local lights at the expense of more calculation
    - need to interpolate the L vector
- no interaction between surfaces
  - cheat!
    - assume that all light reflected off all other surfaces onto a given polygon can be amalgamated into a single constant term: "ambient illumination", add this onto the diffuse and specular illumination

171

Shading: overall equation

- the overall shading equation can thus be considered to be the ambient illumination plus the diffuse and specular reflections from each light source

$$I = I_a k_a + \sum_i I_i k_d (L_i \cdot N) + \sum_i I_i k_s (R_i \cdot V)^n$$



  - the more lights there are in the scene, the longer this calculation will take

172

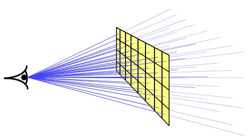
Illumination & shading: comments

- how good is this shading equation?
  - gives reasonable results but most objects tend to look as if they are made out of plastic
  - Cook & Torrance have developed a more realistic (and more expensive) shading model which takes into account:
    - micro-facet geometry (which models, amongst other things, the roughness of the surface)
    - Fresnel's formulas for reflectance off a surface
  - there are other, even more complex, models
- is there a better way to handle inter-object interaction?
  - "ambient illumination" is, frankly, a gross approximation
  - distributed ray tracing can handle specular inter-reflection
  - radiosity can handle diffuse inter-reflection

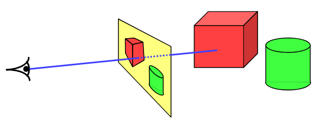
173

Ray tracing

- a powerful alternative to polygon scan-conversion techniques
- given a set of 3D objects, shoot a ray from the eye through the centre of every pixel and see what it hits



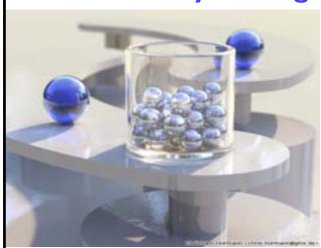
shoot a ray through each pixel



whatever the ray hits determines the colour of that pixel

174

Ray tracing: examples



ray tracing easily handles reflection, refraction, shadows and blur: all of which are difficult to achieve with polygon scan-conversion

ray tracing is considerably slower than polygon scan-conversion

175

Ray tracing algorithm

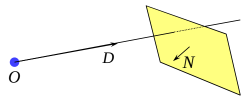
```
select an eye point and a screen plane

FOR every pixel in the screen plane
    determine the ray from the eye through the pixel's centre
    FOR each object in the scene
        IF the object is intersected by the ray
            IF the intersection is the closest (so far) to the eye
                record intersection point and object
            END IF ;
        END IF ;
    END FOR ;
    set pixel's colour to that of the object at the closest intersection point
END FOR ;
```

176

Intersection of a ray with an object 1

- plane



ray: $P = O + sD,\ s \geq 0$

plane: $P \cdot N + d = 0$

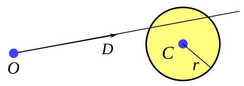
$$s = -\frac{d + N \cdot O}{N \cdot D}$$

- polygon
  - intersection the ray with the plane of the polygon
    - as above
  - then check to see whether the intersection point lies inside the polygon
    - a 2D geometry problem

177

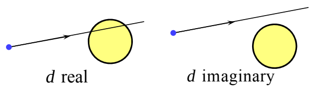
Intersection of a ray with an object 2

- sphere



ray: $P = O + sD,\ s \geq 0$

sphere: $(P - C) \cdot (P - C) - r^2 = 0$

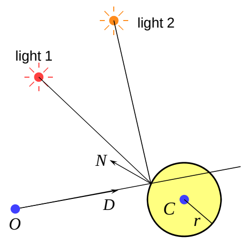


$$a = D \cdot D$$
$$b = 2D \cdot (O - C)$$
$$c = (O - C) \cdot (O - C) - r^2$$
$$d = \sqrt{b^2 - 4ac}$$
$$s_1 = \frac{-b + d}{2a}$$
$$s_2 = \frac{-b - d}{2a}$$

- cylinder, cone, torus
  - all similar to sphere
  - much more on this in the Part II Advanced Graphics course

178

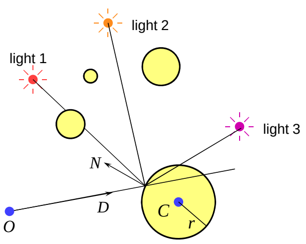
Ray tracing: shading



- once you have the intersection of a ray with the nearest object you can also:
  - calculate the normal to the object at that intersection point
  - shoot rays from that point to all of the light sources, and calculate the diffuse and specular reflections off the object at that point
    - this (plus ambient illumination) gives the colour of the object (at that point)

179

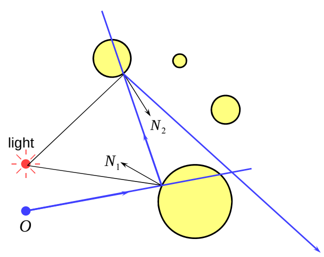
Ray tracing: shadows



- because you are tracing rays from the intersection point to the light, you can check whether another object is between the intersection and the light and is hence casting a shadow
  - also need to watch for self-shadowing

180

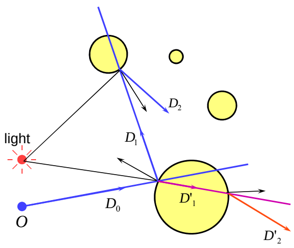
Ray tracing: reflection



- if a surface is totally or partially reflective then new rays can be spawned to find the contribution to the pixel's colour given by the reflection
  - this is perfect (mirror) reflection

181

Ray tracing: transparency & refraction

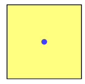


- objects can be totally or partially transparent
  - this allows objects behind the current one to be seen through it
- transparent objects can have refractive indices
  - bending the rays as they pass through the objects
- transparency + reflection means that a ray can split into two parts

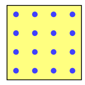
182

Sampling in ray tracing

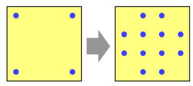
- single point
  - shoot a single ray through the pixel's centre



- super-sampling for anti-aliasing
  - shoot multiple rays through the pixel and average the result
  - regular grid, random, jittered, Poisson disc



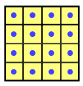
- adaptive super-sampling
  - shoot a few rays through the pixel, check the variance of the resulting values, if similar enough stop, otherwise shoot some more rays

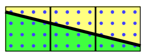


183

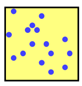
Types of super-sampling 1

- regular grid
  - divide the pixel into a number of sub-pixels and shoot a ray through the centre of each
  - problem: can still lead to noticable aliasing unless a very high resolution sub-pixel grid is used





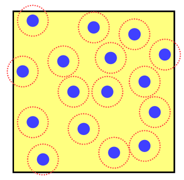
- random
  - shoot N rays at random points in the pixel
  - replaces aliasing artefacts with noise artefacts
    - the eye is far less sensitive to noise than to aliasing

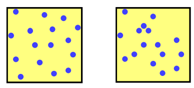


184

Types of super-sampling 2

- Poisson disc
  - shoot N rays at random points in the pixel with the proviso that no two rays shall pass through the pixel closer than ε to one another
  - for N rays this produces a better looking image than pure random sampling
  - very hard to implement properly

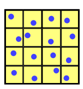


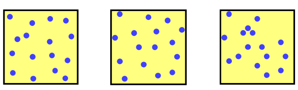


185

Types of super-sampling 3

- jittered
  - divide pixel into N sub-pixels and shoot one ray at a random point in each sub-pixel
  - an approximation to Poisson disc sampling
  - for N rays it is better than pure random sampling
  - easy to implement





186

More reasons for wanting to take multiple samples per pixel

- super-sampling is only one reason why we might want to take multiple samples per pixel
- many effects can be achieved by distributing the multiple samples over some range
  - called *distributed* ray tracing
    - N.B. *distributed* means distributed over a range of values
- can work in two ways
  1. each of the multiple rays shot through a pixel is allocated a random value from the relevant distribution(s)
     - all effects can be achieved this way with sufficient rays per pixel
  2. each ray spawns multiple rays when it hits an object
     - this alternative can be used, for example, for area lights

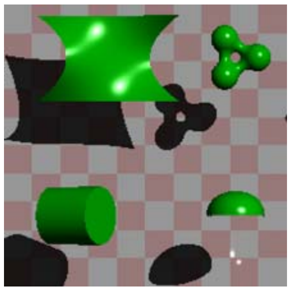
187

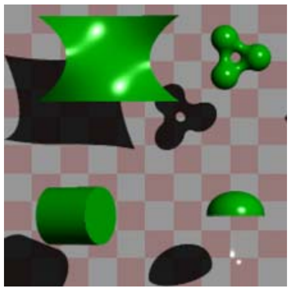
Examples of distributed ray tracing

- distribute the samples for a pixel over the pixel area
  - get random (or jittered) super-sampling
  - used for anti-aliasing
- distribute the rays going to a light source over some area
  - allows area light sources in addition to point and directional light sources
  - produces soft shadows with penumbrae
- distribute the camera position over some area
  - allows simulation of a camera with a finite aperture lens
  - produces depth of field effects
- distribute the samples in time
  - produces motion blur effects on any moving objects

188

Anti-aliasing


one sample per pixel


multiple samples per pixel

189

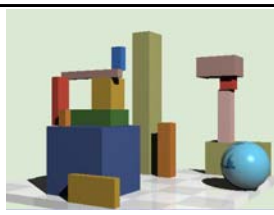
Area *vs* point light source

an area light source produces soft shadows

a point light source produces hard shadows

190

Finite aperture



left, a pinhole camera

below, a finite aperture camera

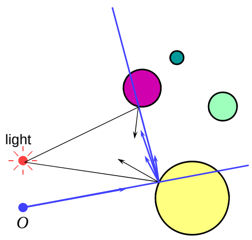
below left, 12 samples per pixel

below right, 120 samples per pixel

note the depth of field blur: only objects at the correct distance are in focus

191

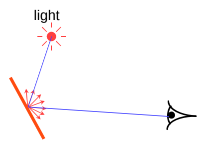
Distributed ray tracing for specular reflection



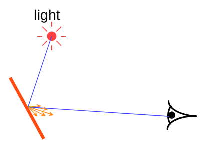
- previously we could only calculate the effect of perfect reflection
- we can now distribute the reflected rays over the range of directions from which specularly reflected light could come
- provides a method of handling some of the inter-reflections between objects in the scene
- requires a very large number of ray per pixel

192

Handling direct illumination



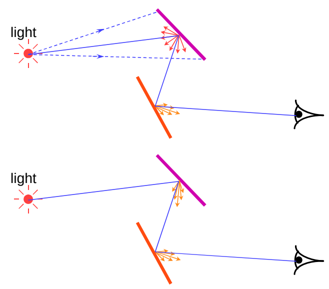
- diffuse reflection
  - handled by ray tracing and polygon scan conversion
  - assumes that the object is a perfect Lambertian reflector



- specular reflection
  - also handled by ray tracing and polygon scan conversion
  - use Phong's approximation to true specular reflection

193

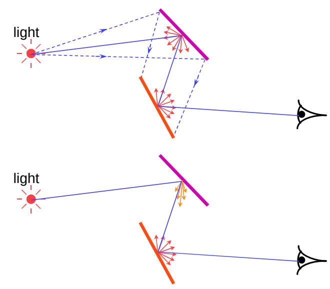
Handing indirect illumination: 1



- diffuse to specular
  - handled by distributed ray tracing
- specular to specular
  - also handled by distributed ray tracing

194

Handing indirect illumination: 2



- diffuse to diffuse
  - handled by radiosity
    - covered in the Part II Advanced Graphics course
- specular to diffuse
  - handled by no usable algorithm
  - some research work has been done on this but uses enormous amounts of CPU time

195

Multiple inter-reflection

- light may reflect off many surfaces on its way from the light to the camera — (diffuse | specular)*
- standard ray tracing and polygon scan conversion can handle a single diffuse or specular bounce — diffuse | specular
- distributed ray tracing can handle multiple specular bounces — (diffuse | specular) (specular)*
- radiosity can handle multiple diffuse bounces — (diffuse)*
- the general case cannot be handled by any efficient algorithm — (diffuse | specular)*

196

Hybrid algorithms

- *polygon scan conversion* and *ray tracing* are the two principal 3D rendering mechanisms
  - each has its advantages
    - *polygon scan conversion* is faster
    - *polygon scan conversion* can be implemented easily in hardware
    - *ray tracing* produces more realistic looking results
- hybrid algorithms exist
  - these generally use the speed of polygon scan conversion for most of the work and use ray tracing only to achieve particular special effects

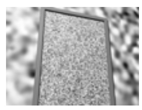
197

Commercial uses

- polygon scan conversion
  - in particular *z*-buffer and A-buffer
  - used almost everywhere
    - games
    - user interfaces
    - most special effects
- ray tracing
  - used when realism or beauty is absolutely crucial
    - advertising
    - some special effects

198

Surface detail


- so far we have assumed perfectly smooth, uniformly coloured surfaces
- real life isn't like that:
  - multicoloured surfaces
    - e.g. a painting, a food can, a page in a book
  - bumpy surfaces
    - e.g. almost any surface! (very few things are perfectly smooth)
  - textured surfaces
    - e.g. wood, marble

199

Texture mapping

without

all surfaces are smooth and of uniform colour

with

most surfaces are textured with 2D texture maps
the pillars are textured with a solid texture

200

Basic texture mapping

- a texture is simply an image, with a 2D coordinate system (u,v)

- each 3D object is parameterised in (u,v) space
- each pixel maps to some part of the surface
- that part of the surface maps to part of the texture

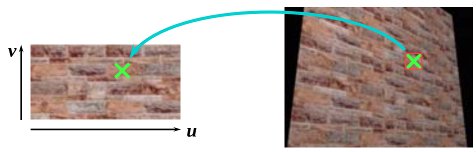
201

Paramaterising a primitive

- polygon: give (u,v) coordinates for three vertices, or treat as part of a plane
- plane: give u-axis and v-axis directions in the plane
- cylinder: one axis goes up the cylinder, the other around the cylinder

202

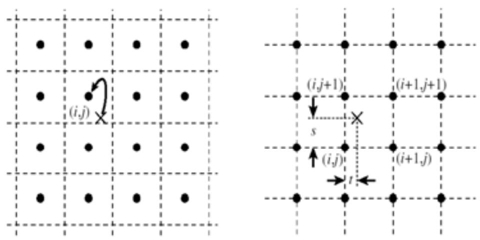
Sampling texture space



Find (u,v) coordinate of the sample point on the object and map this into texture space as shown

203

Sampling texture space: finding the value



- nearest neighbour: the sample value is the nearest pixel value to the sample point
- bilinear reconstruction: the sample value is the weighted mean of the four pixels around the sample point

204

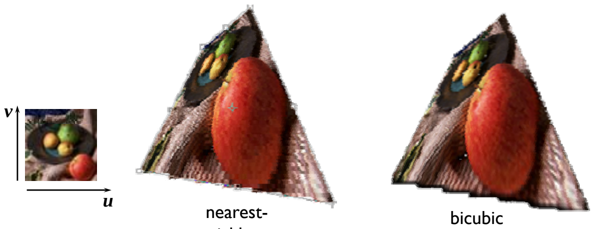
Sampling texture space: interpolation methods

the three standard methods:

- nearest neighbour
  - fast with many artefacts
- bilinear
  - reasonably fast, blurry
- bicubic
  - gives better results
  - uses 16 values (4×4) around the sample location
  - but runs at one quarter the speed of bilinear
- can we get any better?
  - many slower techniques offering *slightly* higher quality
  - biquadratic is an interesting trade-off
    - use 9 values (3×3) around the sample location
    - faster than bicubic, slower than linear, results seem to be nearly as good as bicubic

205

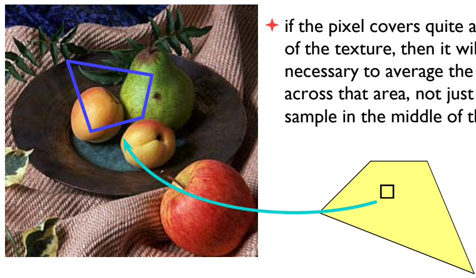
Texture mapping examples



look at the bottom right hand corner of the distorted image to compare the two interpolation methods

206

Down-sampling

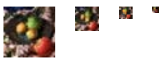


- if the pixel covers quite a large area of the texture, then it will be necessary to average the texture across that area, not just take a sample in the middle of the area

207

Multi-resolution texture

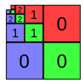
Rather than down-sampling every time you need to, have multiple versions of the texture at different resolutions and pick the appropriate resolution to sample from…



You can use tri-linear interpolation to get an even better result: that is, use bi-linear interpolation in the two nearest levels and then linearly interpolate between the two interpolated values

208

The MIP map



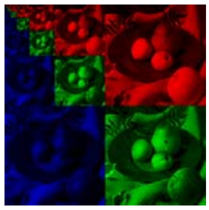
- an efficient memory arrangement for a multi-resolution colour image
- pixel (x,y) is a bottom level pixel location (level 0); for an image of size (m,n), it is stored at these locations in level k:

Red

$$\left(\left\lfloor\frac{m+x}{2^k}\right\rfloor, \left\lfloor\frac{y}{2^k}\right\rfloor\right)$$

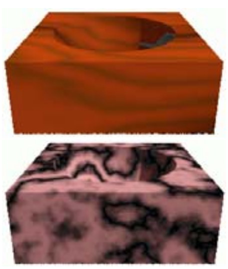
$$\left(\left\lfloor\frac{x}{2^k}\right\rfloor, \left\lfloor\frac{n+y}{2^k}\right\rfloor\right) \qquad \left(\left\lfloor\frac{m+x}{2^k}\right\rfloor, \left\lfloor\frac{n+y}{2^k}\right\rfloor\right)$$

Blue Green



209

Solid textures



- texture mapping applies a 2D texture to a surface
  $colour = f(u,v)$
- solid textures have colour defined for every point in space
  $colour = f(x,y,z)$
- permits the modelling of objects which appear to be carved out of a material

210

What can a texture map modify?

- any (or all) of the colour components
  - ambient, diffuse, specular
- transparency
  - "transparency mapping"
- reflectiveness
- but also the surface normal
  - "bump mapping"

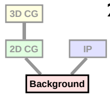
211

Bump mapping

- the surface normal is used in calculating both diffuse and specular reflection
- bump mapping modifies the direction of the surface normal so that the surface appears more or less bumpy
- rather than using a texture map, a 2D function can be used which varies the surface normal smoothly across the plane
- but bump mapping doesn't change the object's outline

212

Background



- what is a digital image?
  - what are the constraints on digital images?
- what hardware do we use?

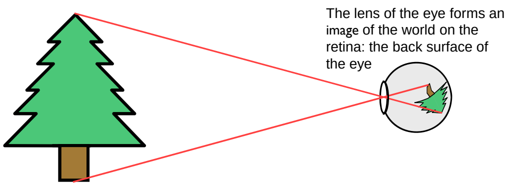
We are now ready to ask:

- how does human vision work?
  - what are the limits of human vision?
  - what can we get away with given these constraints & limits?
- how do we represent colour?
- how do displays & printers work?
  - how do we fool the human eye into seeing what we want it to see?

213

The workings of the human visual system

- to understand the requirements of displays (resolution, quantisation and colour) we need to know how the human eye works...



214

Structure of the human eye

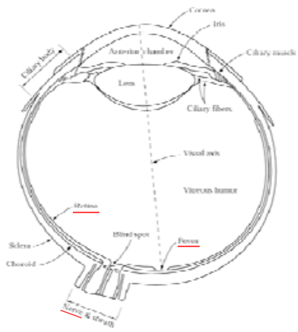


Fig. 2.1 from Gonzalez & Woods

- the retina is an array of light detection cells
- the fovea is the high resolution area of the retina
- the optic nerve takes signals from the retina to the visual cortex in the brain

215

The retina

- consists of about 150 million light receptors
- retina outputs information to the brain along the optic nerve
  - there are about one million nerve fibres in the optic nerve
  - the retina performs significant pre-processing to reduce the number of signals from 150M to 1M
  - pre-processing includes:
    - averaging multiple inputs together
    - colour signal processing
    - local edge detection

www.stlukeseye.com

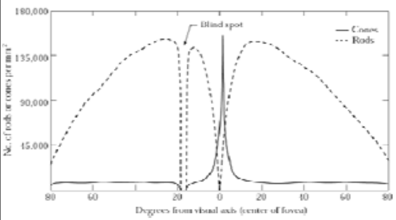
216

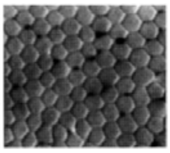
Light detectors in the retina

- two classes
  - rods
  - cones
- cones come in three types
  - sensitive to short, medium and long wavelengths
  - allow you to see in colour
- the cones are concentrated in the macula, at the centre of the retina
- the fovea is a densely packed region in the centre of the macula
  - contains the highest density of cones
  - provides the highest resolution vision

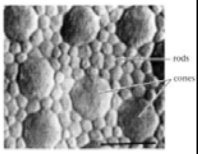
217

Distribution of rods & cones




cones in the fovea

(1) cones in the fovea are squished together more tightly than outside the fovea: higher resolution vision;
(2) as the density of cones drops the gaps between them are filled with rods



rods & cones outside the fovea

Fig. 2.2 from Gonzalez & Woods
www.cis.rit.edu/people/faculty/montag/vandplite/pages/chap_9/ch9p1.html

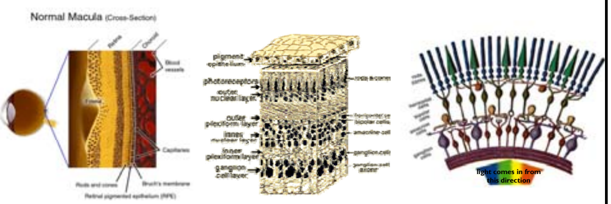
218

Foveal vision

- 150,000 cones per square millimetre in the fovea
  - high resolution
  - colour
- outside fovea: mostly rods
  - lower resolution
    - many rods' inputs are combined to produce one signal to the visual cortex in the brain
  - principally monochromatic
    - there are very few cones, so little input available to provide colour information to the brain
  - provides peripheral vision
    - allows you to keep the high resolution region in context
    - without peripheral vision you would walk into things, be unable to find things easily, and generally find life much more difficult

219

Detailed structure of retinal processing



- a lot of pre-processing occurs in the retina before signals are passed to the brain
- many light receptors have their signals combined into a single signal to the brain

www.ahaf.org/macular/about/NormalMacula.htm
www.phys.ufl.edu/~avery/course/3400/vision/retina_schema.jpg .../retina_schema2.jpg

220

Some of the processing in the eye

- discrimination
  - discriminates between different intensities and colours
- adaptation
  - adapts to changes in illumination level and colour
  - can see about 1:100 contrast at any given time
  - but can adapt to see light over a range of 10^{10}
- persistence
  - integrates light over a period of about 1/30 second
- edge detection and edge enhancement
  - visible in e.g. Mach banding effects

221

Intensity adaptation

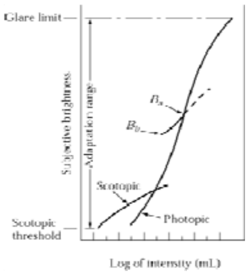


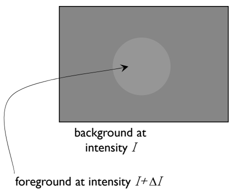
Fig. 2.4 from Gonzalez & Woods

- at any one time the eye can handle intensities over a range of ~100:1
  - this is the curve B_bB_a
  - anything darker is seen as black
    - if everything is black, the eye adjusts down
  - anything brighter causes pain
    - and stimulates the eye to adjust up
- the eye can adjust over a range of 10^7:1 in colour vision
  - the curve B_bB_a slides up or down the *photopic* curve
- at very low light levels only rods are effective
  - this is the *scotopic* curve
  - no colour, because the cones are not able to pick up any light

222

Intensity differentiation

- the eye can obviously differentiate between different colours and different intensities
- Weber's Law tells us how good the eye is at distinguishing different intensities using *just noticeable differences*



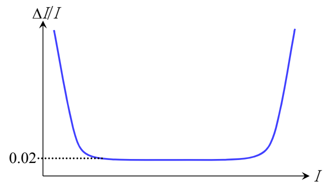
for a range of values of I

- start with $\Delta I=0$
  increase ΔI until human observer can just see a difference
- start with ΔI large
  decrease ΔI until human observer can just *not* see a difference

223

Intensity differentiation

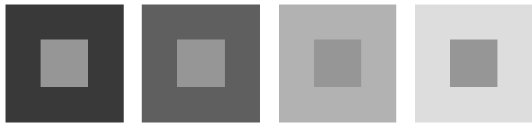
- results for a "normal" viewer
  - a human can distinguish about a 2% change in intensity for much of the range of intensities
  - discrimination becomes rapidly worse as you get close to the darkest or brightest intensities that you can currently see



224

Simultaneous contrast

- the eye performs a range of non-linear operations
- for example, as well as responding to changes in overall light, the eye responds to local changes

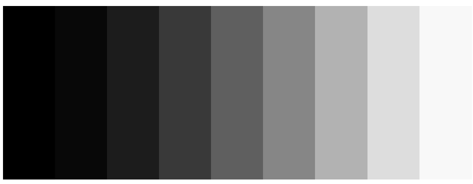


The centre square is the same intensity in all four cases but does not appear to be because your visual system is taking the local contrast into account

225

Mach bands

- show the effect of edge enhancement in the retina's pre-processing

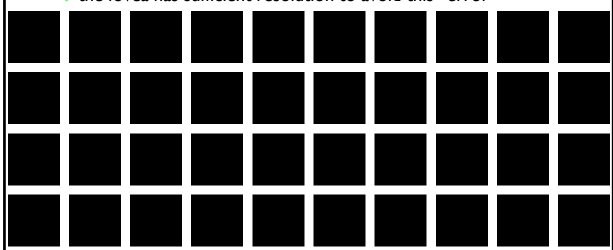


Each of the nine rectangles is a constant colour but you will see each rectangle being slightly brighter at the end which is near a darker rectangle and slightly darker at the end which is near a lighter rectangle

226

Ghost squares

- another effect caused by retinal pre-processing
  - the edge detectors *outside* the fovea cause you to see grey squares at the corners where four black squares join
  - the fovea has sufficient resolution to avoid this "error"



227

Summary of what human eyes do...

- sample the image that is projected onto the retina
- adapt to changing conditions
- perform non-linear pre-processing
  - makes it very hard to model and predict behaviour
- combine a large number of basic inputs into a much smaller set of signals
  - which encode more complex data
    - e.g. presence of an edge at a particular location with a particular orientation rather than intensity at a set of locations
- pass pre-processed information to the visual cortex
  - which performs extremely complex processing
  - discussed in the *Computer Vision* course

228

Implications of vision on resolution

- the acuity of the eye is measured as the ability to see a white gap, 1 minute wide, between two black lines
  - about 300dpi at 30cm
  - the corresponds to about 2 cone widths on the fovea
- resolution decreases as contrast decreases
- colour resolution is much worse than intensity resolution
  - this is exploited in TV broadcast
    - analogue television broadcasts the colour signal at half the horizontal resolution of the intensity signal

229

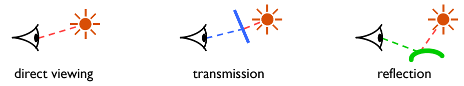
Implications of vision on quantisation

- humans can distinguish, at best, about a 2% change in intensity
  - not so good at distinguishing colour differences
- we need to know what the brightest white and darkest black are
  - most modern display technologies (CRT, LCD, plasma) have contrast ratios in the hundreds
    - ranging from 100:1 to about 600:1
  - movie film has a contrast ratio of about 1000:1
- ⇒ 12–16 bits of intensity information
  - assuming intensities are distributed linearly
    - this allows for easy computation
  - 8 bits are often acceptable, except in the dark regions

230

What is required for vision?

- illumination
  - some source of light
- objects
  - which reflect (or transmit) the light
- eyes
  - to capture the light as an image



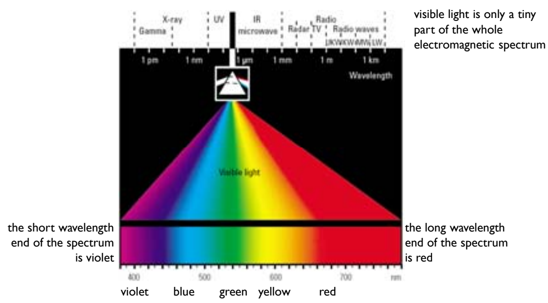
231

Light: wavelengths & spectra

- light is electromagnetic radiation
  - visible light is a tiny part of the electromagnetic spectrum
  - visible light ranges in wavelength from 700nm (red end of spectrum) to 400nm (violet end)
- every light has a spectrum of wavelengths that it emits
- every object has a spectrum of wavelengths that it reflects (or transmits)
- the combination of the two gives the spectrum of wavelengths that arrive at the eye

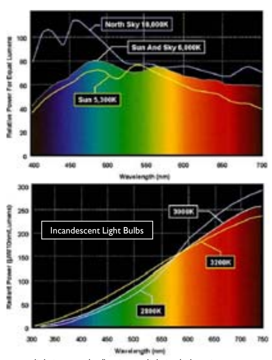
232

The spectrum



233

Illuminants have different characteristics

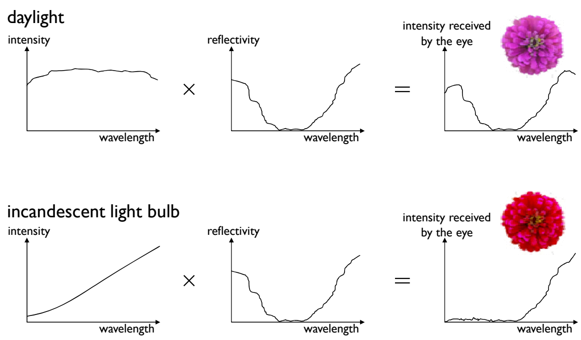


- different lights emit different intensities of each wavelength
  - sunlight is reasonably uniform
  - incandescent light bulbs are very red
  - sodium street lights emit almost pure yellow

www.gelighting.com/na/business_lighting/education_resources/learn_about_light/

234

Illuminant × reflection = reflected light



235

incandescent light bulb — camera flash bulb

winter sunlight — halogen light bulbs (overhead)

Comparison of illuminants

compare these things:

- colour of the monkey's nose and paws: more red under certain lights
- oranges & yellows (similar in all)
- blues & violets (considerably different)

236

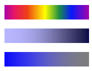
Representing colour

- we need a mechanism which allows us to represent colour in the computer by some set of numbers
  - preferably a small set of numbers which can be quantised to a fairly small number of bits each
- we will discuss:
  - Munsell's *artists'* scheme
    - which classifies colours on a perceptual basis
  - the mechanism of colour vision
    - how colour perception works
  - various *colour spaces*
    - which quantify colour based on either physical or perceptual models of colour

237

Munsell's colour classification system

- three axes
  - hue ➢ the dominant colour
  - value ➢ bright colours/dark colours
  - chroma ➢ vivid colours/dull colours

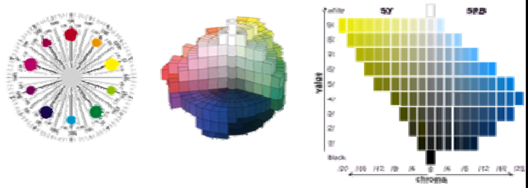


- can represent this as a 3D graph

238

Munsell's colour classification system

- any two adjacent colours are a standard "perceptual" distance apart
  - worked out by testing it on people
  - a highly irregular space
    - e.g. vivid yellow is much brighter than vivid blue

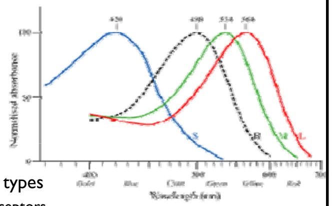


invented by Albert H. Munsell, an American artist, in 1905 in an attempt to systematically classify colours

239

Colour vision

- there are three types of cone
- each responds to a different spectrum
  - *very* roughly long, medium, and short wavelengths
  - each has a response function: l(λ), m(λ), s(λ)



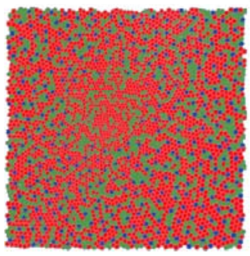
- different numbers of the different types
  - far fewer of the short wavelength receptors
  - so cannot see fine detail in blue
- overall intensity response of the cones can be calculated
  - $y(\lambda) = l(\lambda) + m(\lambda) + s(\lambda)$
  - $y = k \int P(\lambda)\, y(\lambda)\, d\lambda$ is the perceived *luminance* in the fovea
  - $y = k \int P(\lambda)\, r(\lambda)\, d\lambda$ is the perceived *luminance* outside the fovea

r(λ) is the response function of the rods

240

Distribution of different cone types

simulated cone distribution at the centre of the fovea



- this is about 1° of visual angle
- distribution is:
  - 7% short, 37% medium, 56% long
- short wavelength receptors
  - regularly distributed
  - not in the central 1/3°
  - outside the fovea, only 1% of cones are short
- long & medium
  - about 3:2 ratio long:medium

www.cis.rit.edu/people/faculty/montag/vandplite/pages/chap_9/ch9p1.html

241

Colour signals sent to the brain

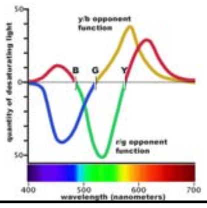
- the signal that is sent to the brain is pre-processed by the retina

long + medium + short = luminance

long – medium = red-green

long + medium – short = yellow-blue

- this theory explains:
  - colour-blindness effects
  - why red, yellow, green and blue are perceptually important colours
  - why you can see e.g. a yellowish red but not a greenish red



242

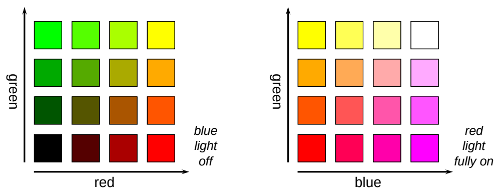
Chromatic metamerism

- many different spectra will induce the same response in our cones
  - the values of the three perceived values can be calculated as:
    - $l = k \int P(\lambda)\, l(\lambda)\, d\lambda$
    - $m = k \int P(\lambda)\, m(\lambda)\, d\lambda$
    - $s = k \int P(\lambda)\, s(\lambda)\, d\lambda$
  - k is some constant, $P(\lambda)$ is the spectrum of the light incident on the retina
  - two different spectra (e.g. $P_1(\lambda)$ and $P_2(\lambda)$) can give the same values of l, m, s
  - we can thus fool the eye into seeing (almost) any colour by mixing correct proportions of some small number of lights

243

Mixing coloured lights

- by mixing different amounts of red, green, and blue lights we can generate a wide range of responses in the human eye



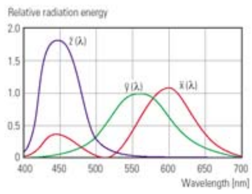
- not all colours can be created in this way

244

FvDFH Sec 13.2.2

XYZ colour space

- not every wavelength can be represented as a mix of red, green, and blue lights
- but matching & defining coloured light with a mixture of three fixed primaries is desirable
- CIE define three standard primaries: X, Y, Z



Y matches the human eye's response to light of a constant intensity at each wavelength (***luminous-efficiency function of the eye***)

X, Y, and Z are not themselves colours, they are used for defining colours – you cannot make a light that emits one of these primaries

XYZ colour space was defined in 1931 by the *Commission Internationale de l'Éclairage* (CIE)

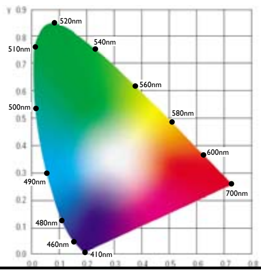
245

CIE chromaticity diagram

- *chromaticity* values are defined in terms of x, y, z

$$x = \frac{X}{X+Y+Z},\quad y = \frac{Y}{X+Y+Z},\quad z = \frac{Z}{X+Y+Z} \quad \therefore \quad x + y + z = 1$$

  - ignores luminance
  - can be plotted as a 2D function
- pure colours (single wavelength) lie along the outer curve
- all other colours are a mix of pure colours and hence lie inside the curve
- points outside the curve do not exist as colours



246

Colour spaces

- CIE *XYZ*, *Yxy*
- Uniform
  - equal steps in any direction make equal perceptual differences
  - CIE $L^*u^*v^*$, CIE $L^*a^*b^*$
- Pragmatic
  - used because they relate directly to the way that the hardware works
  - *RGB*, *CMY*, *CMYK*
- Munsell-like
  - used in user-interfaces
  - considered to be easier to use for specifying colour than are the pragmatic colour spaces
  - map easily to the pragmatic colour spaces
  - *HSV*, *HLS*

247

XYZ is not perceptually uniform

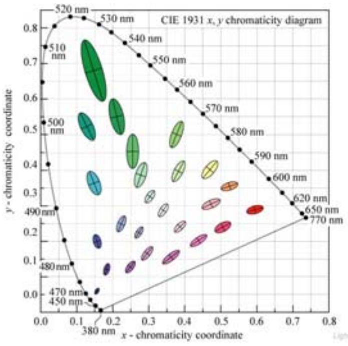


Fig. 17.5. MacAdam ellipses plotted in the CIE 1931 (x, y) chromaticity diagram. The axes of the ellipses are ten times their actual lengths (after MacAdam, 1943; Wright, 1943; MacAdam, 1993).

Each ellipse shows how far you can stray from the central point before a human being notices a difference in colour

248

Luv was designed to be more uniform

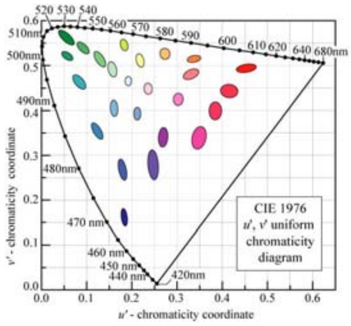
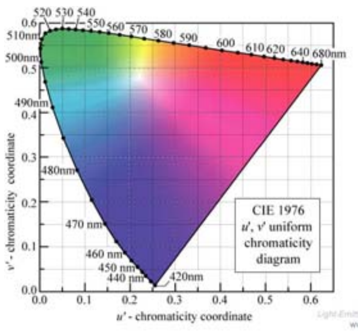


Fig. 17.7. MacAdam ellipses transformed to uniform CIE 1976 (u', v') chromaticity coordinates. For clarity, the axes of the transformed ellipses are ten times their actual lengths. Transformed ellipses are not ellipses in a strict mathematical sense, but their shapes closely resemble those of ellipses. The areas of the transformed ellipses in the (u', v') diagram are much more similar than the MacAdam ellipses in the (x, y) diagram.

249

Luv colour space



L is luminance and is orthogonal to *u* and *v*, the two colour axes

Fig. 17.6. CIE 1976 (u', v') uniform chromaticity diagram calculated using the CIE 1931 2° standard observer.

*L*u*v** is an official CIE colour space. It is a straightforward distortion of *XYZ* space.

250

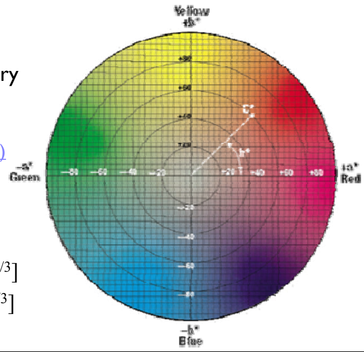
Lab space

- another CIE colour space
- based on complementary colour theory
  - see slide 49 (Colour signals sent to the brain)
- also aims to be perceptually uniform

$L^*=116(Y/Y_n)^{1/3}$

$a^*=500[(X/X_n)^{1/3}-(Y/Y_n)^{1/3}]$

$b^*=200[(Y/Y_n)^{1/3}-(Z/Z_n)^{1/3}]$



251

Lab space

- this visualization shows those colours in *Lab* space which a human can perceive
- again we see that human perception of colour is not uniform
  - perception of colour diminishes at the white and black ends of the *L* axis
  - the maximum perceivable chroma differs for different hues

252

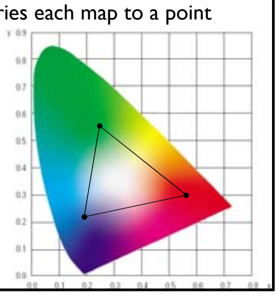
RGB space

- all display devices which output light mix red, green and blue lights to make colour
  - televisions, CRT monitors, video projectors, LCD screens
- nominally, *RGB* space is a cube
- the device puts physical limitations on:
  - the range of colours which can be displayed
  - the brightest colour which can be displayed
  - the darkest colour which can be displayed

253

RGB in *XYZ* space

- CRTs and LCDs mix red, green, and blue to make all other colours
- the red, green, and blue primaries each map to a point in *XYZ* space
- any colour within the resulting triangle can be displayed
  - any colour outside the triangle cannot be displayed
  - for example: CRTs cannot display very saturated purple, turquoise, or yellow



254

CMY space

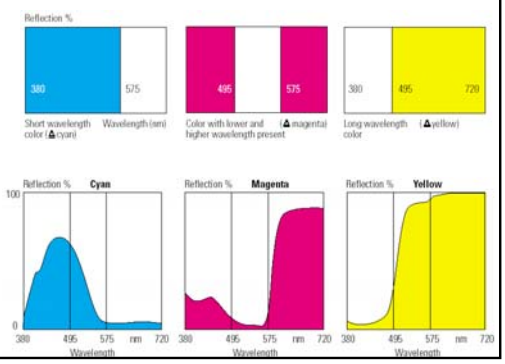
- printers make colour by mixing coloured inks
- the important difference between inks (*CMY*) and lights (*RGB*) is that, while lights *emit* light, inks *absorb* light
  - cyan absorbs red, reflects blue and green
  - magenta absorbs green, reflects red and blue
  - yellow absorbs blue, reflects green and red
- *CMY* is, at its simplest, the inverse of *RGB*
- *CMY* space is nominally a cube

255

Ideal and actual printing ink reflectivities

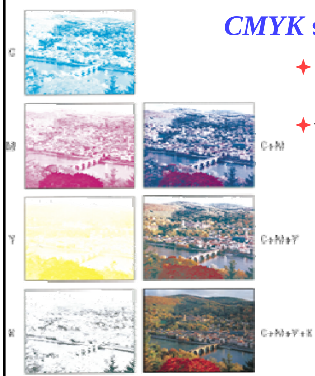
ideal

actual



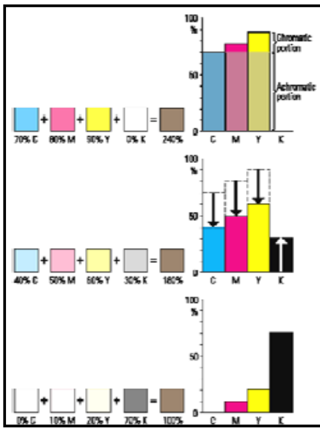
256

CMYK space



- in real printing we use black (key) as well as *CMY*
- why use black?
  - inks are not perfect absorbers
  - mixing *C* + *M* + *Y* gives a muddy grey, not black
  - lots of text is printed in black: trying to align *C*, *M* and *Y* perfectly for black text would be a nightmare

257



Using K

- if we print using just CMY then we can get up to 300% ink at any point on the paper
- removing the achromatic portion of CMY and replacing with K reduces the maximum possible ink coverage to 200%

258

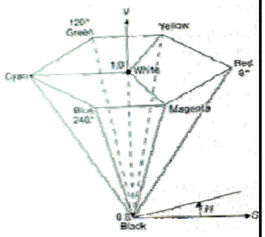
Colour spaces for user-interfaces

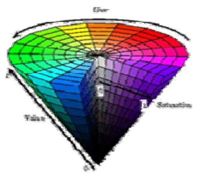
- *RGB* and *CMY* are based on the physical devices which produce the coloured output
- *RGB* and *CMY* are difficult for humans to use for selecting colours
- Munsell's colour system is much more intuitive:
  - hue — what is the principal colour?
  - value — how light or dark is it?
  - chroma — how vivid or dull is it?
- computer interface designers have developed basic transformations of *RGB* which resemble Munsell's human-friendly system

259

HSV: hue saturation value

- three axes, as with Munsell
  - hue and value have same meaning
  - the term "saturation" replaces the term "chroma"



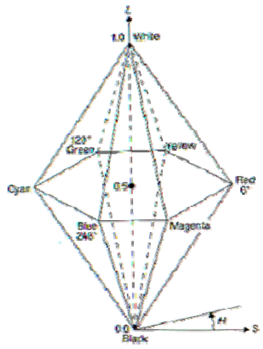


- designed by Alvy Ray Smith in 1978
- algorithm to convert *HSV* to *RGB* and back can be found in Foley et al., Figs 13.33 and 13.34

260

HLS: hue lightness saturation

- a simple variation of *HSV*
  - hue and saturation have same meaning
  - the term "lightness" replaces the term "value"
- designed to address the complaint that *HSV* has all pure colours having the same lightness/value as white
  - designed by Metrick in 1979
  - algorithm to convert *HLS* to *RGB* and back can be found in Foley et al., Figs 13.36 and 13.37



261

Summary of colour spaces

- the eye has three types of colour receptor
- therefore we can validly use a three-dimensional co-ordinate system to represent colour
- *XYZ* is one such co-ordinate system
  - *Y* is the eye's response to intensity (luminance)
  - *X* and *Z* are, therefore, the colour co-ordinates
    - same *Y*, change *X* or *Z* ⇒ same intensity, different colour
    - same *X* and *Z*, change *Y* ⇒ same colour, different intensity
- there are other co-ordinate systems with a luminance axis
  - *L*a*b**, *L*u*v**, *HSV*, *HLS*
- some other systems use three colour co-ordinates
  - *RGB*, *CMY*
  - luminance can then be derived as some function of the three
    - e.g. in *RGB*: $Y = 0.299\,R + 0.587\,G + 0.114\,B$

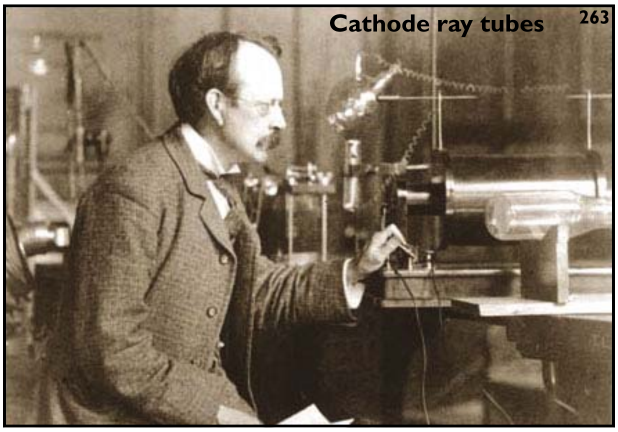
262

Image display

- a handful of technologies cover over 99% of all display devices
  - active displays
    - cathode ray tube — declining use
    - liquid crystal display — rapidly increasing use
    - plasma displays — increasing use
    - digital mirror displays — increasing use in video projectors
  - printers (passive displays)
    - laser printers — the traditional office printer
    - ink jet printers — low cost, rapidly increasing in quality, the traditional home printer
    - commercial printers — for high volume

263

Cathode ray tubes



264

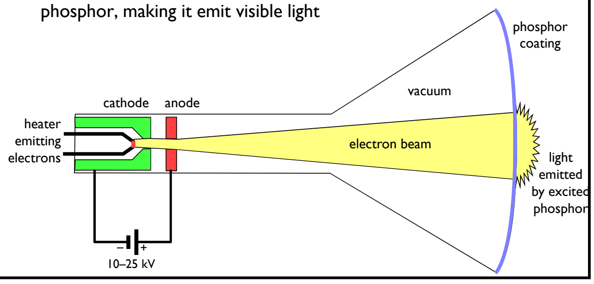
Cathode ray tubes

- focus an electron gun on a phosphor screen
  - produces a bright spot
- scan the spot back and forth, up and down to cover the whole screen
- vary the intensity of the electron beam to change the intensity of the spot
- repeat this fast enough and humans see a continuous picture
  - displaying pictures sequentially at > 20Hz gives illusion of continuous motion
  - but humans are sensitive to flicker at frequencies higher than this...

265

Basic cathode ray tube I

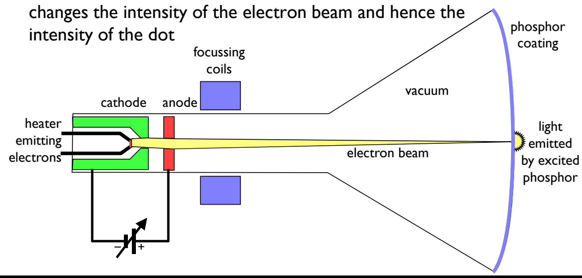
- the heater emits electrons, which are accelerated by the potential difference between cathode and anode
- electrons hitting the front screen excite the phosphor, making it emit visible light



266

Basic cathode ray tube II

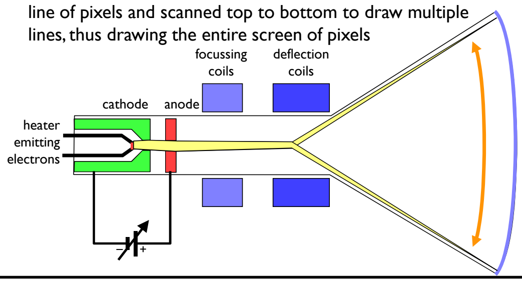
- focussing coils magnetically focus the electron beam to a small spot on the CRT faceplate, making a single dot
- changing the voltage between the cathode and anode changes the intensity of the electron beam and hence the intensity of the dot



267

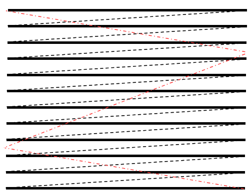
Basic cathode ray tube III

- deflection coils magnetically deflect the electron beam to any position on the screen
- the beam is scanned quickly left to right to draw a single line of pixels and scanned top to bottom to draw multiple lines, thus drawing the entire screen of pixels

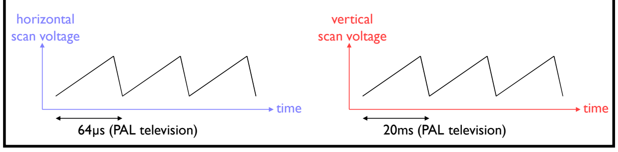


268

Raster scan



- the electron beam cannot be instantly pulled from one side of the screen to the other or from bottom to top, so there is *flyback* time, where the intensity of the beam is made zero and the scanning coils whizz back to the start as fast as is electromagnetically possible
- it is not possible to scan in alternate directions on alternate lines because this would make a zigzag pattern of pixel locations rather than a sequence of lines.



269

How fast do CRTs need to be?

speed at which entire screen is updated is called the "refresh rate"

- 50Hz (PAL TV, used in most of Europe)
  - many people can see a slight flicker
- 60Hz (NTSC TV, used in USA and Japan)
  - better
- 80-90Hz
  - 99% of viewers see no flicker, even on very bright displays
- 100Hz (newer "flicker-free" PAL TV sets)
  - practically no-one can see the image flickering

Flicker/resolution trade-off

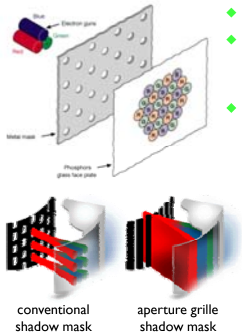
PAL 50Hz 768x576

NTSC 60Hz 640x480

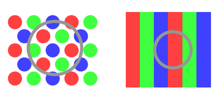
this trade-off is based on an historic maximum line rate from the early days of colour television, modern monitors can go much faster (higher resolution and faster refresh rate)

270

Colour CRTs: shadow masks



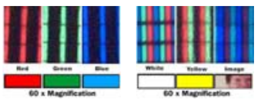
- use three electron guns & colour phosphors
- electrons have no colour
  - use shadow mask to direct electrons from each gun onto the appropriate phosphor
- the electron beams' spots are bigger than the shadow mask pitch
  - can get spot size down to 7/4 of the pitch
  - pitch can get down to 0.25mm with delta arrangement of phosphor dots
  - with a aperture grille shadow mask can reduce this to 0.15mm

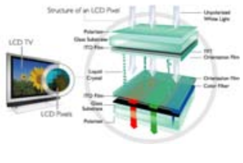


271

Liquid crystal displays I

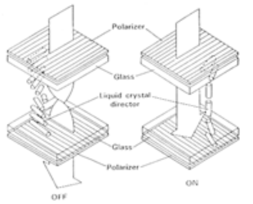
- liquid crystals can twist the polarisation of light
- basic control is by the voltage that is applied across the liquid crystal: either on or off, transparent or opaque
- greyscale can be achieved with some types of liquid crystal by varying the voltage
- colour is achieved with colour filters





272

Liquid crystal displays II



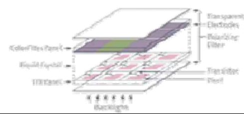
there are two polarizers at right angles to one another on either side of the liquid crystal: under normal circumstances these would block all light

there are liquid crystal directors: micro-grooves which align the liquid crystal molecules next to them

the liquid crystal molecules try to line up with one another; the micro-grooves on each side are at right angles to one another which forces the crystals' orientations to twist gently through 90° as you go from top to bottom, causing the polarization of the light to twist through 90°, making the pixel transparent

liquid crystal molecules are polar: they have a positive and a negative end

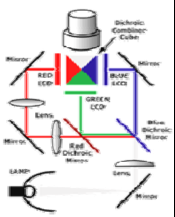
applying a voltage across the liquid crystal causes the molecules to stand on their ends, ruining the twisting phenomenon, so light cannot get through and the pixel is opaque



273

Liquid crystal displays III

- low power consumption compared to CRTs although the back light uses a lot of power
- image quality historically not as good as cathode ray tubes, but improved dramatically over the last ten years
- uses
  - laptops
  - video projectors
  - rapidly replacing CRTs as desk top displays
  - increasing use as televisions

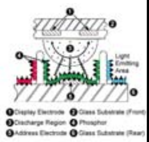


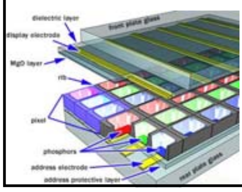
a three LCD video projector, with colour made by devoting one LCD panel to each of red, green and blue, and by splitting the light using dichroic mirrors which pass some wavelengths and reflect others

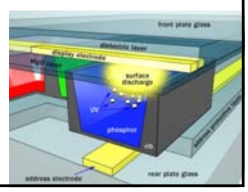
274

Plasma displays I

- a high voltage across the electrodes ionizes a noble gas (xenon, neon) which emits ultraviolet light, this excites the phosphor, which emits visible light
  - a plasma display therefore is essentially just thousands of tiny fluorescent lights







275

Plasma displays II

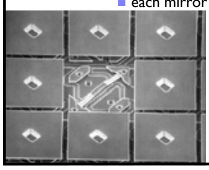
- plasma displays have been commercially available since 1993
  - but have been widely marketed since about 2000
- advantages
  - can be made larger than LCD panels
    - although LCD panels are getting bigger
  - much thinner than CRTs for equivalent screen sizes
- disadvantages
  - resolution (pixels per inch) not as good as either LCD or CRT
  - uses lots of power & gets hot
  - expensive compared to LCD or CRT technology
- uses
  - mostly used in television and advertising

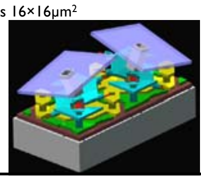
January 2004: Samsung release an LCD TV as big as a plasma TV at about twice the cost. Will plasma survive the challenge?

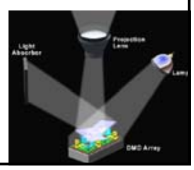
276

Digital micromirror devices I

- developed by Texas Instruments
  - often referred to as Digital Light Processing (DLP) technology
- invented in 1987, following ten year's work on deformable mirror devices
- manufactured like a silicon chip!
  - a standard 5 volt, 0.8 micron, CMOS process
  - micromirrors are coated with a highly reflected aluminium alloy
  - each mirror is 16×16μm²



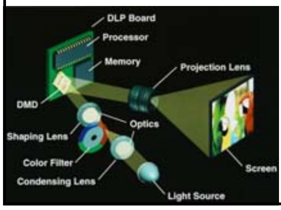


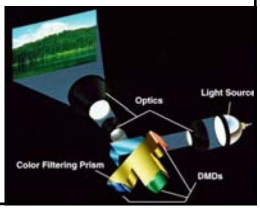


277

Digital micromirror devices II

- used increasingly in video projectors
- widely available from late 1990s
- colour is achieved using either three DMD chips or one chip and a rotating colour filter





278

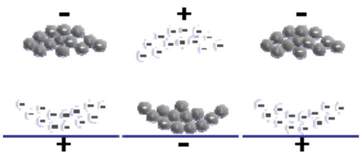
Electrophoretic displays I

- electronic paper widely used in e-books
- iRex iLiad, Sony Reader, Amazon Kindle
- 200 dpi passive display

279

Electrophoretic displays II

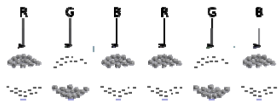
- transparent capsules ~40μ diameter
  - filled with dark oil
  - negatively charged 1μ titanium dioxide particles
- electrodes in substrate attract or repel white particles
- image persists with no power consumption



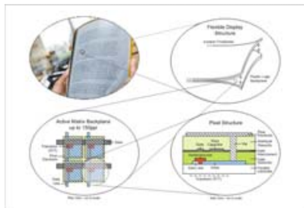
280

Electrophoretic displays III

- colour filters over individual pixels



- flexible substrate using plastic semiconductors (Plastic Logic)



281

Printers

- many types of printer
  - ink jet
    - sprays ink onto paper
  - laser printer
    - uses a laser to lay down a pattern of charge on a drum; this picks up charged toner which is then pressed onto the paper
  - commercial offset printer
    - an image of the whole page is put on a roller
    - this is repeatedly inked and pressed against the paper to print thousands of copies of the same thing
- all make marks on paper
  - essentially binary devices: mark/no mark

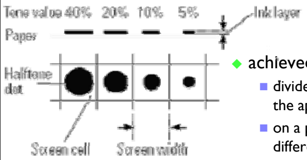
282

Printer resolution

- laser printer
  - 300–1200dpi
- ink jet
  - used to be lower resolution & quality than laser printers but now have comparable resolution
- phototypesetter for commercial offset printing
  - 1200–2400 dpi
- bi-level devices: each pixel is either on or off
  - black or white (for monochrome printers)
  - *ink* or *no ink* (in general)

283

What about greyscale?

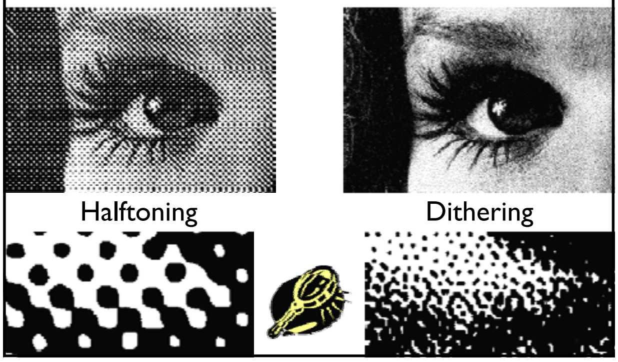


- achieved by halftoning
  - divide image into cells, in each cell draw a spot of the appropriate size for the intensity of that cell
  - on a printer each cell is $m \times m$ pixels, allowing m^2+1 different intensity levels
  - e.g. 300dpi with 4×4 cells ⇒ 75 cells per inch, 17 intensity levels
  - phototypesetters can make 256 intensity levels in cells so small you can only just see them
- an alternative method is dithering
  - dithering photocopies badly, halftoning photocopies well

will discuss halftoning and dithering in Image Processing *section of course*

284

Halftoning & dithering examples



285

What about colour?

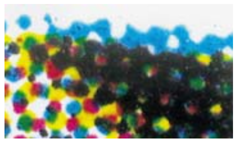
- generally use cyan, magenta, yellow, and black inks (CMYK)
- inks *aborb* colour
  - c.f. lights which *emit* colour
  - CMY is the inverse of RGB
- why is black (K) necessary?
  - inks are not perfect aborbers
  - mixing C + M + Y gives a muddy grey, not black
  - lots of text is printed in black: trying to align C, M and Y perfectly for black text would be a nightmare

see slide 64 CMYK space

286

How do you produce halftoned colour?

- print four halftone screens, one in each colour
- carefully angle the screens to prevent interference (moiré) patterns



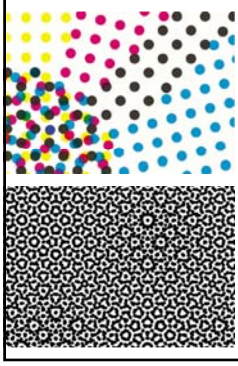
150 lpi × 16 dots per cell = 2400 dpi phototypesetter (16×16 dots per cell = 256 intensity levels)

Standard rulings (in lines per inch)

65 lpi	
85 lpi	newsprint
100 lpi	
120 lpi	uncoated offset paper
133 lpi	uncoated offset paper
150 lpi	matt coated offset paper or art paper publication: books, advertising leaflets
200 lpi	very smooth, expensive paper very high quality publication

287

Four colour halftone screens



- Standard angles
  - Cyan 15°
  - Black 45°
  - Magenta 75°
  - Yellow 90°
- At bottom is the moiré pattern
  - this is the best possible (minimal) moiré pattern
  - produced by this optimal set of angles
  - all four colours printed in black to highlight the effect

Magenta, Cyan & Black are at 30° relative to one another

Yellow (least distinctive colour) is at 15° relative to Magenta and Cyan

288

Range of printable colours

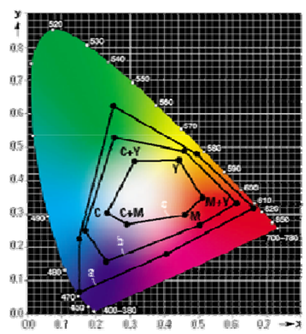
a: colour photography (diapositive)

b: high-quality offset printing

c: newspaper printing

why the hexagonal shape?

because we can print dots which only partially overlap making the situation more complex than for coloured lights



289

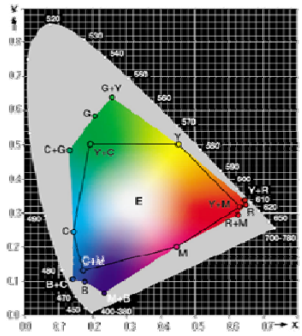
Beyond four colour printing

- printers can be built to do printing in more colours
  - gives a better range of printable colours
- six colour printing
  - for home photograph printing
  - dark & light cyan, dark & light magenta, yellow, black
- eight colour printing
  - 3× cyan, 3× magenta, yellow, black
  - 2× cyan, 2× magenta, yellow, 3× black
- twelve colour printing
  - 3× cyan, 3× magenta, yellow, black red, green, blue, orange

290

The extra range of colour

- this gamut is for so-called HiFi colour printing
  - uses cyan, magenta, yellow, plus red, green and blue inks



291

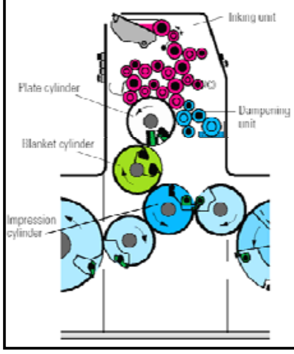
Commercial offset printing I

- for printing thousands of copies of the same thing
- produce printing plates at high resolution (e.g. 2400dpi) and bolt them into the printer
- often set up for five colour printing
  - CMYK plus a "spot colour"
    - CMYK allows you to reproduce photographs
    - the spot colour allows you to reproduce one particular colour (e.g. the corporate logo) accurately without halftoning

a typical sheet-fed offset press (Heidelberg Speedmaster SM 74–5–P–H)

292

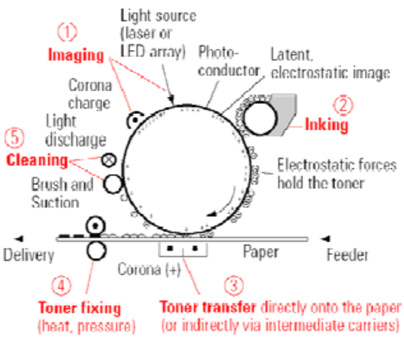
Commercial offset printing II



- the plate cylinder is where the printing plate is held
- this is dampened and inked anew on every pass
- the impression from the plate cylinder is passed onto the blanket cylinder
- it is then transferred it onto the paper which passes between the blanket and impression cylinders
- the blanket cylinder is there so that the printing plate does not come into direct contact with the paper

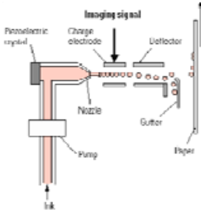
293

Laser printer

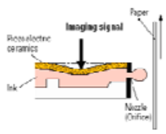


294

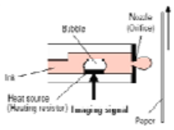
Ink jet printers



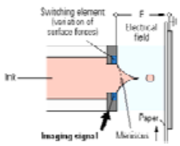
continuous ink jet (left)



piezo ink jet (right)



thermal ink jet *or* bubble jet (left)

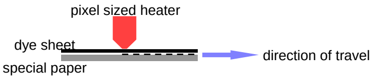


electrostatic ink jet (right)

295

Dye sublimation printers: true greyscale

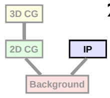
- dye sublimation gives true greyscale



- dye sublimes off dye sheet and onto paper in proportion to the heat level
- colour is achieved by using four different coloured dye sheets in sequence — the heat mixes them
- extremely expensive
- modern inkjet technology gives results of similar quality

296

Image Processing

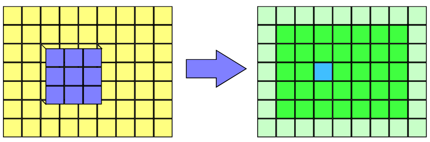


- filtering
  - convolution
  - nonlinear filtering
- point processing
  - intensity/colour correction
- compositing
- halftoning & dithering
- compression
  - various coding schemes

297

Filtering

- move a filter over the image, calculating a new value for every pixel



298

Filters - discrete convolution

- convolve a discrete filter with the image to produce a new image
  - in one dimension:

$$f'(x) = \sum_{i=-\infty}^{+\infty} h(i) \times f(x-i)$$

where $h(i)$ is the filter

  - in two dimensions:

$$f'(x,y) = \sum_{i=-\infty}^{+\infty} \sum_{j=-\infty}^{+\infty} h(i,j) \times f(x-i, y-j)$$

299

Example filters - averaging/blurring

Basic 3x3 blurring filter

$$\begin{bmatrix} \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \end{bmatrix} = \frac{1}{9} \times \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Gaussian 3x3 blurring filter

$$\frac{1}{16} \times \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

Gaussian 5x5 blurring filter

$$\frac{1}{112} \times \begin{bmatrix} 1 & 2 & 4 & 2 & 1 \\ 2 & 6 & 9 & 6 & 2 \\ 4 & 9 & 16 & 9 & 4 \\ 2 & 6 & 9 & 6 & 2 \\ 1 & 2 & 4 & 2 & 1 \end{bmatrix}$$

300

Example filters - edge detection

Horizontal **Vertical** **Diagonal**

Prewitt filters

$$\begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix} \quad \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} \quad \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & -1 \\ 0 & -1 & -1 \end{bmatrix}$$

Sobel filters

$$\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \quad \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \quad \begin{bmatrix} 2 & 1 & 0 \\ 1 & 0 & -1 \\ 0 & -1 & -2 \end{bmatrix}$$

Roberts filters

$$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$$

301

Example filter - horizontal edge detection

Horizontal edge detection filter

1	1	1
0	0	0
-1	-1	-1

$*$

Image

100	100	100	100	100	100	100	100	100
100	100	100	100	100	100	100	100	100
100	100	100	100	100	100	100	100	100
0	0	0	0	0	100	100	100	100
0	0	0	0	0	0	100	100	100
0	0	0	0	0	0	100	100	100
0	0	0	0	0	0	100	100	100

=

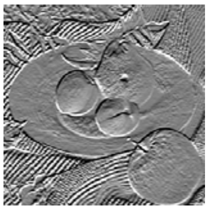
Result

0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
300	300	300	300	200	100	0	0	0
300	300	300	300	300	200	100	0	0
0	0	0	0	100	100	100	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0

302

Example filter - horizontal edge detection

original image



after use of a 3×3 Prewitt horizontal edge detection filter

mid-grey = no edge, black or white = strong edge

303

Median filtering

- not a convolution method
- the new value of a pixel is the median of the values of all the pixels in its neighbourhood

e.g. 3×3 median filter

10	15	17	21	24	27
12	16	20	25	99	37
15	22	23	25	38	42
18	37	36	39	40	44
34	2	40	41	43	47

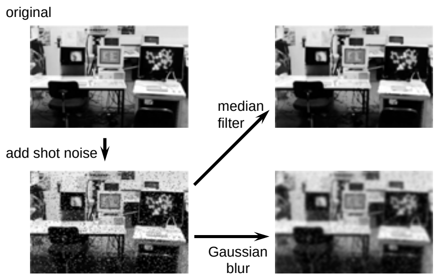
(16,20,22,23,
25,
25,36,37,39)

	16	21	24	27	
	20	25	36	39	
	23	36	39	41	

sort into order and take median

304

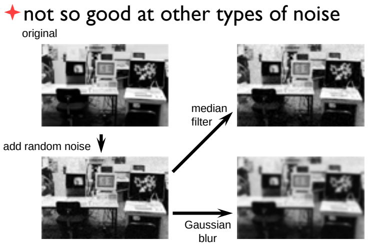
Median filter - example



305

Median filter - limitations

- copes well with shot (impulse) noise
- not so good at other types of noise



in this example, median filter reduces noise but doesn't eliminate it

Gaussian filter eliminates noise at the expense of excessive blurring

306

Point processing

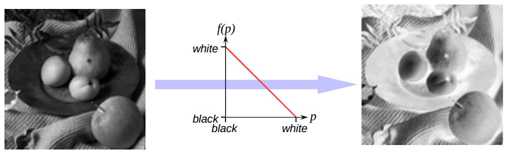
- each pixel's value is modified
- the modification function only takes that pixel's value into account

$$p'(i,j) = f\{p(i,j)\}$$

- where $p(i,j)$ is the value of the pixel and $p'(i,j)$ is the modified value
- the modification function, $f(p)$, can perform any operation that maps one intensity value to another

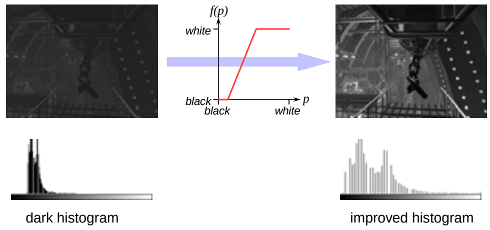
307

Point processing inverting an image



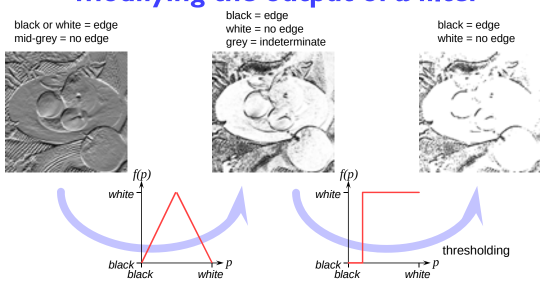
308

Point processing improving an image's contrast



309

Point processing modifying the output of a filter



310

Point processing: gamma correction

- the intensity displayed on a CRT is related to the voltage on the electron gun by:

$$i \propto V^{\gamma}$$

- the voltage is directly related to the pixel value:

$$V \propto p$$

- gamma correction modifies pixel values in the inverse manner:

$$p' = p^{1/\gamma}$$

- thus generating the appropriate intensity on the CRT:

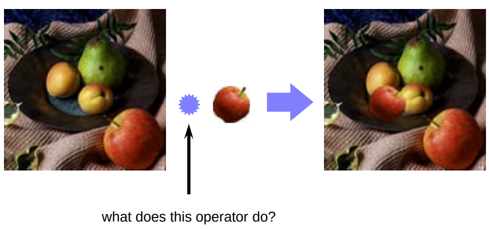
$$i \propto V^{\gamma} \propto p'^{\gamma} \propto p$$

- CRTs generally have gamma values around 2.0

311

Image compositing

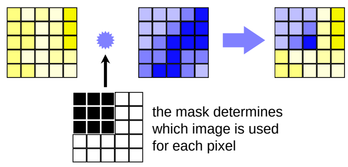
- merging two or more images together



312

Simple compositing

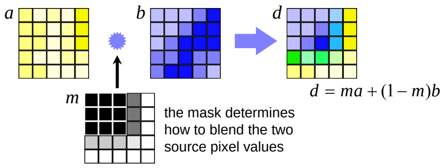
- copy pixels from one image to another
  - only copying the pixels you want
  - use a mask to specify the desired pixels



313

Alpha blending for compositing

- instead of a simple boolean mask, use an alpha mask
  - value of alpha mask determines how much of each image to blend together to produce final pixel



314

Arithmetic operations

- images can be manipulated arithmetically
  - simply apply the operation to each pixel location in turn
- multiplication
  - used in masking
- subtraction (difference)
  - used to compare images
  - e.g. comparing two x-ray images before and after injection of a dye into the bloodstream

315

Difference example

the two images are taken from slightly different viewpoints

take the difference between the two images

$$d = 1 - |a - b|$$

where 1 = white and 0 = black

black = large difference
white = no difference

316

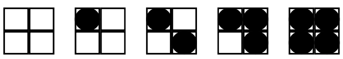
Halftoning & dithering

- mainly used to convert greyscale to binary
  - e.g. printing greyscale pictures on a laser printer
  - 8-bit to 1-bit
- is also used in colour printing, normally with four colours:
  - cyan, magenta, yellow, black

317

Halftoning

- each greyscale pixel maps to a square of binary pixels
  - e.g. five intensity levels can be approximated by a 2×2 pixel square
    - 1-to-4 pixel mapping



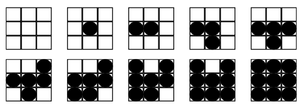
0-51 52-102 103-153 154-204 205-255

8-bit values that map to each of the five possibilities

318

Halftoning dither matrix

- one possible set of patterns for the 3×3 case is:



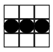
- these patterns can be represented by the *dither matrix*:

7	9	5
2	1	4
6	3	8

- **1-to-9 pixel mapping**

319

Rules for halftone pattern design

- mustn't introduce visual artefacts in areas of constant intensity
  - e.g. this won't work very well: 
- every *on* pixel in intensity level *j* must also be *on* in levels > *j*
  - i.e. *on* pixels form a *growth sequence*
- pattern must grow outward from the centre
  - simulates a dot getting bigger
- all *on* pixels must be connected to one another
  - this is essential for printing, as isolated *on* pixels will not print very well (if at all)

320

Ordered dither

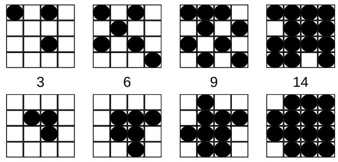
- halftone prints and photocopies well, at the expense of large dots
- an ordered dither matrix produces a nicer visual result than a halftone dither matrix

ordered dither

1	9	3	11
15	5	13	7
4	12	2	10
14	8	16	6

halftone

16	8	11	14
12	1	2	5
7	4	3	10
15	9	6	13



Exercise: phototypesetters may use halftone cells up to size16x16, with 256 entries; either construct a halftone dither matrix for a cell that large or, better, an algorithm to generate an appropriate halftone dither matrix

321

1-to-1 pixel mapping

- a simple modification of the ordered dither method can be used
  - turn a pixel *on* if its intensity is greater than (or equal to) the value of the corresponding cell in the dither matrix

e.g.

quantise 8 bit pixel value

$$q_{i,j} = p_{i,j} \text{ div } 15$$

find binary value

$$b_{i,j} = (q_{i,j} \geq d_{i \bmod 4,\, j \bmod 4})$$

$d_{m,n}$	m: 0	1	2	3
n: 0	1	9	3	11
1	15	5	13	7
2	4	12	2	10
3	14	8	16	6

322

Error diffusion

- error diffusion gives a more pleasing visual result than ordered dither
- method:
  - work left to right, top to bottom
  - map each pixel to the closest quantised value
  - pass the quantisation error on to the pixels to the right and below, and add in the errors before quantising these pixels

323

Error diffusion - example (1)

- map 8-bit pixels to 1-bit pixels
  - quantise and calculate new error values

8-bit value $f_{i,j}$	1-bit value $b_{i,j}$	error $e_{i,j}$
0-127	0	$f_{i,j}$
128-255	1	$f_{i,j} - 255$

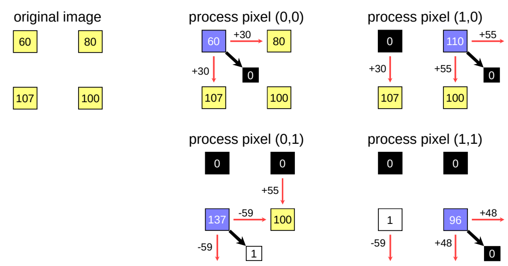
  - each 8-bit value is calculated from pixel and error values:

$$f_{i,j} = p_{i,j} + \tfrac{1}{2} e_{i-1,j} + \tfrac{1}{2} e_{i,j-1}$$

in this example the errors from the pixels to the left and above are taken into account

324

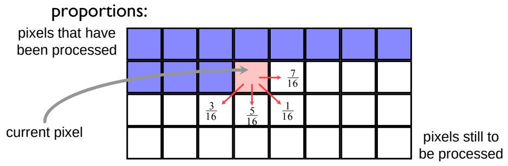
Error diffusion - example (2)



325

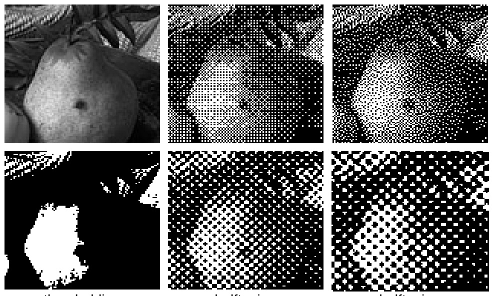
Error diffusion

- Floyd & Steinberg developed the error diffusion method in 1975
  - often called the "Floyd-Steinberg algorithm"
- their original method diffused the errors in the following proportions:



326

Halftoning & dithering — examples



327

Halftoning & dithering — examples

original

halftoned with a very fine screen

ordered dither

the regular dither pattern is clearly visible

error diffused

more random than ordered dither and therefore looks more attractive to the human eye

thresholding

<128 ⇒ black

≥128 ⇒ white

halftoning

the larger the cell size, the more intensity levels available

the smaller the cell, the less noticable the halftone dots

328

Encoding & compression

- introduction
- various coding schemes
  - difference, predictive, run-length, quadtree
- transform coding
  - Fourier, cosine, wavelets, JPEG

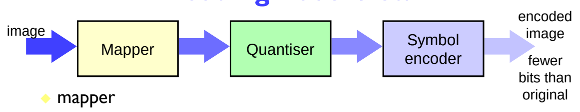
329

What you should note about image data

- there's lots of it!
  - an A4 page scanned at 300 ppi produces:
    - 24MB of data in 24 bit per pixel colour
    - 1MB of data at 1 bit per pixel
      - the Encyclopaedia Britannica would require 25GB at 300 ppi, 1 bit per pixel
- adjacent pixels tend to be very similar
- compression is therefore both feasible and necessary

330

Encoding - overview



- mapper
  - maps pixel values to some other set of values
  - designed to reduce inter-pixel redundancies
- quantiser
  - reduces the accuracy of the mapper's output
  - designed to reduce psychovisual redundancies
- symbol encoder
  - encodes the quantiser's output
  - designed to reduce symbol redundancies

all three operations are optional

331

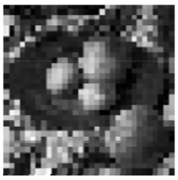
Lossless *vs* lossy compression

- lossless
  - allows you to exactly reconstruct the pixel values from the encoded data
    - implies no quantisation stage and no losses in either of the other stages
- lossy
  - loses some data, you cannot exactly reconstruct the original pixel values

332

Raw image data

- can be stored simply as a sequence of pixel values
  - no mapping, quantisation, or encoding



32×32 pixels

1024 bytes

333

Symbol encoding on raw data

(an example of symbol encoding)

- pixels are encoded by variable length symbols
  - the length of the symbol is determined by the frequency of the pixel value's occurence

e.g.

p	$P(p)$	Code 1	Code 2
0	0.19	000	11
1	0.25	001	01
2	0.21	010	10
3	0.16	011	001
4	0.08	100	0001
5	0.06	101	00001
6	0.03	110	000001
7	0.02	111	000000

with Code 1 each pixel requires 3 bits
with Code 2 each pixel requires 2.7 bits

Code 2 thus encodes the data in 90% of the space of Code 1

334

Quantisation as a compression method

(an example of quantisation)

- quantisation, on its own, is not normally used for compression because of the visual degradation of the resulting image
- however, an 8-bit to 4-bit quantisation using error diffusion would compress an image to 50% of the space

335

Difference mapping

(an example of mapping)

- every pixel in an image will be very similar to those either side of it
- a simple mapping is to store the first pixel value and, for every other pixel, the difference between it and the previous pixel

67	73	74	69	53	54	52	49	127	125	125	126
67	+6	+1	-5	-16	+1	-2	-3	+78	-2	0	+1

336

Difference mapping - example (1)

Difference	Percentage of pixels
0	3.90%
-8..+7	42.74%
-16..+15	61.31%
-32..+31	77.58%
-64..+63	90.35%
-128..+127	98.08%
-255..+255	100.00%

- this distribution of values will work well with a variable length code

337

Difference mapping - example (2)

(an example of mapping and symbol encoding combined)

- this is a very simple variable length code

Difference value	Code	Code length	Percentage of pixels
-8..+7	0XXXX	5	42.74%
-40..-9 +8..+39	10XXXXXX	8	38.03%
-255..-41 +40..+255	11XXXXXXXXX	11	19.23%

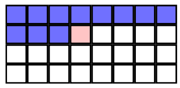
7.29 bits/pixel
91% of the space of the original image

338

Predictive mapping

(an example of mapping)

- when transmitting an image left-to-right top-to-bottom, we already know the values above and to the left of the current pixel
- predictive mapping uses those known pixel values to predict the current pixel value, and maps each pixel value to the difference between its actual value and the prediction



e.g. prediction

$$\tilde{p}_{i,j} = \tfrac{1}{2} p_{i-1,j} + \tfrac{1}{2} p_{i,j-1}$$

difference - this is what we transmit

$$d_{i,j} = p_{i,j} - \tilde{p}_{i,j}$$

339

Run-length encoding

(an example of symbol encoding)

- based on the idea that images often contain runs of identical pixel values
  - method:
    - encode runs of identical pixels as *run length* and *pixel value*
    - encode runs of non-identical pixels as *run length* and *pixel values*

original pixels

34	36	37	38	38	38	38	39	40	40	40	40	40	49	57	65	65	65	65

run-length encoding

3	34	36	37	4	38	1	39	5	40	2	49	57	4	65

340

Run-length encoding - example (1)

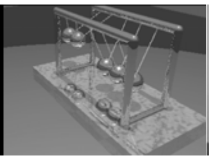
- run length is encoded as an 8-bit value:
  - first bit determines type of run
    - 0 = identical pixels, 1 = non-identical pixels
  - other seven bits code length of run
    - binary value of *run length* - 1 (*run length* ∈{1,…,128})
- pixels are encoded as 8-bit values
- best case: all runs of 128 identical pixels
  - compression of 2/128 = 1.56%
- worst case: no runs of identical pixels
  - compression of 129/128=100.78% (expansion!)

341

Run-length encoding - example (2)

- works well for computer generated imagery
- not so good for real-life imagery
- especially bad for noisy images

19.37%



44.06%

99.76%

compression ratios

342

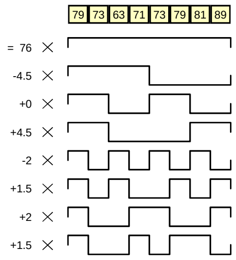
CCITT fax encoding

- fax images are binary
  - transmitted digitally at relatively low speed over the ordinary telephone lines
  - compression is vital to ensuring efficient use of bandwidth
- 1D CCITT group 3
  - binary image is stored as a series of run lengths
  - don't need to store pixel values!
- 2D CCITT group 3 & 4
  - predict this line's runs based on previous line's runs
  - encode differences

343

Transform coding

- transform N pixel values into coefficients of a set of N basis functions
- the basis functions should be chosen so as to squash as much information into as few coefficients as possible
- quantise and encode the coefficients



344

Mathematical foundations

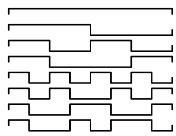
- each of the N pixels, $f(x)$, is represented as a weighted sum of coefficients, $F(u)$

$$f(x) = \sum_{u=0}^{N-1} F(u) H(u,x)$$

H(u,x) is the array of weights

e.g. $H(u,x)$

u \ x	0	1	2	3	4	5	6	7
0	+1	+1	+1	+1	+1	+1	+1	+1
1	+1	+1	+1	+1	-1	-1	-1	-1
2	+1	+1	-1	-1	+1	+1	-1	-1
3	+1	+1	-1	-1	-1	-1	+1	+1
4	+1	-1	+1	-1	+1	-1	+1	-1
5	+1	-1	+1	-1	-1	+1	-1	+1
6	+1	-1	-1	+1	+1	-1	-1	+1
7	+1	-1	-1	+1	-1	+1	+1	-1



345

Calculating the coefficients

- the coefficients can be calculated from the pixel values using this equation:

$$F(u) = \sum_{x=0}^{N-1} f(x) h(x,u)$$

forward transform

- compare this with the equation for a pixel value, from the previous slide:

$$f(x) = \sum_{u=0}^{N-1} F(u) H(u,x)$$

inverse transform

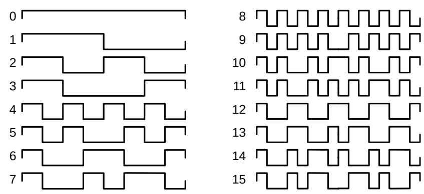
346

Walsh-Hadamard transform

- "square wave" transform
- $h(x,u) = {}^1/_N \, H(u,x)$

the first sixteen Walsh basis functions

(Hadamard basis functions are the same, but numbered differently!)



invented by Walsh (1923) and Hadamard (1893) - the two variants give the same results for N a power of 2

347

2D transforms

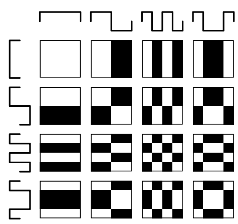
- the two-dimensional versions of the transforms are an extension of the one-dimensional cases

	one dimension	two dimensions
forward transform	$F(u) = \sum_{x=0}^{N-1} f(x) h(x,u)$	$F(u,v) = \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} f(x,y) h(x,y,u,v)$
inverse transform	$f(x) = \sum_{u=0}^{N-1} F(u) H(u,x)$	$f(x,y) = \sum_{u=0}^{N-1} \sum_{v=0}^{N-1} F(u,v) H(u,v,x,y)$

348

2D Walsh basis functions

- these are the Walsh basis functions for N=4
- in general, there are N^2 basis functions operating on an N×N portion of an image



349

Discrete Fourier transform (DFT)

- forward transform:

$$F(u) = \sum_{x=0}^{N-1} f(x) \frac{e^{-i2\pi ux/N}}{N}$$

- inverse transform:

$$f(x) = \sum_{u=0}^{N-1} F(u) e^{i2\pi xu/N}$$

  - thus:

$$h(x,u) = \tfrac{1}{N} e^{-i2\pi ux/N}$$
$$H(u,x) = e^{i2\pi xu/N}$$

350

DFT – alternative interpretation

- the DFT uses complex coefficients to represent real pixel values
- it can be reinterpreted as:

$$f(x) = \sum_{u=0}^{\frac{N}{2}} A(u) \cos(2\pi ux + \theta(u))$$

  - where $A(u)$ and $\theta(u)$ are real values
- a sum of weighted & offset sinusoids

351

Discrete cosine transform (DCT)

- forward transform:

$$f(x) = \sum_{u=0}^{N-1} F(u)\alpha(u)\cos\left(\frac{(2x+1)u\pi}{2N}\right)$$

- inverse transform:

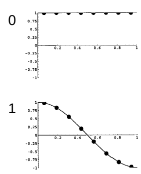
$$F(u) = \sum_{x=0}^{N-1} f(x)\alpha(x)\cos\left(\frac{(2x+1)u\pi}{2N}\right)$$

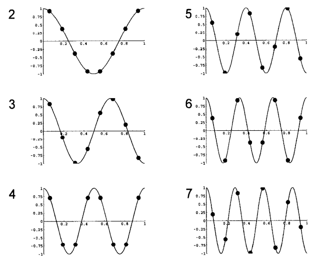
where: $\alpha(z) = \begin{cases} \sqrt{\frac{1}{N}} & z = 0 \\ \sqrt{\frac{2}{N}} & z \in \{1,2,\ldots N-1\} \end{cases}$

352

DCT basis functions

the first eight DCT basis functions showing the values of $h(u,x)$ for $N=8$





353

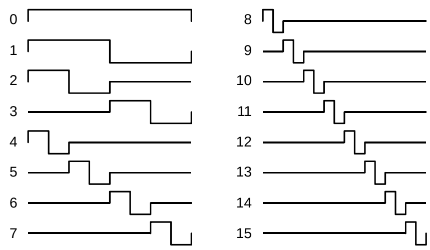
Haar transform: wavelets

- "square wave" transform, similar to Walsh-Hadamard
- Haar basis functions get progressively more local
  - c.f. Walsh-Hadamard, where all basis functions are global
- simplest wavelet transform

354

Haar basis functions

the first sixteen Haar basis functions



355

Karhunen-Loève transform (KLT)

"eigenvector", "principal component", "Hotelling" transform

- based on statistical properties of the image source
- theoretically best transform encoding method
- but different basis functions for every different image source
- if we assume a statistically random image source
  - all images are then equally likely
  - the KLT basis functions are very similar to the DCT basis functions
    - the DCT basis functions are much easier to compute and use
  - therefore use the DCT for statistically random image sources

first derived by Hotelling (1933) for discrete data; by Karhunen (1947) and Loève (1948) for continuous data

356

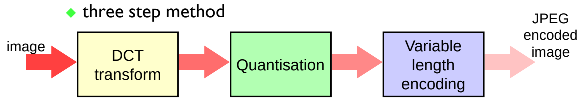
JPEG: a practical example

- compression standard
  - JPEG = Joint Photographic Expert Group
- three different coding schemes:
  - baseline coding scheme
    - based on DCT, lossy
    - adequate for most compression applications
  - extended coding scheme
    - for applications requiring greater compression or higher precision or progressive reconstruction
  - independent coding scheme
    - lossless, doesn't use DCT

357

JPEG sequential baseline scheme

- input and output pixel data limited to 8 bits
- DCT coefficients restricted to 11 bits
- three step method

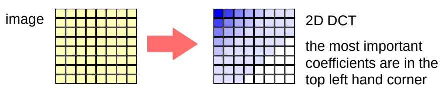


the following slides describe the steps involved in the JPEG compression of an 8 bit/pixel image

358

JPEG example: DCT transform

- subtract 128 from each (8-bit) pixel value
- subdivide the image into 8×8 pixel blocks
- process the blocks left-to-right, top-to-bottom
- calculate the 2D DCT for each block



359

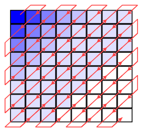
JPEG example: quantisation

- quantise each coefficient, $F(u,v)$, using the values in the quantisation matrix and the formula:

$$\hat{F}(u,v) = \text{round}\left[\frac{F(u,v)}{Z(u,v)}\right]$$

$Z(u,v)$

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99



- reorder the quantised values in a zigzag manner to put the most important coefficients first

360

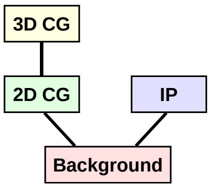
JPEG example: symbol encoding

- the DC coefficient (mean intensity) is coded relative to the DC coefficient of the previous 8x8 block
- each non-zero AC coefficient is encoded by a variable length code representing both the coefficient's value and the number of preceding zeroes in the sequence
  - this is to take advantage of the fact that the sequence of 63 AC coefficients will normally contain long runs of zeroes

361

Course Structure – a review

- Background [3L]
  - images, human vision, displays
- 2D computer graphics [4L]
  - lines, curves, clipping, polygon filling, transformations
- 3D computer graphics [6L]
  - projection (3D→2D), surfaces, clipping, transformations, lighting, filling, ray tracing, texture mapping
- Image processing [3L]
  - filtering, compositing, half-toning, dithering, encoding, compression



362

What next?

- Advanced graphics
  - Modelling, splines, subdivision surfaces, complex geometry, more ray tracing, radiosity, animation
- Human-computer interaction
  - Interactive techniques, quantitative and qualitative evaluation, application design
- Information theory and coding
  - Fundamental limits, transforms, coding
- Computer vision
  - Inferring structure from images

363

And then?

- Graphics
  - multi-resolution modelling
  - animation of human behaviour
  - æsthetically-inspired image processing
- HCI
  - large displays and new techniques for interaction
  - emotionally intelligent interfaces
  - applications in education and for special needs
  - design theory
- http://www.cl.cam.ac.uk/research/rainbow/

Computer Graphics & Image Processing: Exercises

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Introduction

Some of these exercises were written by three computer graphics supervisors in 1997. Subsequently extra exercises have been added, along with all past examination questions up to 2000. Credit for the original set of questions goes to Dr Chris Faigle, Dr James Gain, and Dr Jonathan Pfautz. Any mistakes remain the problem of Professorr Dodgson.

There are four sets of exercises, one for each of the four parts of the course. Each set of exercises contains four sections:

W. Warmups

These are short answer questions that should only take you a few minutes each.

P. Problems

These are geared more towards real exam questions. You should make sure that you can answer these.

E. Old Exam Problems

These are relevant problems from exams before 2000. You should definitely make sure that you can answer these. All exam questions from 2000 onwards are relevant also. These may be found on the Lab's website.

F. Further Problems

These are harder problems that will be worth your while to answer if you find the other stuff too easy.

Information for supervisors

Be selective: do not set your supervisees every single question from each part — that would overload them.

Do not feel that you have to give one supervision on each part: choose your questions with regard to when your supervisions fall relative to the lectures.

You can specify questions as: Part/Section/Question. For example: 2/W/1.

Solution notes. Starred exercises (★) have *solution notes*. Doubly starred exercises (★★) (these are all pre-2000 exam questions) have *model answers* marked by Dr Dodgson according to the official marking scheme. Some post-2000 exam questions have *solution notes*. All of these are available ***to supervisors only*** from the student administration office.

Part 1: Background

W. Warmups

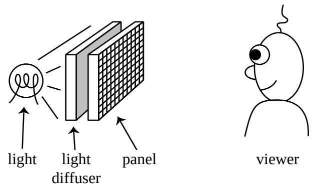
1. *[moved elsewhere]*
2. ★ Suppose you are designing a user interface for someone who is colour blind. Describe how some user interface of your choice should be suitably modified.
3. ★ Why is it better to look at stars and comets slightly off-centre rather than looking directly at them?
4. In a CAD system blue would be a poor choice for the colour of an object being designed. Why is this?
5. ★ In New Zealand, warning road signs are black on yellow, it being alleged that this is easier to see than black on white. Why might this be true?

P. Problems

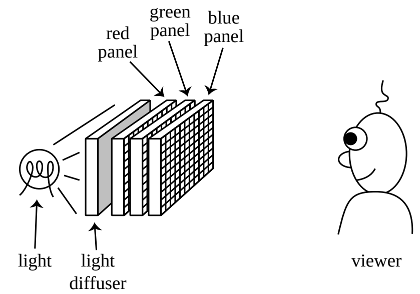
1. ★ **Colour Spaces.** Explain the use of each of the following colour spaces: (a) RGB; (b) XYZ; (c) HLS; (d) Luv
2. **Monitor Resolution.** Calculate the ultimate monitor resolution (i.e. colour pixels/inch) at which point better resolution will be indistinguishable.
3. **CRTs.** Explain the basic principles of a Cathode Ray Tube as used for television.
4. **Pixels.** Why do we use square pixels? Would hexagonal pixels be better? What about triangles? Do you see any difficulties building graphics hardware with these other two schemes?
5. **Additive *vs* Subtractive.** Explain the difference between additive colour (RGB) and subtractive colour (CMY). Where is each used and why is it used there?

E. Old Exam Problems

1. ★ [94.3.8] (a) Explain how a shadow mask cathode ray tube works. What do you think some of the manufacturing difficulties might be? (b) What might be the point of extending the scheme to accommodate five electron guns?
2. ★★ [97.6.4 *first part*] It is convenient to be able to specify colours in terms of a three-dimensional co-ordinate system. Three such co-ordinate systems are: RGB, HLS, L*a*b*.
   Choose *two* of these three co-ordinate systems.
   For *each* of your chosen two:
   (a) describe what each of the three co-ordinates represents
   (b) describe why the co-ordinate system is a useful representation of colour
3. ★★ [98.6.4 *first part*] An inventor has recently developed a new display device: it is a transparent panel with a rectangular array of square pixels. The panel is tinted with a special ink which allows each pixel to range from totally transparent to transmitting only the colour of the ink. Each pixel has an 8-bit value. For example, if the ink is blue then a pixel value of 0 would be totally transparent, 255 totally blue (only blue light transmitted) and 100 a light blue.



The inventor has recently found that he can make the special ink in *any* colour he likes, but that each panel can be tinted with only one of these colours. He proposes to use three inks in three panels to make a 24-bit colour display: a red-tinted panel, a green-tinted panel and a blue-tinted panel will be stacked up to make a full-colour display (see picture). A value of (0,0,0) will thus be white (transparent), (255,0,0) red and (255,255,255) black.



Explain why this will not work. [4]
Modify the three-panel design so that it will work. [3]
In common with other 24-bit "full-colour" displays (for example CRT, LCD), your display *cannot* display *every* colour which a human can perceive. Why not? [3]

4. ★★ [99.5.4 *first part*] A company wishes to produce a greyscale display with pixels so small that a human will be unable to see the individual pixels under normal viewing conditions. What is the minimum number of pixels per inch required to achieve this? Please state all of the assumptions that you make in calculating your answer. [Note: it may be helpful to know that there are 150 000 cones per square millimetre in the human fovea, and that there are exactly 25.4 millimetres in an inch. [6]
   If the pixels could be only black or white, and greyscale was to be achieved by halftoning, then what would the minimum number of pixels per inch be in order that a human could not see the halftone dots? Again, state any assumptions that you make. [2]

F. Further Problems

1. ★ **Liquid Crystal Displays**. Explain how a liquid crystal display works.
2. ★ **Head Mounted Display**. Ivan Sutherland described, in 1965, the "Ultimate Display". This display was to match all of the human senses so that the display images were indistinguishable from reality. He went on to build the world's first head-mounted display (HMD) over the next few years. Needless to say, he was far from accomplishing his goal.
   You are to work out the flat-screen pixel resolution (height and width) necessary for an Ultimate HMD (that is, so that the display seems to match the real world) given the following information:
   viewing distance: 10 cm
   human visual acuity: 1 minute of visual arc
   human vertical field of view: 100°
   human horizontal field of view: 200°
   Give at least five assumptions that have been made in your calculations (and in the question itself!).
3. ★ Why is the sky blue? [Hints: Why *might* it be blue? Why are sunsets red? Are the red of a sunset and the blue of the sky related?]
4. Printing. Select one of (a) laser printing, (b) ink jet printing, (c) offset printing. Find out how it works and write a 1000 word description which a 12 year old could understand.
5. Displays. Find out in detail and explain how either (a) a plasma display or (b) a DMD display works.

Part 2: 2D Computer Graphics

W. Warmups

1. ★ **GUIs**. How has computer graphics been affected by the advent of the graphical user interface?
2. **Matrices**. Give as many reasons as possible why we use matrices to represent transformations. Explain why we use homogeneous co-ordinates.
3. **BitBlt**. What factors do you think affect the efficiency of BitBlt's ability to move images quickly?

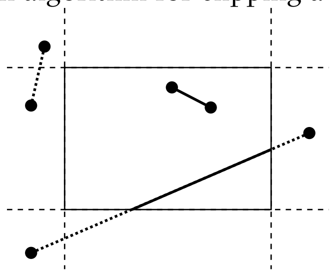
P. Problems

1. ★ **Circle Drawing.** Give the circle drawing algorithm, in detail, for the *first* octant. Prove that it works by checking it on a circle of radius 6.
2. **Oval Drawing**. (a) In slide 124 of the notes Dr. Dodgson writes that one must use points of 45° slope to divide the oval into eight sections. Explain what is meant by this and why it is so.
3. **Line Drawing.** On paper run Bresenham's algorithm (slide 112) for the line running from (0,0) to (5,3).
4. ★ **Subdivision**. If we are subdividing Bézier curves in order to draw them, how do we know when the curve is within a given tolerance? (i.e. what piece of mathematics do we need to do to check whether or not we are within tolerance?)
5. **Bézier cubics.** Derive the conditions necessary for two Bézier curves to join with (a) just *C0*-continuity; (b) *C1*-continuity; (c) *C2*-continuity. Why would it be difficult (if not impossible) to get three Bézier curves to join in sequence with *C2*-continuity at the two joins?
6. **Triangle drawing.** Describe, in detail, an algorithm to fill a triangle. Show that it works using the triangle with vertices (0,0), (5,3) and (2,5).
7. **Triangle drawing.** Implement an algorithm to fill triangles. Demonstrate it working.
8. **Bézier cubics.** Implement the Bézier curve algorithm on slides 133–137. Demonstrate it working.
9. **Lines & circles.** Implement the Midpoint line and circle drawing algorithms. Demonstrate them working.

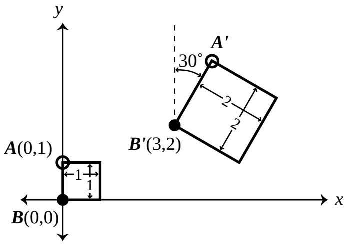
E. Old Exam Problems

1. [96.5.4] Consider the control of detail in a curve represented as a sequence of straight line segments. Describe how Douglas and Pücker's algorithm might be used to remove superfluous points.
   Describe how Overhauser interpolation can be used to introduce additional points.
2. ★ [95.5.4] Why are matrix representations used to describe point transformations in computer graphics?
   Describe how to represent three different 2D transformations as matrices.
   Explain how to derive a sequence of transformations to achieve the overall effect of performing a 2D rotation about an arbitrary point.
3. ★★ [97.4.10] Describe an algorithm to draw a straight line using only integer arithmetic. You may assume that the line is in the first octant, that the line starts and ends at integer co-ordinates, and that the function *setpixel(x,y)* turns on the pixel at location *(x,y)*.
   Explain how straight lines can be used to draw Bézier cubic curves.

4. ★★ [98.5.4] Describe an algorithm for clipping a line against a rectangle. [9]



Show that it works using the above three examples. [3]



The above diagram shows a complicated 2D transformation applied to a unit square. The overall transformation can be described in terms of a number of simpler transformations. Describe each of these simple transformations and give a matrix representation of each using homogeneous coordinates. [6]
Use the matrices from the previous part to find the (x,y) coordinates of point A', the image of point A under the overall transformation. [2]

5. ★★ [99.4.10 *first part*] Give the formula for a Bézier cubic curve. Derive the conditions necessary for two Bézier cubic curves to join with (i) just C0-continuity and (ii) C1-continuity. Give a geometric interpretation of each condition in terms of the locations of control points. Explain (mathematically) why a Bézier cubic curve is guaranteed to lie within the convex hull of its control points. [8]
6. ★★ [99.6.4 *second part*] Describe an algorithm which clips an arbitrary polygon against an arbitrary *convex* polygon (in 2D). [8]
   Will your algorithm correctly clip an arbitrary polygon against an arbitrary *non-convex* polygon? If so, explain why and demonstrate an example which illustrates that it does work in such cases. If not, explain why not and outline how your algorithm could be extended to perform clipping in such [4]

F. Further Problems

1. **Bernstein Polynomials.** Prove that the Bernstein polynomials sum to 1.
2. **Polygon Filling.** Explain *in detail* how a polygon filling algorithm works including details of how to interpolate along the edges and how to maintain the necessary data structures.
3. **Polygon Filling.** Implement the polygon filling algorithm, including clipping the polygon, and demonstrate it working on a number of polygons.
4. **Bézier cubics.** An alternative way of implementing cubics would be to modify the algorithm on slide 134 to say:

```
IF ( P0 and P3 are less than a pixel apart ) THEN
        draw that pixel
ELSE…
```

   Implement both this algorithm and the algorithm on slide 134 and compare their output.

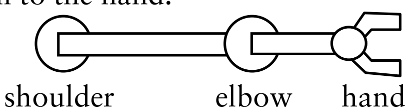
Part 3: 3D Computer Graphics

W. Warmups

1. **Texture Mapping.** Are there any problems with texture mapping onto a sphere?
2. *<3/W/2 deleted in 1999 because the course no longer covers this material>*
3. ★ **Shading Model.** Do you see any problems with the naïve shading model? Can you suggest anything to improve its faults?
4. ★ **Illumination.** Describe the following in terms of illumination: ambient, diffuse, specular.
5. **Phong.** Describe how Phong's specular reflection models real specular reflection. Why is it only a rough approximation? Why is it useful?
6. **Level of Detail.** You are given several representations of each object in your 3D scene. When you render the scene, you know all of the world/viewing parameters necessary. Describe several (at least three) ways to determine which representation to choose for each object.
7. ★ **Coordinate Systems.** Draw pictures to show what is meant by:
   object coordinates,
   world coordinates,
   viewing coordinates,
   screen coordinates.
8. ★ **Projections.** Illustrate the difference between orthographic (parallel) and perspective projection.
9. ★ **Triangle mesh approximations.** We use a lot of triangles to approximate stuff in computer graphics. Why are they good? Why are they bad? Can you think of any alternatives?

P. Problems

1. **Projection.** Draw and explain two different scenes which have the same projection as seen by the viewer. What other cues can you give so that the viewer can distinguish the depth information.
2. **3D Transformations.** The Rainbow Graphics Group has available a simple robot arm. This robot arm has, for the purposes of this question, the following characteristics:
   • Two joints, a shoulder joint and an elbow joint. The shoulder joint is connected by a 2-unit length upper arm to the elbow joint, which in turn is connected by a single unit lower arm to the hand.



   • The joints can only rotate in the *xy*-plane.
   • The shoulder joint is attached to a *z*-axis vertical slider, which can raise or lower the entire robot arm, but is not able to translate it in the *xy*-plane.
   • The initial position of the arm is:
      shoulder joint: position (0,0,0), rotation 0°
      elbow joint: position (2,0,0), rotation 0°
      hand: position (3,0,0)

   There is a soft drink can located at position (1,1,1). The robot hand must touch this can. Specify the transformation matrices of the joints needed to achieve this.
3. **Perspective Projection Geometry**. (a) Give a matrix, in homogeneous co-ordinates, which will project 3D space onto the plane $z=d$ with the origin as centre of projection. (b) Modify this so that it gives the same x and y as in (a), but also gives

1/z as the third co-ordinate. (c) Now give a matrix, based on the one in (a), which projects onto the plane $z=d$ with an arbitrary point as centre of projection.

4. ★ **Bounding Volumes**. For a cylinder of radius 2, with endpoints (1,2,3) and (2,4,5), show how to calculate: (a) an axis-aligned bound box, (b) a bounding sphere.
5. **Depth Interpolation.** Prove that depth interpolation is correct as given in the notes.
6. ★ **BSP Tree**. Break down the following (2D!) lines into a BSP-tree, splitting them if necessary.
   (0, 0)→(2, 2), (3,4)→(1, 3), (1, 0)→(-3, 1), (0, 3)→(3, 3) and (2, 0)→(2, 1)
7. ★ **Sphere Subdividing**. We often use triangles to represent a sphere. Describe two methods of generating triangles from a sphere.
8. ★ **Compare and Contrast**. Compare and contrast: *texture mapping*, *bump mapping* and *displacement mapping* (you will need to do a bit of background reading).
9. **Shading**. Develop pseudocode for algorithms that shade a triangle according to:
   (a) Gouraud shading.
   (b) Phong shading.
   You can use the scan line polygon fill algorithm as a starting point. Detail your inputs and outputs explicitly.
10. **Rendering**. Compare the computation and memory requirements of the following rendering algorithms:
    (a) z-Buffer
    (b) Ray tracing
    State explicitly any assumptions that you make (e.g. the resolution of the screen). Present your results both numerically and in graph form (with number of polygons on the x-axis).
11. ★ **Bézier Joins**. Explain the three types of joins for Bézier curves and Bézier patches.
12. ★ **3D Clipping**. (a) Compare the two methods of doing 3D clipping in terms of efficiency. (b) How would using bounding volumes improve the efficiency of these methods?
13. ★ **Rotation**.
    Show how to perform 2D rotation around an arbitrary point.
    Show how to perform 3D rotation around an arbitrary axis parallel to the x-axis.
    Show how to perform 3D rotation around an arbitrary axis.
14. **3D Polygon Scan Conversion**
    Describe a complete algorithm to do 3D polygon scan conversion, including details of clipping, projection, and the underlying 2D polygon scan conversion algorithm.

E. Old Exam Problems

1. ★ [96.6.4] What are *homogeneous coordinates*? How can they be used in computer graphics to model (a) translation? and (b) simple perspective?
2. ★ [95.4.8] (a) Explain the purpose of the A-buffer in rendering a sequence of images into the framestore. (b) Exhibit an example that shows an advantage over the use of a z-buffer.
3. ★ [95.6.4] In ray tracing a large computational cost is associated with determining ray-object intersections. Explain how the use of bounding volumes and space subdivision methods may reduce this cost.
4. ★ [94.5.4] (a) Discuss sampling artifacts and their effect on image quality on a raster display. (b) What can be done to reduce or eliminate them?
5. ★★ [97.5.2] Describe the z-buffer polygon scan conversion algorithm.
   Explain how the A-buffer improves on the z-buffer.

6. ★★ [98.4.10] In ray tracing, *ambient*, *diffuse* and *Phong's specular* shading can be used to define the colour at a point on a surface. Explain what each of the three terms refers to, and what real effect each is trying to model. [9]

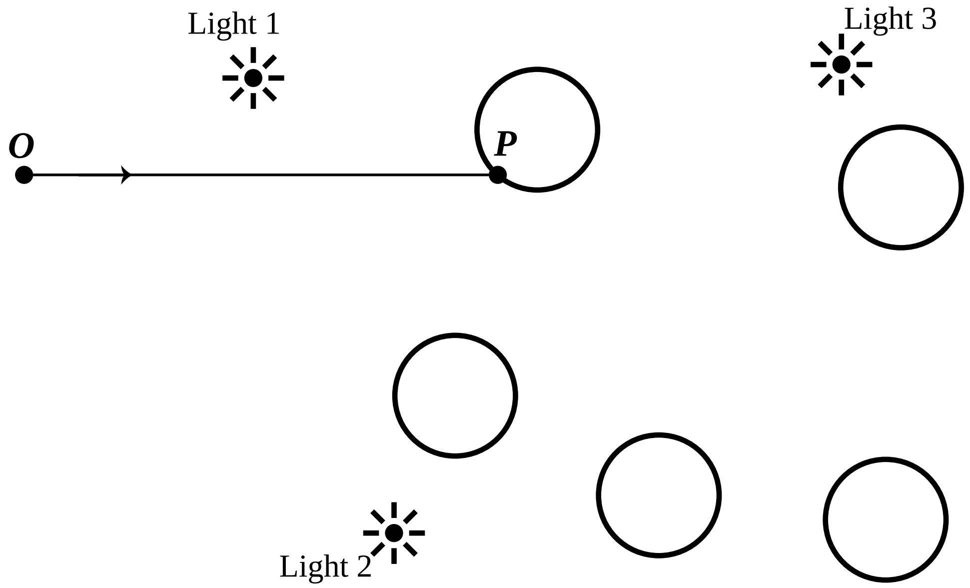
The diagram below represents a scene being ray traced. The circles may be taken to represent the cross-sections of spheres. In answering the remaining parts of this question you should use copies of the diagram below (see page 10).

A particular ray from the eyepoint ***O*** has been found to have its closest intersection with an object at point ***P***. Show, on a diagram, all subsequent rays and vectors which must be found in order to calculate the shading at point ***P***. Explain the purpose of each one.

Assume that:

- each object has ambient, diffuse and specular reflections, but is *not* a perfect reflector
- each object is opaque
- all rays and vectors lie in the plane of the paper
- we are *not* using distributed ray tracing [8]

Assume now that all of the objects are perfect reflectors (in addition to having ambient, diffuse and specular reflection). Show, on a separate diagram, the extra rays which need to be calculated and explain the purpose of each one. [3]



[Note: in 1998 about half of the attempts at this question were wildly wrong. Please check your answer to ensure that a computer would actually be able to perform the calculations that you draw on your diagrams.]

7. ★★ [99.4.10 *second part*] Basic ray tracing uses a single sample per pixel. Describe four distinct reasons why one might use multiple samples per pixel. Explain the effect that each is trying to achieve, and outline the mechanism by which it achieves the effect. [8]

   Describe the differences in the computational complexity of the depth sort and binary space partition (BSP) tree algorithms for polygon scan conversion. If you were forced to choose between the two algorithms for a particular application, what factors would be important in your choice [4]

8. ★★ [99.6.4 *first part*] You have a cone of height one unit; apex at the origin; and base of diameter one unit centred on the negative z-axis. You wish to transform this cone so that the apex is at (-1,3,7), the base is centred at (2,7,-5), and the base's radius is four units. What transformations are required to achieve this and in what order should they be performed? [8]

F. Further Problems

1. **Snell's Laws**. Look up Snell's laws and address how they relate to ray tracing.
2. **Ray Tracing Bézier Patches**. Can you suggest how we might ray-trace a Bézier patch?
3. **Bézier Patches**. Describe how you would form a good approximation to a cylinder from Bézier patches. Draw the patches and their control points and give the co-ordinates of the control points.
4. **Bézier Patches**. Given the following sixteen points, calculate the first eight of the next patch joining it as t increases so that the join has continuity $C1$. Here the points are listed with $s=0$, $t=0$ on the bottom left, with s increasing upwards and t increasing to the right.

(-.2, 3.4, .3)	(1, 3.1, -.2)	(2, 2.6, -.2)	(3.1, 2.8, .2)
(0, 1.2, .4)	(1.2, 2.0, 1.2)	(1.4, 1.9, -.2)	(2.7, 1.8, .2)
(.2, 1, -.2)	(1.1, .8, .5)	(1.4, 1.0, 0)	(3.1, 1.1, -.2)
(0, 0, 0)	(1, 0, .5)	(2, .2, .4)	(2.7, 0, -.2)

5. **Web 3D Language**. Describe some features that you think might be important in a language designed to describe 3D scenes and be used across the web.
6. **Rotations**. Define and then compare and contrast the following methods of specifying rotation in 3D. [you will need to look these up]
   (a) Quaternions
   (b) Euler Angles
7. **DOOM-Style Rendering**. Describe the enhancements and restrictions that the DOOM rendering engine employs to improve efficiency in the interests of interactivity. Define any terms (such as texture-mapping) that you might use. A useful starting point for your research is <http://www.gamers.org/dEngine/doom/>.
8. **Improved shading models**. Find out about the Cook-Torrance shading model and explain how this improves on the naïve diffuse+specular+ambient model (slide 251).
9. **Improved shading models**. Find out about the bi-directional reflectance distribution function (BRDF) and explain how this improves on the naïve diffuse+specular+ambient model (slide 251).

Part 4: Image Processing

P. Problems

1. **Image Coding Schemes**. Describe the following image encoding schemes:
   (a) GIF
   (b) JPEG
   Show their operation with a suitable small sample image. Compare and contrast them (note that GIF file encoding is not in the notes so this requires some research).
2. **Image Coding**. Explain each of the three stages of image coding, as presented in the notes, and why each helps to reduce the number of bits required to store the image.
3. **Error Diffusion**. Compare the two methods of Error Diffusion described in the notes, with the aid of a sample image.

E. Old Exam Questions

1. ★ [96.4.10] An image processing package allows the user to design 3 × 3 convolution filters. Design 3 × 3 filters to perform the following tasks:
   (a) Blurring
   (b) Edge detection of vertical edges
   Choose one of the two filters (a) or (b) from the previous part. Explain how it works, using the following image as an example (you may round off any calculated values to the nearest integer).

100	100	100	0	0	0
100	100	100	0	0	0
100	100	100	0	0	0
100	100	100	100	100	100
100	100	100	100	100	100
100	100	100	100	100	100

2. ★★ [97.6.4 *second part*] Draw *either*:
   (i) the first eight one-dimensional Haar basis functions
   or
   (ii) the first eight one-dimensional Walsh-Hadamard basis functions
   Calculate the co-efficients of your eight basis functions from the previous part for the following one-dimensional image data:
   12 16 20 24 24 16 8 8
   Explain why, in general, the Haar or Walsh-Hadamard encoded version of an image is preferable to the original image for storage or transmission.
3. ★★ [98.6.4 *second part*] In image compression we utilise three different mechanisms to compress pixel data:
   (a) mapping the pixel values to some other set of values
   (b) quantising those values
   (c) symbol encoding the resulting values
   Explain each mechanism, why it helps us to compress the image, and whether (giving reasons) the resulting image noticeably differs. [10]
4. ★★ [99.5.4 *second part*] A company produces a display device with two-bit greyscale (that is: four different shades of grey). Describe an error-diffusion algorithm which will convert an eight-bit greyscale image into a two-bit image suitable for display on this device. [Note: the two images must have the same number of pixels.] [7]
   Illustrate that your algorithm works using the following test image. [2]

200	40
250	220

   You are asked to design a 4×4 ordered dither matrix. What rules will you follow in the design? [3]

F. Further Problems

1. JPEG2000. Find out how the wavelet-based JPEG2000 image compression method works and write a concise description.

Copies of the diagram for answering 3/E/6 [96.4.10]