Computer Graphics & Image Processing

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Computer Science Tripos Part IB

Neil Dodgson & Peter Robinson

Michaelmas Term 2014

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. 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.

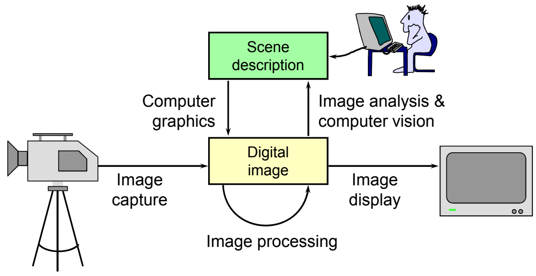
Computer Graphics & Image Processing

1

- Sixteen lectures for Part IB CST
  - Background
  - Simple rendering
  - Graphics pipeline
  - Underlying algorithms
  - Colour and displays
  - Image processing
- Four supervisions suggested
- 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…

Course Structure

5

- Background [2L]
  - images, colour, human vision, resolution
- Simple rendering [2L]
  - perspective, surface reflection, geometric models, ray tracing
- Graphics pipeline [4L]
  - polygonal models, transformations, projection (3D→2D), hardware and OpenGL, lighting and shading, texture
- Underlying algorithms [4L]
  - drawing lines and curves, clipping, filling, depth, anti-aliasing
- Colour and displays [2L]
- Image processing [2L]
  - filtering, compositing, half-toning, dithering, encoding

Course books

6

- *Fundamentals of Computer Graphics*
  - Shirley & Marschner
    CRC Press 2009 (3rd edition)
- *Computer Graphics: Principles & Practice*
  - Hughes, van Dam, McGuire, Skalar, Foley, Feiner & Akeley
    Addison-Wesley 2013 (3rd edition)
- *Computer Graphics & Virtual Environments*
  - Slater, Steed, & Chrysanthou
    Addison Wesley 2001
- *Digital Image Processing*
  - Gonzalez & Woods
    Prentice Hall 2007 (3rd edition)

7

Computer Graphics & Image Processing

- Background
  - Digital images
  - Lighting and colour
  - Human vision
- Simple rendering
- Graphics pipeline
- Underlying algorithms
- Colour and displays
- Image processing

8

Background

- what is a digital image?
  - what are the constraints on digital images?
- how does human vision work?
  - what are the limits of human vision?
  - what can we get away with given these constraints & limits?
- what are the implications?

Later on in the course we will ask:

- how do we represent colour?
- how do displays & printers work?
  - how do we fool the human eye into seeing what we want?

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

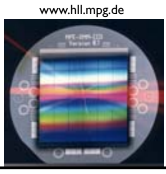
11

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
    - CMOS camera chips

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

area CCD
www.hll.mpg.de

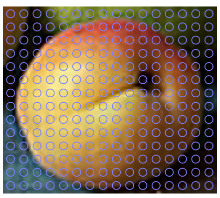


flatbed scanner
www.nuggetlab.com

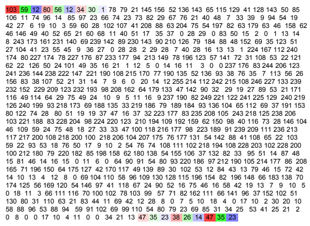
Heidelberg
drum scanner

12

Image capture example



A real image



A digital image

13

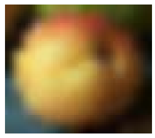
Image display

- a digital image is an array of integers, how do you display it?
- reconstruct a real image on some sort of display device
  - LCD — portable computer, video projector
  - DMD — video projector
  - EPS – electrophoretic display “e-paper”
  - printer — ink jet, laser printer, dot matrix, dye sublimation, commercial typesetter

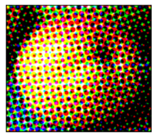
14

Different ways of displaying the same digital image

Nearest-neighbour
e.g. LCD



Gaussian
e.g. cathode ray tube



Half-toning
e.g. inkjet printer

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

15

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

16

Sampling resolution

256×256 128×128 64×64 32×32

2×2 4×4 8×8 16×16

17

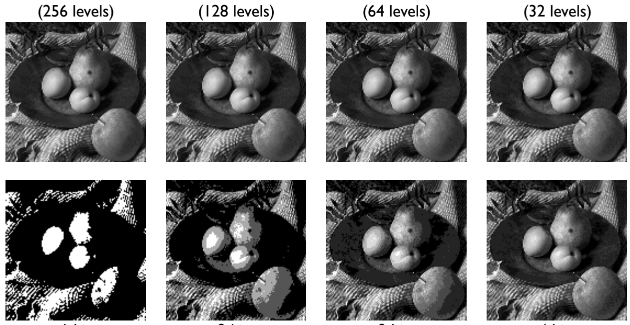
Quantisation

- 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

18

Quantisation levels

8 bits (256 levels) | 7 bits (128 levels) | 6 bits (64 levels) | 5 bits (32 levels)

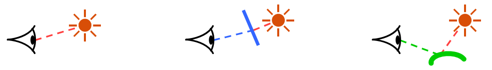


1 bit (2 levels) | 2 bits (4 levels) | 3 bits (8 levels) | 4 bits (16 levels)

19

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



20

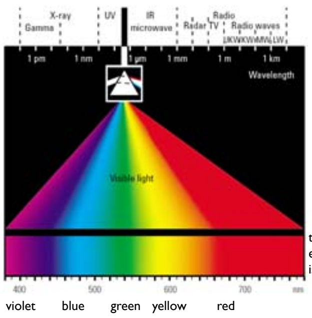
Light: wavelengths & spectra

- 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

21

The spectrum

visible light is only a tiny part of the whole electromagnetic spectrum

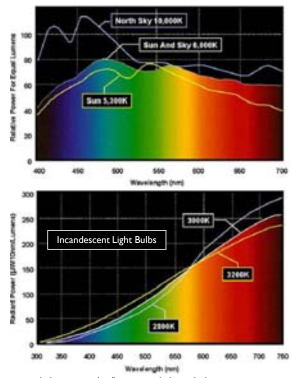


the short wavelength end of the spectrum is violet

the long wavelength end of the spectrum is red

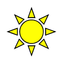
22

Illuminants have different characteristics



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

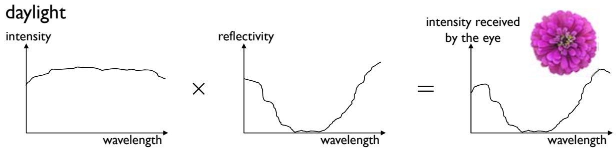
- different lights emit different intensities of each wavelength
  - sunlight is reasonably uniform
  - incandescent light bulbs are very red
  - sodium street lights emit almost pure yellow



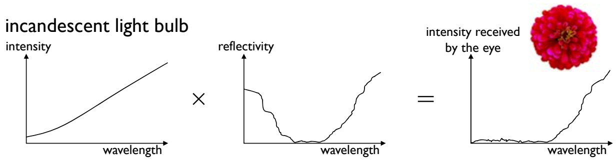
23

Illuminant × reflection = reflected light

daylight



incandescent light bulb



24

incandescent light bulb — camera flash bulb

winter sunlight — halogen light bulbs (overhead)

Comparison of illuminants

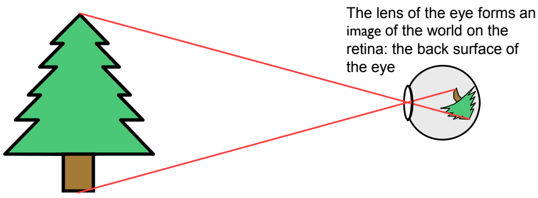
compare these things:

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

25

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...



26

Structure of the human eye

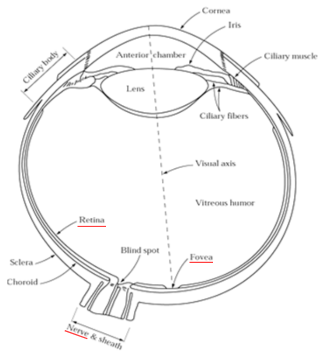


Fig. 2.1 from Gonzalez & Woods

- 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

27

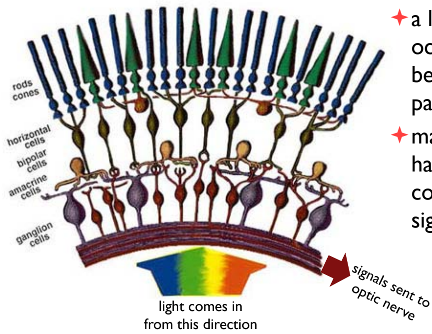
The retina

- 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

28

Detailed structure of retinal processing



www.phys.ufl.edu/~avery/course/3400/vision/retina_schema.jpg

- 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

29

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

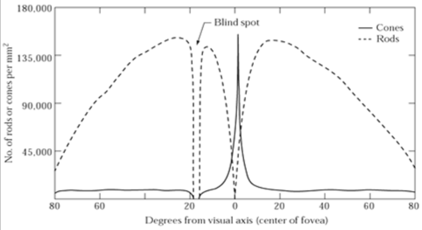
30

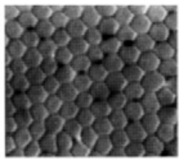
Foveal vision

- 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

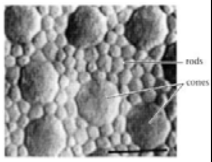
31

Distribution of rods & cones





cones in the fovea



rods & cones outside 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

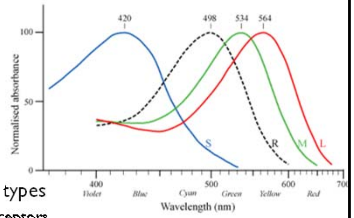
Fig. 2.2 from Gonzalez & Woods
www.cis.rit.edu/people/faculty/montag/vandplite/pages/chap_9/ch9p1.html

32

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: $l(\lambda)$, $m(\lambda)$, $s(\lambda)$
- 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

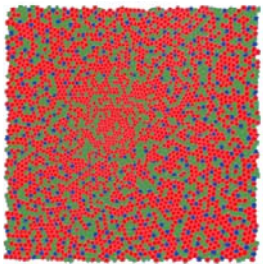
$r(\lambda)$ is the response function of the rods



33

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

34

Colour signals sent to the brain

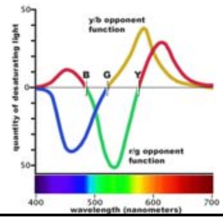
- the signal that is sent to the brain is pre-processed by the retina

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



35

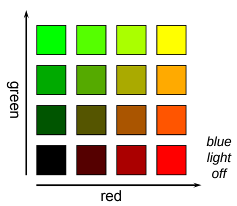
Chromatic metamerism

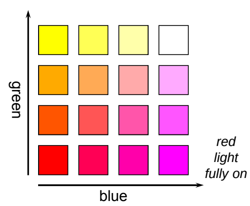
- many different spectra will induce the same response in our cones
  - the values of the three perceived values can be calculated as:
    - $l = k \int P(\lambda)\, l(\lambda)\, d\lambda$
    - $m = k \int P(\lambda)\, m(\lambda)\, d\lambda$
    - $s = k \int P(\lambda)\, s(\lambda)\, d\lambda$
  - 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

36

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

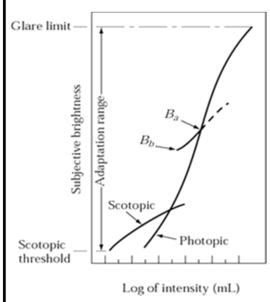
37

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

38

Intensity adaptation



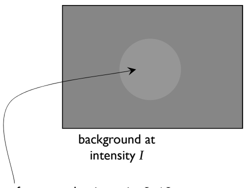
- at any one time the eye can handle intensities over a range of ~100:1
  - this is the curve B_bB_a
  - anything darker is seen as black
    - if everything is black, the eye adjusts down
  - anything brighter causes pain
    - and stimulates the eye to adjust up
- the eye can adjust over a range of 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

Fig. 2.4 from Gonzalez & Woods

39

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*



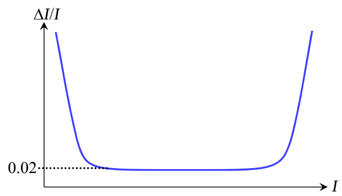
for a range of values of I

- start with ΔI=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

40

Intensity differentiation

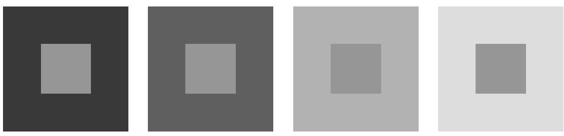
- results for a "normal" viewer
  - a human can distinguish about a 2% change in intensity for much of the range of intensities
  - discrimination becomes rapidly worse as you get close to the darkest or brightest intensities that you can currently see



41

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

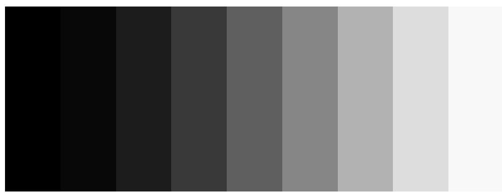


The centre square is the same intensity in all four cases but does not appear to be because your visual system is taking the local contrast into account

42

Mach bands

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

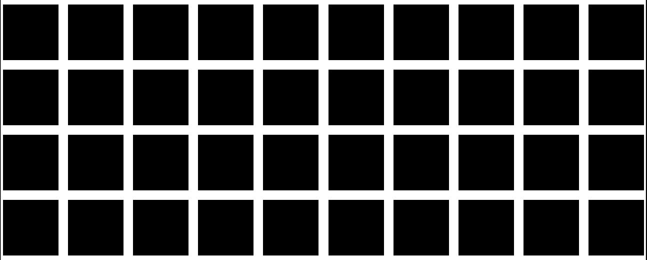


Each of the nine rectangles is a constant colour but you will see each rectangle being slightly brighter at the end which is near a darker rectangle and slightly darker at the end which is near a lighter rectangle

43

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"



44

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

45

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 lower than intensity resolution
  - this is exploited in video encoding
    - the colour information in analogue television has half the spatial resolution of the intensity information
    - the colour information in digital television has less spatial resolution and fewer quantisation levels than the intensity information

46

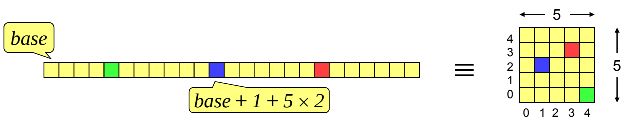
Implications of vision on quantisation

- 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 (LCD or DLP) have static contrast ratios quoted in the thousands
    - actually in the hundreds other in a completely dark room
  - 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

47

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



48

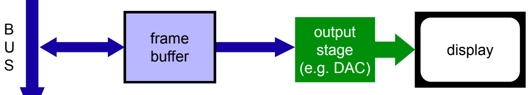
Colour images

- 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

49

The frame buffer

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

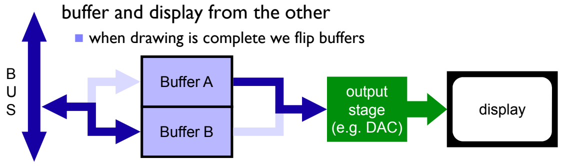


- the frame buffer normally consists of dual-ported Dynamic RAM (DRAM)
  - sometimes referred to as Video RAM (VRAM)

50

Double buffering

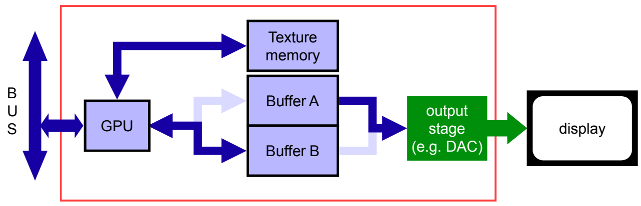
- if we allow the currently displayed image to be updated then we may see bits of the image being displayed halfway through the update
  - this can be visually disturbing, especially if we want the illusion of smooth animation
- double buffering solves this problem: we draw into one frame buffer and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



51

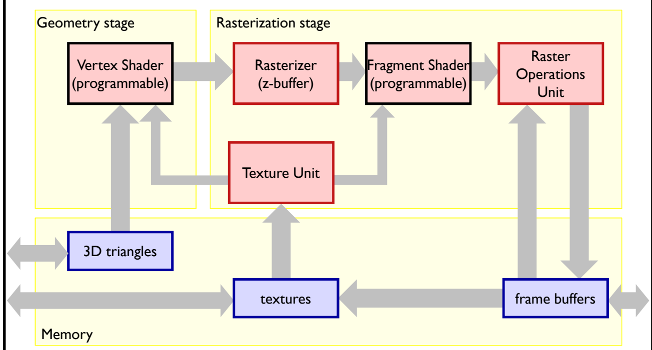
Modern graphics cards

- 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



52

Graphics card architecture



based on nVIDIA's GeForce 6 architecture

53

54

55

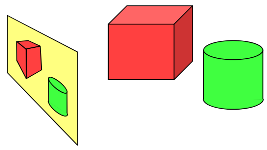
Computer Graphics & Image Processing

- Background
- Simple rendering
  - Projection, depth and perspective
  - Reflection from surfaces
  - Ray tracing
- Graphics pipeline
- Underlying algorithms
- Colour and displays
- Image processing

56

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



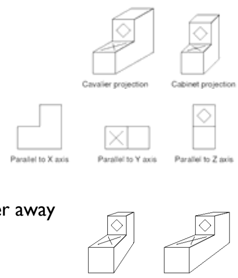
The 3D world is described as a set of (mathematical) objects

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

57

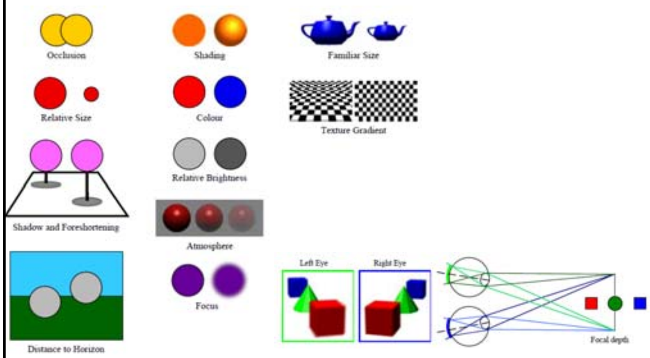
Types of projection

- 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



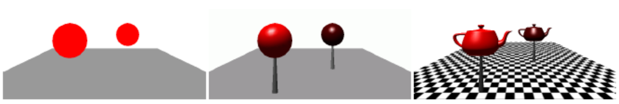
58

Depth cues



59

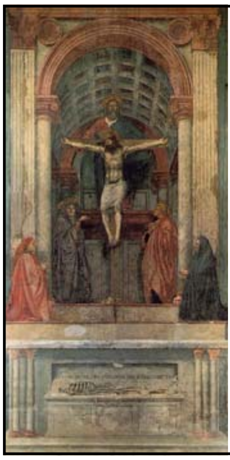
Rendering depth



60

Wrong perspective

- Adoring saints
- Lorenzo Monaco 1407-09
- National Gallery London



Perspective

- Holy Trinity fresco
- Masaccio (Tommaso di Ser Giovanni di Simone) 1425
- Santa Maria Novella Florence

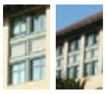
More perspective

- The Annunciation with Saint Emidius
- Carlo Crivelli 1486
- National Gallery London

Perspective projection examples

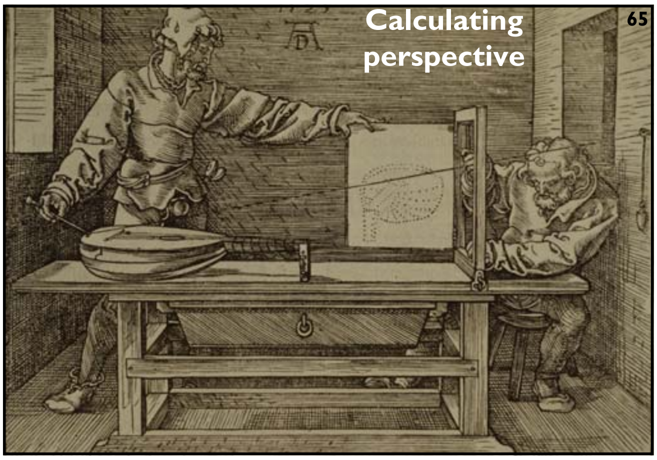
Gates Building – the rounded version (Stanford)

Gates Building – the rectilinear version (Cambridge)



False perspective

Calculating perspective



Illumination and shading

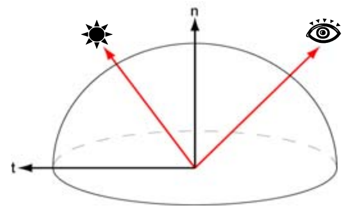
- Dürer's method allows us to calculate what part of the scene is visible in any pixel
- But what colour should it be?
- Depends on:
  - lighting
  - shadows
  - properties of surface material

67

Different materials have different reflectances

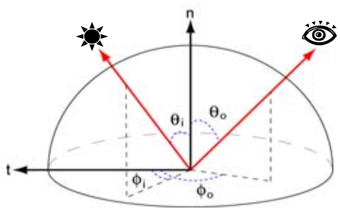
BRDF

- Bidirectional Reflectance Distribution Function



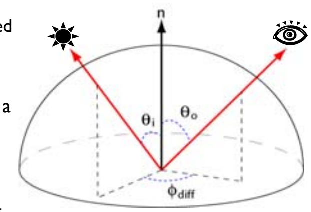
BRDF

- Bidirectional Reflectance Distribution Function
  - $\rho(\theta_i, \phi_i ; \theta_o, \phi_o)$

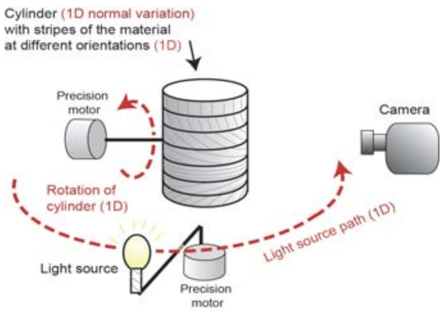


BRDF

- Bidirectional Reflectance Distribution Function
  - $\rho(\theta_i, \phi_i ; \theta_o, \phi_o)$
- Isotropic material
  - Invariant when material is rotated
  - BRDF is 3D
  - $\rho(\theta_i, \theta_o, \phi_{diff})$
- We can lookup the ρ value for a point (e.g., a vertex) if we know:
  - the light's position
  - the camera's position
  - position and normal at the point



Capturing an anisotropic BRDF

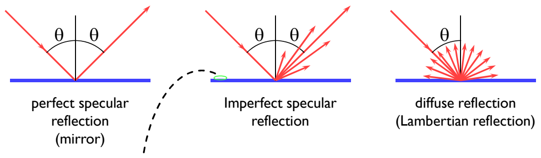


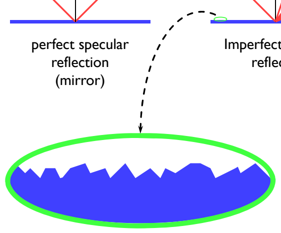
72

Equations for lighting

- Rather than using a BRDF look-up table, we might prefer a simple equation
  - This is the sort of trade-off that has occurred often in the history of computing
  - Early years: memory is expensive, so use a calculated *approximation* to the truth
  - More recently: memory is cheap, so use a large look-up table captured from the real world to give an accurate answer
  - Examples include: surface properties in graphics, sounds for electric pianos/organs, definitions of 3D shape

How do surfaces reflect light?



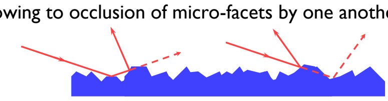


the surface of a specular reflector is facetted, each facet reflects perfectly but in a slightly different direction to the other facets

Johann Lambert, 18th century German mathematician

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

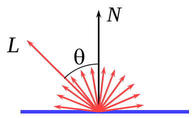


- plastics are good examples of surfaces with:
  - specular reflection in the light's colour
  - diffuse reflection in the plastic's colour

Calculating the shading of a surface

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

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 surface

I_l is the intensity of the light source

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

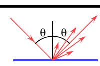
I is the intensity of the light reflected by the surface

use this equation to calculate the colour of a pixel

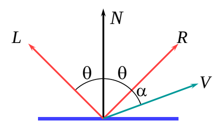
Diffuse shading: comments

- 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 surfaces?
  - 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
- this is essentially a simple one-parameter (θ) BRDF

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 surface

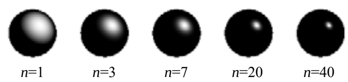
V is a normalised vector pointing at the viewer

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

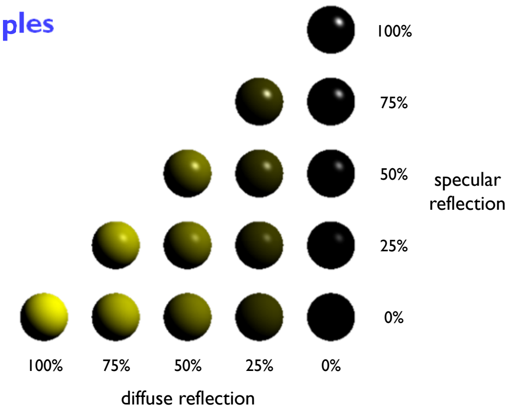
I is the intensity of the specularly reflected light



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

79

Examples



80

The gross assumptions revisited

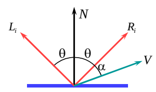
- only diffuse reflection
  - now have a method of approximating specular reflection
- no shadows
  - need to do ray tracing or shadow mapping to get shadows
- lights at infinity
  - can add local lights at the expense of more calculation
    - need to interpolate the L vector
- no interaction between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

81

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

82

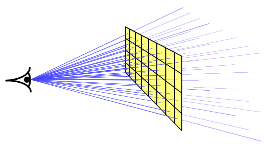
Illumination & shading: comments

- 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 a gross approximation
  - distributed ray tracing can handle specular inter-reflection
  - radiosity can handle diffuse inter-reflection

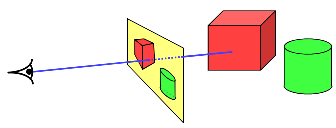
83

Ray tracing

- Identify point on surface and calculate illumination
- Given a set of 3D objects, shoot a ray from the eye through the centre of every pixel and see what surfaces it hits



shoot a ray through each pixel



whatever the ray hits determines the colour of that pixel

84

Ray tracing: examples

ray tracing easily handles reflection, refraction, shadows and blur

ray tracing is computationally expensive

85

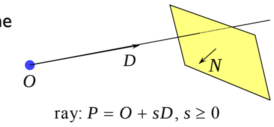
Ray tracing algorithm

```
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 ;
```

86

Intersection of a ray with an object 1

- plane



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

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

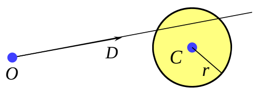
$$s = -\frac{d + N \cdot O}{N \cdot D}$$

- polygon or disc
  - 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 (which is simple for a disc)

87

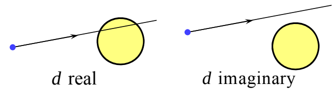
Intersection of a ray with an object 2

- sphere



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

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

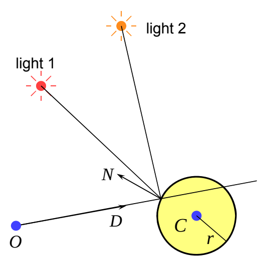


$$a = D \cdot D$$
$$b = 2D \cdot (O - C)$$
$$c = (O - C)\cdot(O - C) - r^2$$
$$d = \sqrt{b^2 - 4ac}$$
$$s_1 = \frac{-b + d}{2a}$$
$$s_2 = \frac{-b - d}{2a}$$

- cylinder, cone, torus
  - all similar to sphere
  - try them as an exercise

88

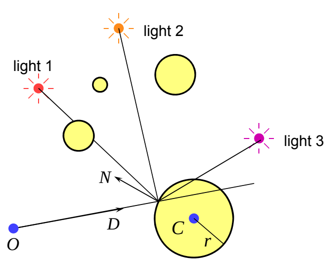
Ray tracing: shading



- 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)

89

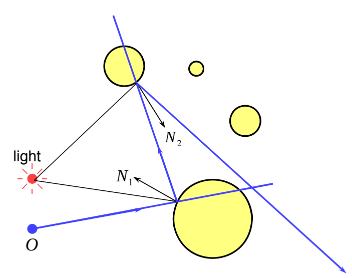
Ray tracing: shadows



- 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

90

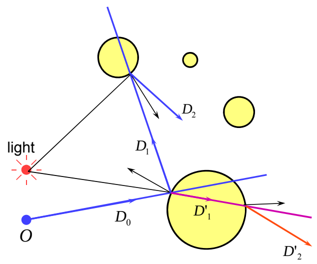
Ray tracing: reflection



- 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

91

Ray tracing: transparency & refraction

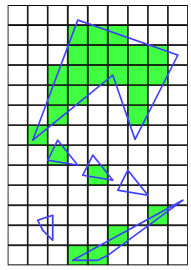


- objects can be totally or partially transparent
  - this allows objects behind the current one to be seen through it
- transparent objects can have refractive indices
  - bending the rays as they pass through the objects
- transparency + reflection means that a ray can split into two parts

92

Sampling

- we have assumed so far that each ray passes through the centre of a pixel
  - i.e. the value for each pixel is the colour of the object which happens to lie exactly under the centre of the pixel
- this leads to:
  - stair step (jagged) edges to objects
  - small objects being missed completely
  - thin objects being missed completely or split into small pieces



93

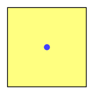
Anti-aliasing

- 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

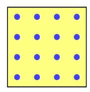
94

Sampling in ray tracing

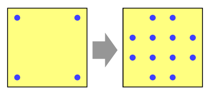
- single point
  - shoot a single ray through the pixel's centre



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



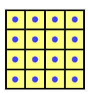
- adaptive super-sampling
  - shoot a few rays through the pixel, check the variance of the resulting values, if similar enough stop, otherwise shoot some more rays

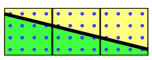


95

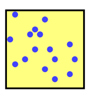
Types of super-sampling 1

- 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





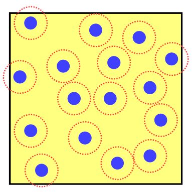
- random
  - shoot N rays at random points in the pixel
  - replaces aliasing artefacts with noise artefacts
    - the eye is far less sensitive to noise than to aliasing

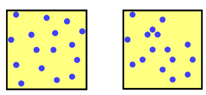


96

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



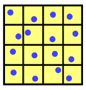


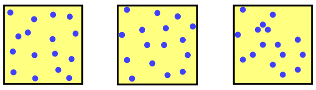
Poisson disc pure random

97

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





jittered — Poisson disc — pure random

98

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
  - ➊ 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
  - ➋ each ray spawns multiple rays when it hits an object
    - this alternative can be used, for example, for area lights

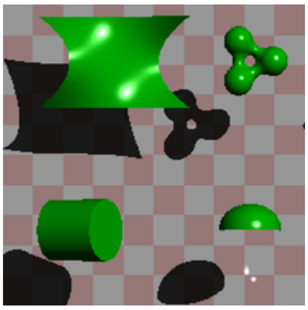
99

Examples of distributed ray tracing

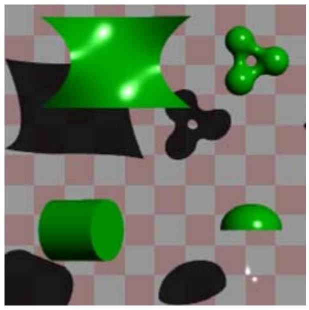
- distribute the samples for a pixel over the pixel area
  - get random (or jittered) super-sampling
  - used for anti-aliasing
- distribute the rays going to a light source over some area
  - allows area light sources in addition to point and directional light sources
  - produces soft shadows with 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

100

Anti-aliasing



one sample per pixel



multiple samples per pixel

101

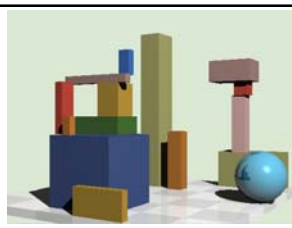
Area *vs* point light source

an area light source produces soft shadows

a point light source produces hard shadows

102

Finite aperture



left, a pinhole camera

below, a finite aperture camera

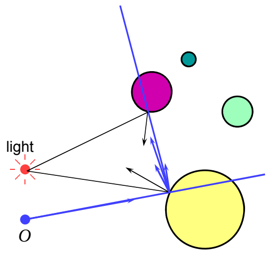
below left, 12 samples per pixel

below right, 120 samples per pixel

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

103

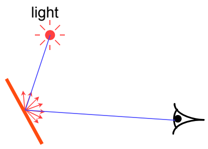
Distributed ray tracing for specular reflection



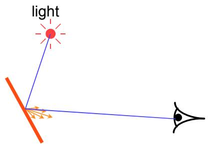
- previously we could only calculate the effect of perfect reflection
- we can now distribute the reflected rays over the range of directions from which specularly reflected light could come
- provides a method of handling some of the inter-reflections between objects in the scene
- requires a very large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

104

Handling direct illumination



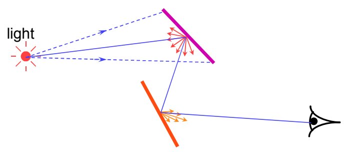
- diffuse reflection
  - handled by ray tracing and polygon scan conversion
  - assumes that the object is a perfect Lambertian reflector



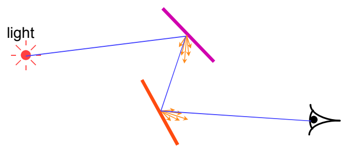
- specular reflection
  - also handled by ray tracing and polygon scan conversion
  - use Phong's approximation to true specular reflection

105

Handing indirect illumination: 1



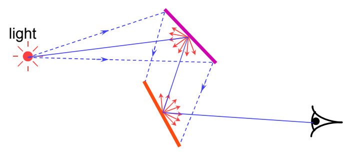
- diffuse to specular
  - handled by distributed ray tracing



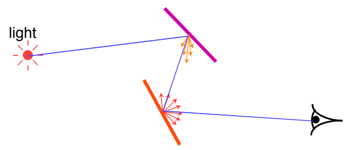
- specular to specular
  - also handled by distributed ray tracing

106

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

107

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)*

108

109

Computer Graphics & Image Processing

- Background
- Simple rendering
- Graphics pipeline
  - Polyhedral models
  - Perspective, shading and texture
  - OpenGL
- Underlying algorithms
- Colour and displays
- Image processing

110

Unfortunately…

- Ray tracing is computationally expensive
  - used by hobbyists and for super-high visual quality
- Video games and user interfaces need something faster
- So:
  - Model surfaces as polyhedra – meshes of polygons
  - Use composition to build scenes
  - Apply perspective transformation and project into plane of screen
  - Work out which surface was closest
  - Fill pixels with colour of nearest visible polygon
- Modern graphics cards have hardware to support this

111

Three-dimensional objects

- Polyhedral surfaces are made up from meshes of multiple connected polygons

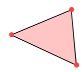
- Polygonal meshes
  - open or closed
  - manifold or non-manifold

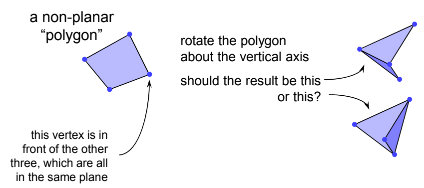
- Curved surfaces
  - must be converted to polygons to be drawn

112

Surfaces in 3D: polygons

- Easier to consider planar polygons
  - 3 vertices (triangle) must be planar
  - > 3 vertices, not necessarily planar

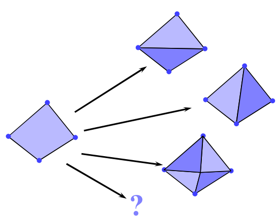




113

Splitting polygons into triangles

- Most Graphics Processing Units (GPUs) are optimised to draw triangles
- Split polygons with more than three vertices into triangles

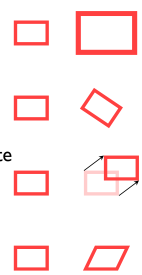


which is preferable?

114

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

115

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***

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

116

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***

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

117

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:

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

118

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*

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

119

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

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

120

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:

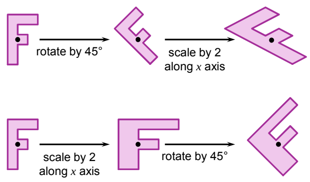
$$\begin{bmatrix} x'' \\ y'' \\ w'' \end{bmatrix} = \overset{\text{scale}}{\begin{bmatrix} m & 0 & 0 \\ 0 & m & 0 \\ 0 & 0 & 1 \end{bmatrix}} \begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} \qquad \begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \overset{\text{shear}}{\begin{bmatrix} 1 & a & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}} \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

$$\begin{bmatrix} x'' \\ y'' \\ w'' \end{bmatrix} = \overset{\text{scale}}{\begin{bmatrix} m & 0 & 0 \\ 0 & m & 0 \\ 0 & 0 & 1 \end{bmatrix}} \overset{\text{shear}}{\begin{bmatrix} 1 & a & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}} \begin{bmatrix} x \\ y \\ w \end{bmatrix} = \overset{\text{both}}{\begin{bmatrix} m & ma & 0 \\ 0 & m & 0 \\ 0 & 0 & 1 \end{bmatrix}} \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

121

Transformation are 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

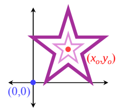
$$\begin{bmatrix} 1/\sqrt{2} & -1/\sqrt{2} & 0 \\ 1/\sqrt{2} & 1/\sqrt{2} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

rotate

122

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)



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

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

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

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

Exercise: show how to perform rotation about an arbitrary point

123

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

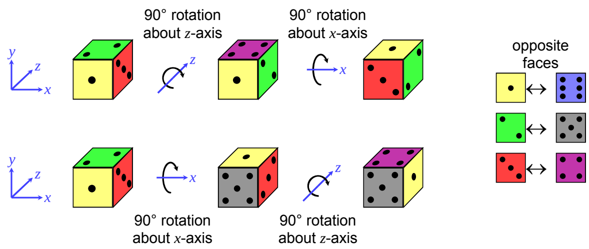
$$\begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

rotation about y-axis

$$\begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

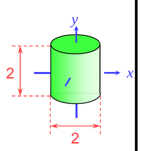
124

3D transformations are not commutative



125

Model transformation 1

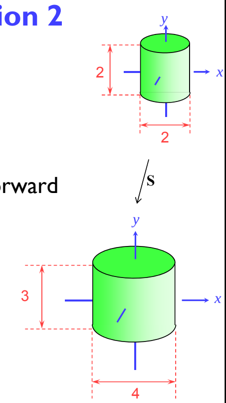


- the graphics package Open Inventor defines a cylinder to be:
  - **centre at the origin, (0,0,0)**
  - **radius 1 unit**
  - **height 2 units, aligned along the y-axis**
- this is the only cylinder that can be drawn, *but* the package has a complete set of 3D transformations
- we want to draw a cylinder 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?

126

Model transformation 2

- order is important:
  - scale first
  - rotate
  - translate last
- scaling and translation are straightforward



$$\mathbf{S} = \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 1.5 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

scale from size (2,2,2) to size (4,3,4)

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & 1.5 \\ 0 & 1 & 0 & 3 \\ 0 & 0 & 1 & 4 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

translate centre of cylinder from (0,0,0) to halfway between (1,2,3) and (2,4,5)

127

Model transformation 3

- rotation is a multi-step process
  - break the rotation into steps, each of which is rotation about a principal axis
  - work these out by taking the desired orientation back to the original axis-aligned position
    - **the centres of its two ends located at (1,2,3) and (2,4,5)**
  - desired axis: (2,4,5)–(1,2,3) = (1,2,2)
  - original axis: y-axis = (0,1,0)

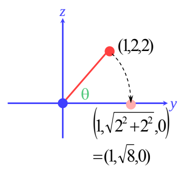
128

Model transformation 4

- desired axis: (2,4,5)–(1,2,3) = (1,2,2)
- original axis: y-axis = (0,1,0)
- zero the z-coordinate by rotating about the x-axis

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

$$\theta = -\arcsin\frac{2}{\sqrt{2^2+2^2}}$$



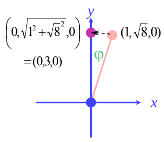
129

Model transformation 5

- then zero the x-coordinate by rotating about the z-axis
- we now have the object's axis pointing along the y-axis

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

$$\varphi = \arcsin\frac{1}{\sqrt{1^2+\sqrt{8}^2}}$$



130

Model transformation 6

- the overall transformation is:
  - first scale
  - then take the inverse of the rotation we just calculated
  - finally translate to the correct position

$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \mathbf{T} \times \mathbf{R}_1^{-1} \times \mathbf{R}_2^{-1} \times \mathbf{S} \times \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

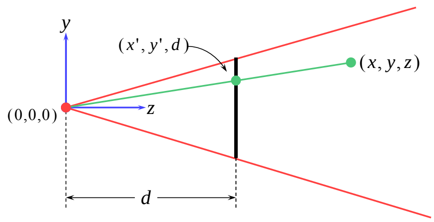
131

Application: display multiple instances

- transformations allow you to define an object at one location and then place multiple instances in your scene

132

Geometry of perspective projection



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

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

133

Projection as a matrix operation

$$\begin{bmatrix} x \\ y \\ 1/d \\ z/d \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1/d \\ 0 & 0 & 1/d & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

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

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

remember $\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} \rightarrow \begin{bmatrix} x/w \\ y/w \\ z/w \end{bmatrix}$

This is useful in the z-buffer algorithm where we need to interpolate $1/z$ values rather than z values.

$$z' = \frac{1}{z}$$

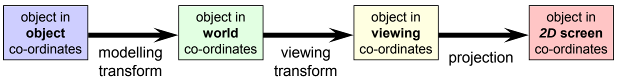
134

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

135

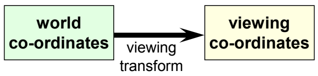
A variety of transformations



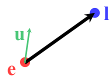
- the modelling transform and viewing transform can be multiplied together to produce a single matrix taking an object directly from object co-ordinates into viewing co-ordinates
- either or both of the modelling transform and viewing transform matrices can be the identity matrix
  - e.g. objects can 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

136

Viewing transformation 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 $\overline{\mathbf{el}}$



137

Viewing transformation 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

$$|\overline{\mathbf{el}}| = \sqrt{(l_x - e_x)^2 + (l_y - e_y)^2 + (l_z - e_z)^2}$$

$$\mathbf{S} = \begin{bmatrix} d/|\overline{\mathbf{el}}| & 0 & 0 & 0 \\ 0 & d/|\overline{\mathbf{el}}| & 0 & 0 \\ 0 & 0 & d/|\overline{\mathbf{el}}| & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

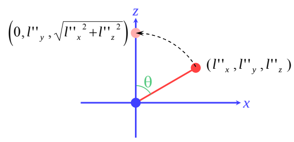
138

Viewing transformation 3

- need to align line $\overline{\mathbf{el}}$ with z-axis
  - first transform e and l into new co-ordinate system
    $\mathbf{e}'' = \mathbf{S} \times \mathbf{T} \times \mathbf{e} = \mathbf{0}$ $\mathbf{l}'' = \mathbf{S} \times \mathbf{T} \times \mathbf{l}$
  - then rotate e''l'' into yz-plane, rotating about y-axis

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

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



139

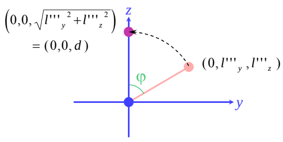
Viewing transformation 4

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

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

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

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



140

Viewing transformation 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

141

Viewing transformation 6

world co-ordinates $\xrightarrow{\text{viewing transform}}$ viewing co-ordinates

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

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

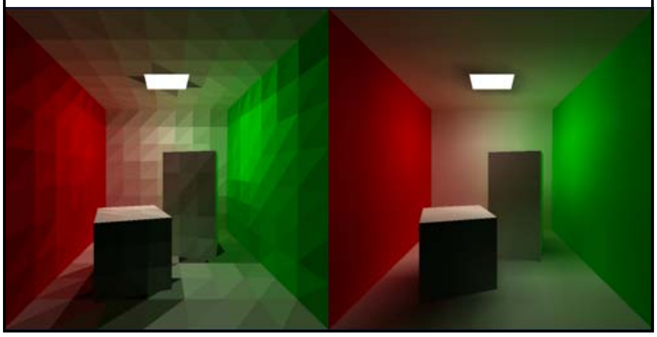
- in particular: $\mathbf{e} \to (0,0,0) \quad \mathbf{l} \to (0,0,d)$
- the matrices depend only on **e**, **l**, and **u**, so they can be pre-multiplied together

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

142

Illumination & shading

- Drawing polygons with uniform colours gives poor results
- Interpolate colours across polygons



143

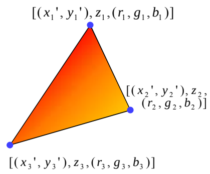
Illumination & shading

- Interpolating colours across polygons needs
  - colour at each vertex
  - algorithm to blend between the colours across the polygon
- Works for ambient lighting and diffuse reflection
- Specular reflection requires more information than just the colour

145

Gouraud shading

- for a polygonal model, calculate the diffuse illumination at each ***vertex***
  - 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 between the vertices across the polygon
- surface will look smoothly curved
  - rather than looking like a set of polygons
  - surface outline will still look polygonal

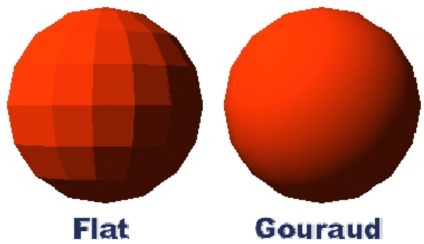


Henri Gouraud, "Continuous Shading of Curved Surfaces", *IEEE Trans Computers*, **20**(6), 1971

146

Flat *vs* Gouraud shading

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

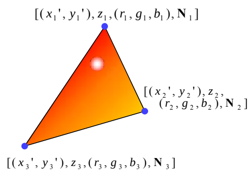


http://computer.howstuffworks.com/question484.htm

147

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



148

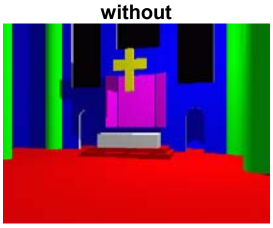
Surface detail

- 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

149

Texture mapping

without



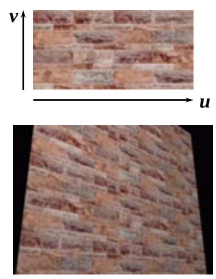
all surfaces are smooth and of uniform colour

with

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

150

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

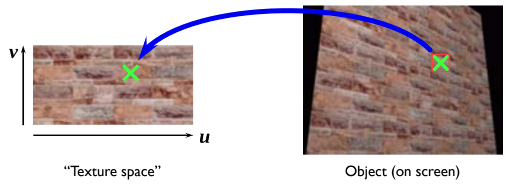
151

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

152

Sampling texture space

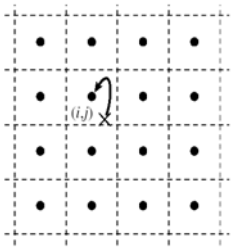


Find (u,v) coordinate of the sample point on the object and map this into texture space

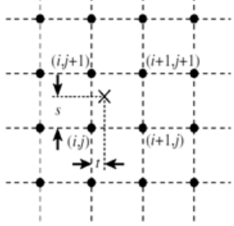
Sample texture space to determine the pixel's colour

153

Sampling texture space: finding the value



Nearest neighbour: the sample value is the nearest pixel value to the sample point.

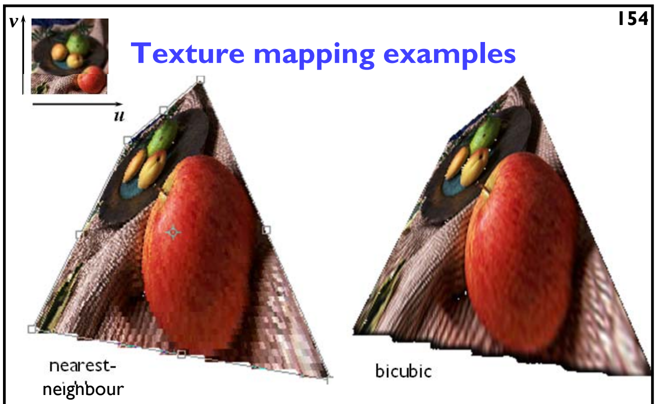


Bi-linear: the sample value is the weighted mean of the four pixels around the sample point.

Bi-cubic (not shown): the sample value is the weighted mean of the sixteen pixels around the sample point. Runs at a quarter the speed of bi-linear.

154

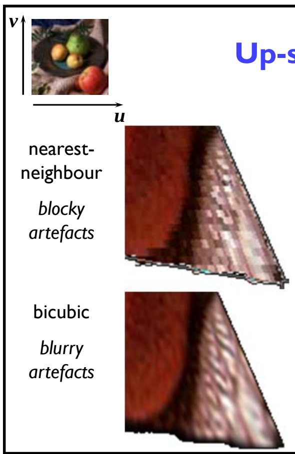
Texture mapping examples



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

155

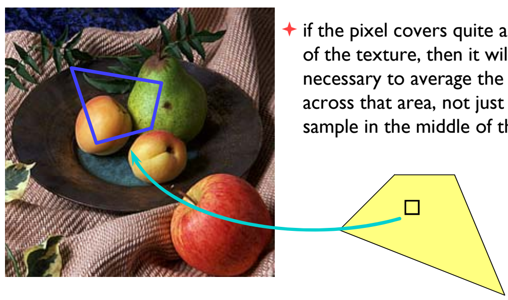
Up-sampling



- if one pixel in the texture map covers several pixels in the final image, you get visible artefacts
- only practical way to prevent this is to ensure that texture map is of sufficiently high resolution that it does not happen

156

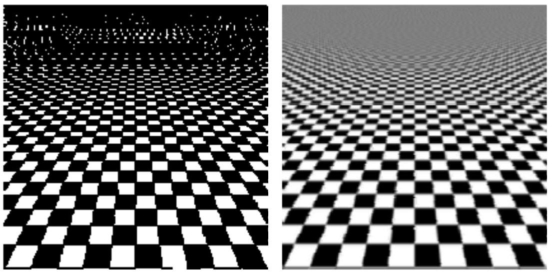
Down-sampling



- 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

Down-sampling

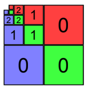
without area averaging | with area averaging



Multi-resolution texture

- rather than down-sampling when necessary, pre-calculate multiple versions of the texture at different resolutions and pick the appropriate resolution to sample from…
- 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

The MIP map



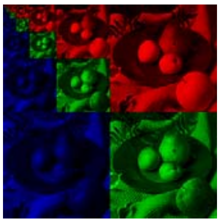
- an efficient memory arrangement for a multi-resolution colour image
- pixel (x,y) is a bottom level pixel location (level 0); for an image of size (m,n), it is stored at these locations in level k:

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



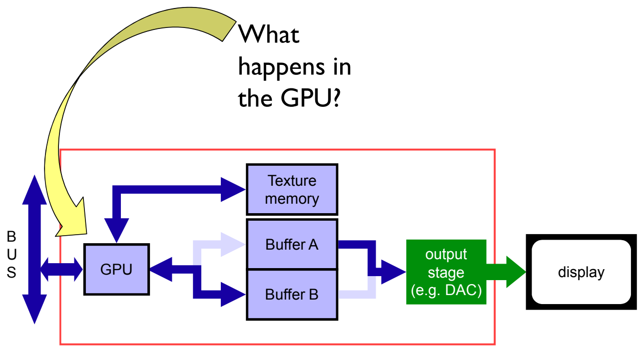
What can a texture map modify?

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

Bump mapping

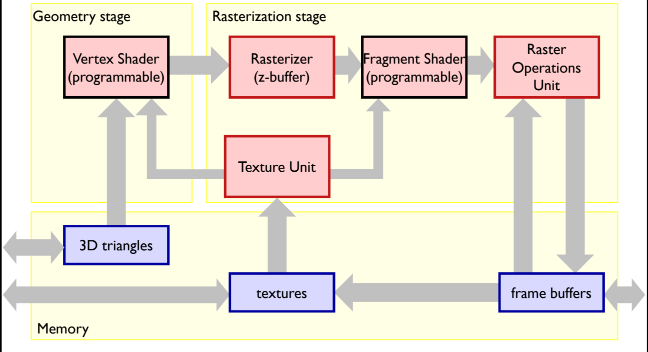
- the surface normal is used in calculating both diffuse and specular reflection
- bump mapping modifies the direction of the surface normal so that the surface appears more or less bumpy
- rather than using a texture map, a 2D function can be used which varies the surface normal smoothly across the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

Graphics card architecture



163

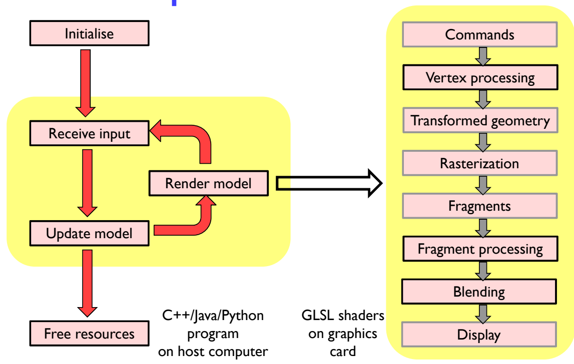
Graphics card architecture



based on nVIDIA's GeForce 6 architecture

164

OpenGL architecture



165

OpenGL in Java

```
public class MyCanvas extends GLCanvas implements GLEventListener {
        public void init (GLAutoDrawable drawable) {
        // Build model in buffers
        // Compile and link shaders
        }
        public void display (GLAutoDrawable drawable) {
        // Update transformations
        // Draw
        }
        public void dispose (GLAutoDrawable drawable) { … }
        public void reshape (GLAutoDrawable drawable,
                int x, int y, int width, int height) { … }
}
```

166

OpenGL shaders

- Compute
- Vertex
  - Receives coordinates, colour and transformations
  - Applies model and view transformations to vertices
  - Outputs transformed coordinates and colour
- Tessellation control and evaluation
- Geometry
- Fragment
  - Receives interpolated values from vertex shader
  - Calculates lighting and shading for each visible pixel
  - Outputs fragment colour

167

OpenGL Shading Language

- Vertex shader
  - *uniform* inputs per object – e.g. transformations
  - *in* inputs per vertex – e.g. position and colour
  - applies transformations to vertices
  - *out* outputs per vertex – will be interpolated across a face
- Fragment shader
  - *in* inputs interpolated between vertices
  - calculates lighting and shading
  - outputs gl_FragColor for pixel

168

Computer Graphics & Image Processing

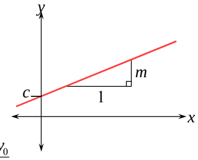
- Background
- Simple rendering
- Graphics pipeline
- Underlying algorithms
  - Drawing lines, curves and polygons in 2D
  - Clipping
  - 3D scan conversion
- Colour and displays
- Image processing

Drawing a straight line

- a straight line can be defined by:

$$y = mx + c$$

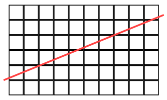
the slope of the line



For a line passing through (x_0, y_0) and (x_1, y_1): $m = \frac{y_1 - y_0}{x_1 - x_0}$

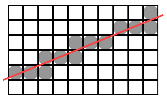
$c = y_0 - m x_0$

- 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?



Which pixels do we use?

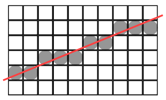
- there are two reasonably sensible alternatives:



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

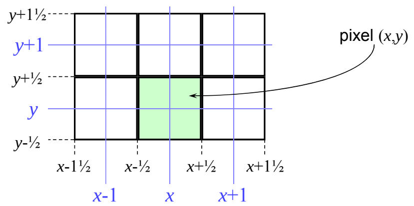
for lines of slope less than 45° we always have just one pixel in every column

✓

- **in general, use this**

A line drawing algorithm — preparation 1

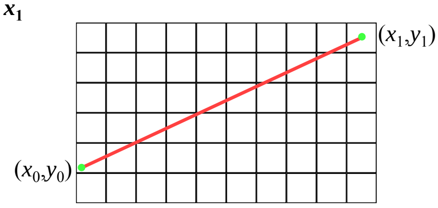
- pixel (x,y) has its centre at real co-ordinate (x,y)
  - it thus stretches from $(x-½, y-½)$ to $(x+½, y+½)$



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

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 $(0 \le m \le 1)$
- $x_0 < x_1$



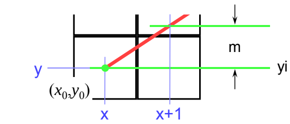
Bresenham's line drawing algorithm for integer end points

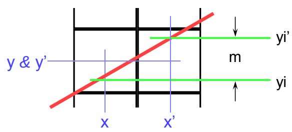
Initialisation

m = $(y_1 - y_0) / (x_1 - x_0)$
x = x_0
yi = y_0
y = y_0
DRAW(x,y)

Iteration

WHILE x < x_1 DO
  x = x + 1
  yi = yi + m
  y = ROUND(yi)
  DRAW(x,y)
END WHILE



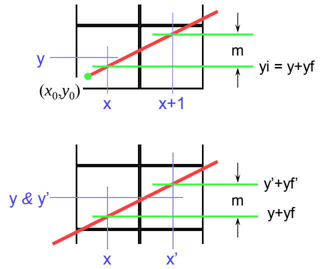


J. E. Bresenham, "Algorithm for Computer Control of a Digital Plotter", *IBM Systems Journal*, **4**(1), 1965

175

Bresenham's algorithm for floating point end points

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

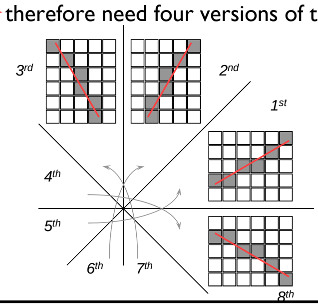


We need to calculate the initial y from the rounded off initial position of x_0 because we will not necessarily get the right answer by rounding x_0 and y_0 independently.
Splitting the y-coordinate into fractional (yf) and integer (y) parts avoids rounding on every cycle.

176

Bresenham's algorithm — more details

- 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

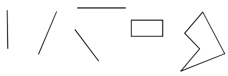


Exercise: work out what changes need to be made to the algorithm for it to work in each of the other three octants

177

Uses of the line drawing algorithm

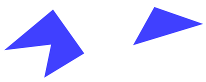
- to draw lines



- 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



178

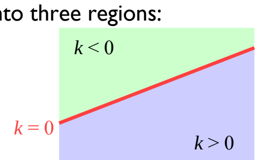
A second line drawing algorithm

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

$$k(x,y) = ax + by + c$$

For a line segment from (x_0,y_0) to (x_1,y_1), the line is defined by: $a = y_1 - y_0$
$b = -(x_1 - x_0)$
$c = x_1y_0 - x_0y_1$

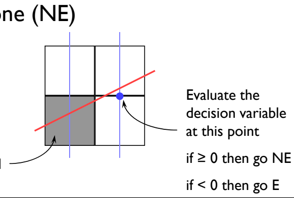
- this divides the plane into three regions:
  - above the line $k < 0$
  - below the line $k > 0$
  - on the line $k = 0$



179

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



180

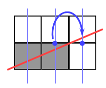
Midpoint line drawing algorithm 2

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

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

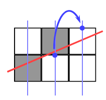
- if go E then the new decision variable is at $(x+2, y+½)$

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



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

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

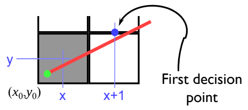


181

Midpoint line drawing algorithm 3

Initialisation

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



Iteration

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

E case
just increment x

NE case
increment x & y

182

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

183

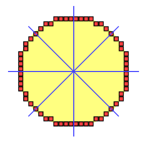
Curves

- circles & ellipses
- Bézier 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
- NURBS
  - Non-Uniform Rational B-Splines
  - more powerful than Bezier & now more widely used
  - consider these in Part II

184

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



185

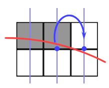
Midpoint circle algorithm 2

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

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

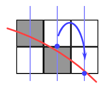
- if go E then the new decision variable is at $(x+2, y-½)$

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



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

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

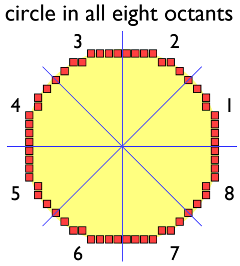


186

Midpoint circle algorithm 3

- Drawing an origin-centred circle in all eight octants

Call	Octant
Draw(x,y)	2
Draw(-x,y)	3
Draw(-x,-y)	6
Draw(x,-y)	7
Draw(y,x)	1
Draw(-y,x)	4
Draw(-y,-x)	5
Draw(y,-x)	8

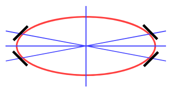


The second-octant algorithm thus allows you to draw the whole circle.

187

Taking circles further

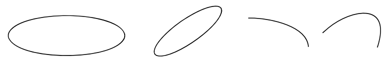
- the algorithm can be easily extended to circles not centred at the origin
- a similar method can be derived for ovals
  - but: cannot naively use octants
    - use points of 45° slope to divide oval into eight sections
  - and: ovals must be axis-aligned
    - there is a more complex algorithm which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



188

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

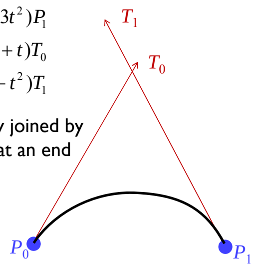
189

Hermite cubic

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

$$P(t) = (2t^3 - 3t^2 + 1)P_0 + (-2t^3 + 3t^2)P_1 + (t^3 - 2t^2 + t)T_0 + (t^3 - t^2)T_1$$

- 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

190

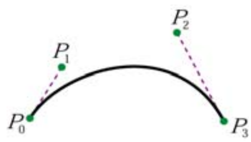
Bézier cubic

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

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

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

$0 \le t \le 1$



Pierre Bézier worked for Renault in the 1960s

191

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
  - because weights sum to 1 and all weights are non-negative

192

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

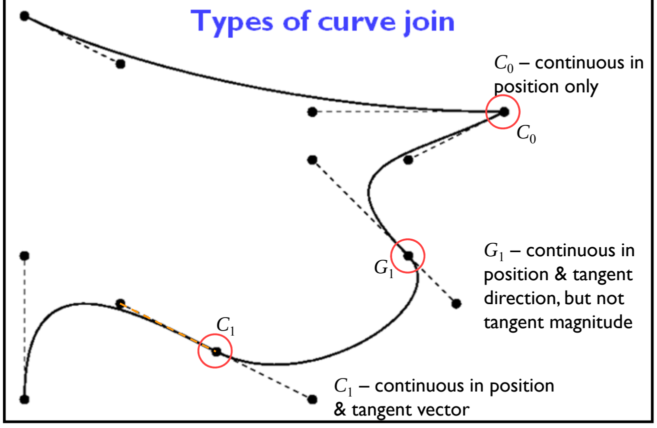
C_n (mathematical continuity): continuous in all derivatives up to the n^{th} derivative

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

$C_n \Rightarrow G_n$

193

Types of curve join



194

Drawing a Bezier cubic – iterative method

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

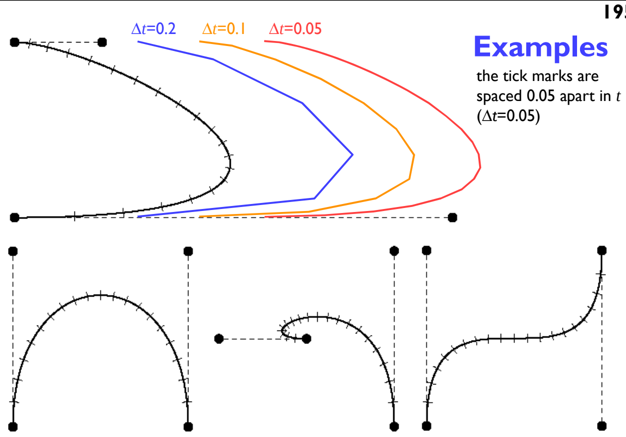
```
(x0,y0) = Bezier(0)
FOR t = 0.05 TO 1 STEP 0.05 DO
        (x1,y1) = Bezier(t)
        DrawLine( (x0,y0), (x1,y1) )
        (x0,y0) = (x1,y1)
END FOR
```

- 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

195

Examples

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



196

Drawing a Bezier cubic – adaptive 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

197

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

198

Checking for flatness

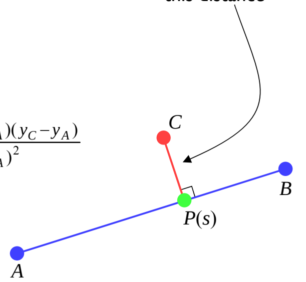
$$P(s) = (1-s)A + sB$$

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

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

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

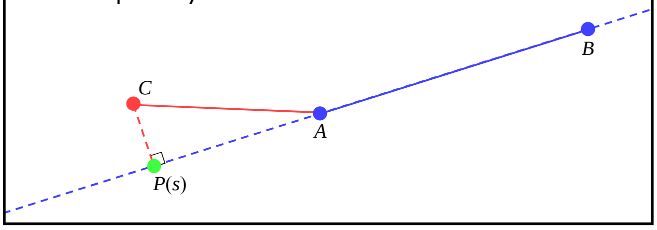
we need to know this distance



199

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



200

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

201

The effect of different tolerances

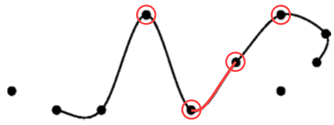
- this is the same Bezier curve drawn with four different tolerances

100 20 5 0.2

202

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

203

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:

      $P_0=B \qquad P_1=B+(C-A)/6$

      $P_3=C \qquad P_2=C-(D-B)/6$
- 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

204

Simplifying line chains

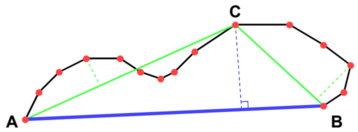
- this can be thought of as an inverse problem to the one of drawing Bezier curves
- problem specification: you are given a chain of line segments at a very high resolution, how can you reduce the number of line segments without compromising quality
  - e.g. given the coastline of Britain defined as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

205

Douglas & Peucker'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

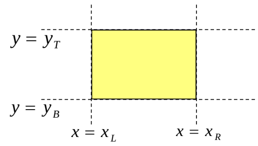


Douglas & Peucker, *Canadian Cartographer*, **10**(2), 1973

206

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:

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

207

Clipping lines against a rectangle — naïvely

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

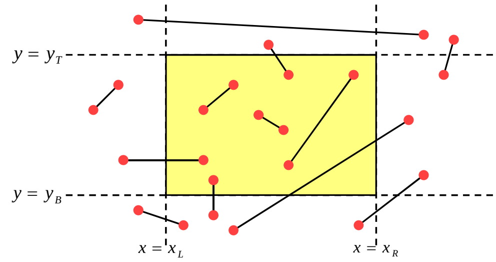
- do this operation for each of the four edges

  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

This is naïve because a lot of unnecessary operations will be done for most lines.

208

Clipping lines against a rectangle — examples



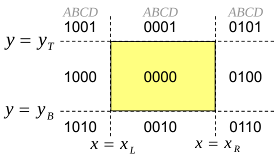
- you can naïvely check every line against each of the four edges
  - this works but is obviously inefficient
- adding a little cleverness improves efficiency enormously
  - Cohen-Sutherland clipping algorithm

209

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

  $Q_1 = A_1 B_1 C_1 D_1 \quad Q_2 = A_2 B_2 C_2 D_2$

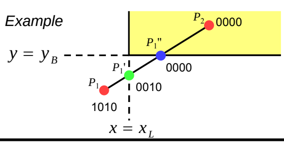
Ivan Sutherland is one of the founders of Evans & Sutherland, manufacturers of flight simulator systems

210

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

Example



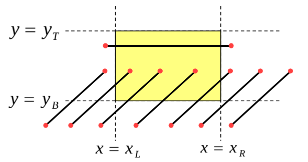
$$x_1' = x_L \qquad y_1' = y_1 + (y_2 - y_1)\frac{x_L - x_1}{x_2 - x_1}$$

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

211

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?



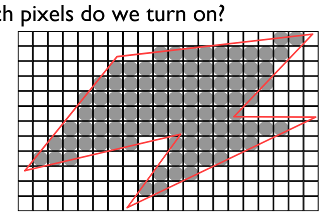
Exercise: what happens in each of the cases at left?
[Assume that, where there is a choice, the algorithm always tries to intersect with x_L or x_R before y_B or y_T.]

Try some other cases of your own devising.

212

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

213

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)*

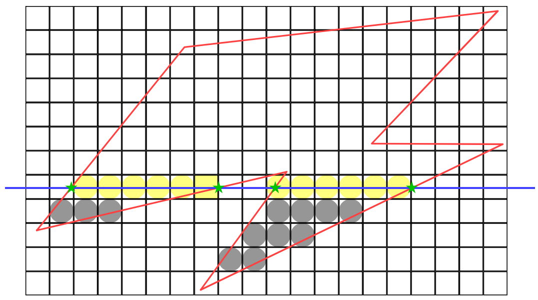
❸ for each edge in the AEL: find the intersection point with the current scanline; sort these into ascending order on the *x* value

❹ fill between pairs of intersection points

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

214

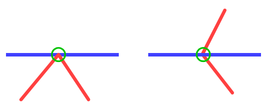
Scanline polygon fill example



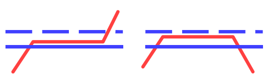
215

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

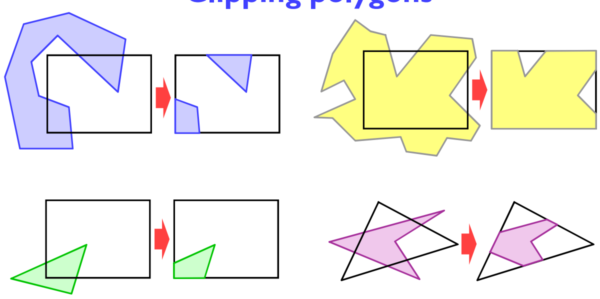


- what about horizontal edges?
  - can throw them out of the *edge list*, they contribute nothing



216

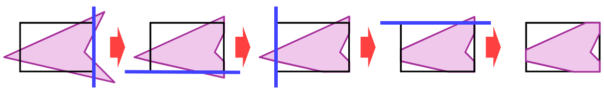
Clipping polygons



217

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

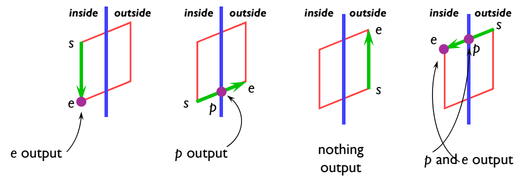


Sutherland & Hodgman, "Reentrant Polygon Clipping," *Comm. ACM*, **17**(1), 1974

218

Sutherland-Hodgman polygon clipping 2

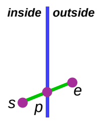
- the algorithm progresses around the polygon checking if each edge crosses the clipping line and outputting the appropriate points



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

219

Sutherland-Hodgman polygon clipping 3



- line segment defined by (x_s,y_s) and (x_e,y_e)
- line segment is: $p(t) = (1\text{-}t)s+te$
- clipping edge defined by $ax+by+c=0$
- test to see which side of edge s and e are on:
  - $k=ax+by+c$
  - k negative: inside, k positive: outside, $k=0$: on edge
- if k_s and k_e differ in sign then intersection point can be found by:

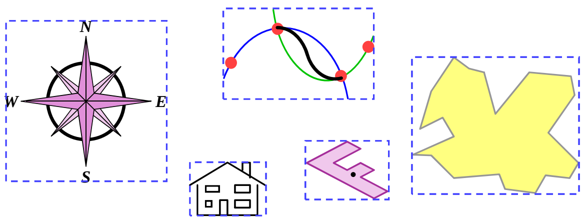
$$a((1-t)x_s + tx_e) + b((1-t)y_s + ty_e) + c = 0$$

$$\Rightarrow t = \frac{ax_s + by_s + c}{a(x_s - x_e) + b(y_s - y_e)}$$

220

Bounding boxes

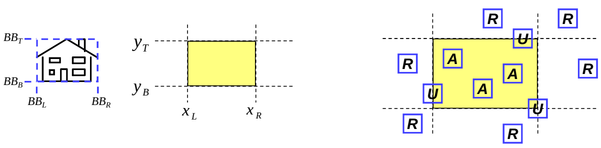
- when working with complex objects, bounding boxes can be used to speed up some operations



221

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

222

Clipping Bézier curves

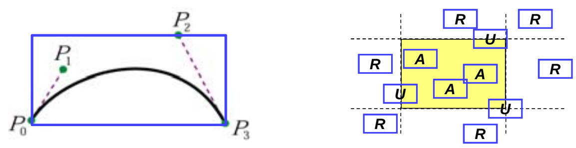
If flat ⇒ draw using clipped line drawing algorithm

Else consider the Bézier's bounding box

accept ⇒ draw using normal (unclipped) Bézier algorithm

reject ⇒ do not draw at all

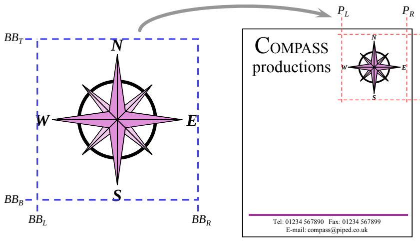
unsure ⇒ split into two Béziers, recurse



223

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

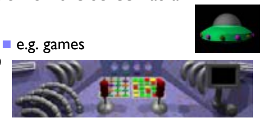


use the eight values to translate and scale the original to the appropriate position in the destination document

224

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

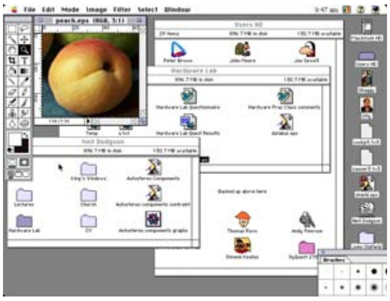


- copying an image from place to place is essentially a memory operation
  - can be made very fast
  - e.g. 32×32 pixel icon can be copied, say, 8 adjacent pixels at a time, if there is an appropriate memory copy operation

225

Application 1: user interface

- early graphical user-interfaces needed to use objects that were quick to draw
  - straight lines
  - filled rectangles
- complicated bits were done using predrawn icons
- typefaces also tended to be predrawn



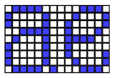
226

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*** – Arial
abcd *efgh* **ijkl** ***mnop*** – Times
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.

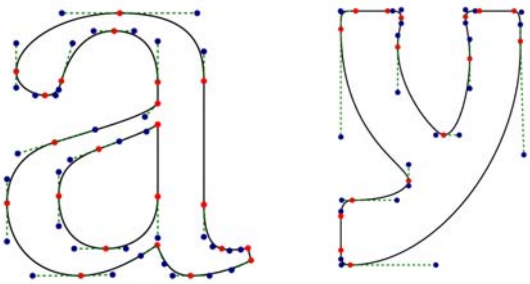
227

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

228

Examples which are Bezier-friendly

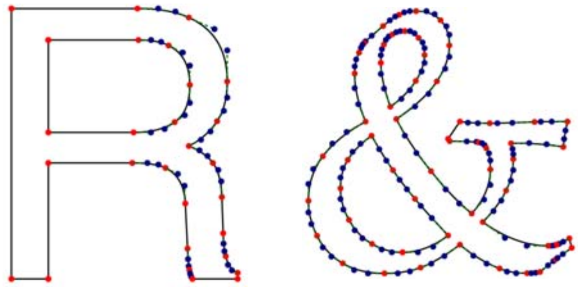


typeface: Utopia (1989)
designed as a Postscript typeface by Robert Slimbach at Adobe

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

229

Examples which are more fussy



typeface: Helvetica (1957)
abcdQRST2345&

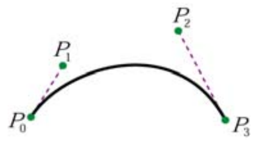
typeface: Palatino (1950)
abcdQRST2345&

230

Curves in 3D

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

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

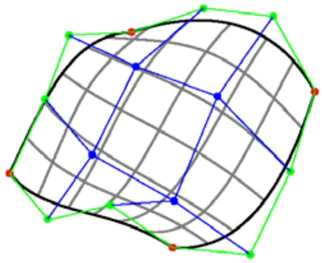


where: $P_i \equiv (x_i, y_i, z_i)$

231

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



232

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

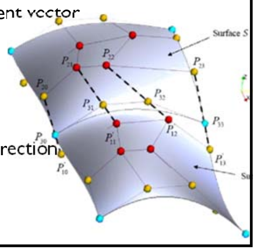
- compare this with the 2D version:

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

233

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 other and be equidistant from the edge point
  - G_1 – continuous in position and tangent direction
    - the four edge control points must match
    - the relevant control points must be co-linear
    - see picture



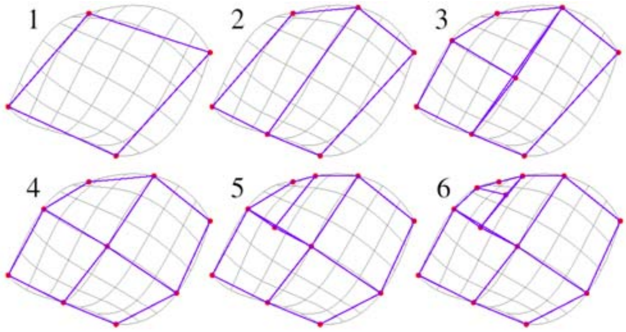
234

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 adaptive 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!

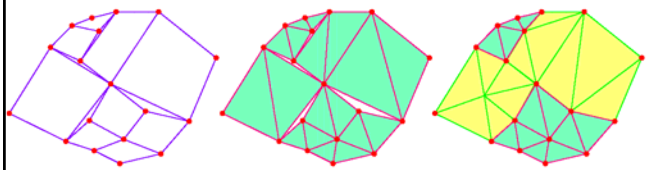
235

Subdividing a Bezier patch — example



236

Triangulating the subdivided patch



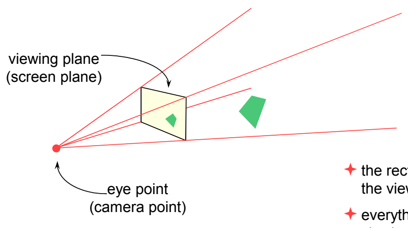
Final quadrilateral mesh | Naïve triangulation | More intelligent triangulation

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

- consider whether it is worth doing this adaptive method

237

Viewing volume

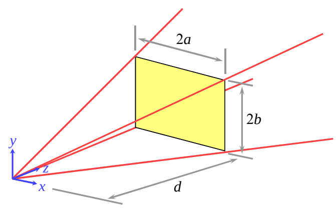


- the rectangular pyramid is the viewing volume
- everything within the viewing volume is projected onto the viewing plane

238

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:

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

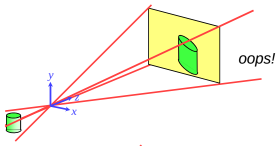
$$y > -z\frac{b}{d} \qquad y < z\frac{b}{d}$$

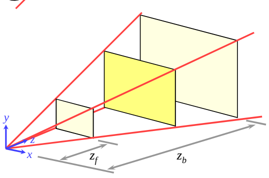
239

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*





240

Clipping in 3D — two methods

which is best?

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

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

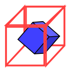
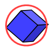
- project to 2D (retaining z) and clip against the axis-aligned cuboid
  - still need to clip against six planes

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

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

241

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

242

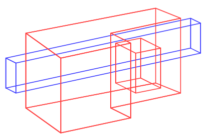
3D scan conversion

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

243

3D line drawing

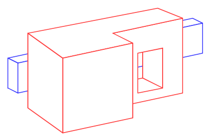
- given a list of 3D lines we draw them by:
  - projecting end points onto the 2D screen
  - using a line drawing algorithm on the resulting 2D lines
- this produces a wireframe version of whatever objects are represented by the lines



244

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



245

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

246

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

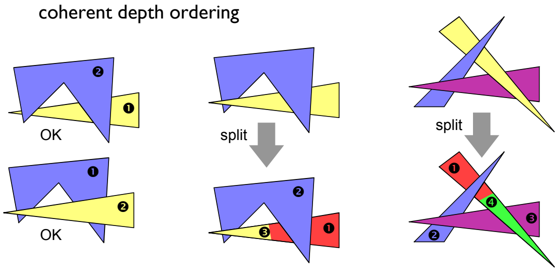
❹ draw the polygons in depth order from back to front

- "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

Resolving ambiguities in depth sort

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

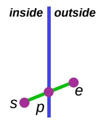


Resolving ambiguities: algorithm

- 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

Split a polygon by a plane



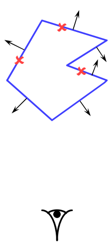
- remember the Sutherland-Hodgman algorithm
  - splits a 2D polygon against a 2D line
- do the same in 3D: split a (planar) polygon by a plane
- line segment defined by (x_s,y_s,z_s) and (x_e,y_e,z_e)
- clipping plane defined by $ax+by+cz+d=0$
- test to see which side of plane a point is on:
  - $k=ax+by+cz+d$
  - k negative: inside, k positive: outside, $k=0$: on edge
  - apply this test to all vertices of a polygon; if all have the same sign then the polygon is entirely on one side of the plane

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
- it is reasonably cheap for small number of polygons, but becomes expensive for large numbers of polygons
- the ordering is only valid from one particular viewpoint

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



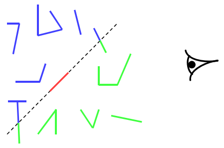
Binary Space-Partitioning trees

- 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

253

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



254

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

You need to be able to tell which side of an arbitrary plane a vertex lies on and how to split a polygon by an arbitrary plane – both of which were discussed for the depth-sort algorithm.

255

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

256

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

257

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
  - this is the algorithm used on graphics cards

258

z-buffer basics

- store both *colour* and *depth* at each pixel
- scan convert one polygon at a time in any order
- 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

259

z-buffer algorithm

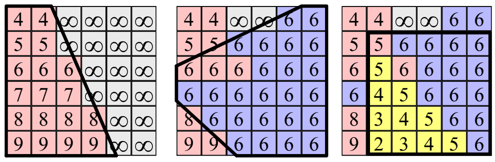
```
FOR every pixel (x,y)
    Colour[x,y] = background colour ;
    Depth[x,y] = infinity ;
END FOR ;

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 requires you to project the polygon's vertices to 2D and run the 2D polygon scan-conversion algorithm

this requires you to modify the 2D algorithm so that it can compute the *z*-value at each pixel

260

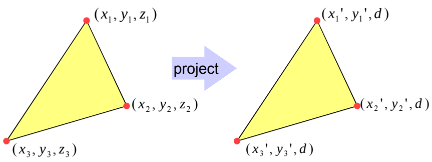
z-buffer example



261

Interpolating depth values 1

- just as we incrementally interpolate x as we move along each edge of the polygon, we can incrementally interpolate z:
  - as we move along the edge of the polygon
  - as we move across the polygon's projection



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

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

262

Interpolating depth values 2

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

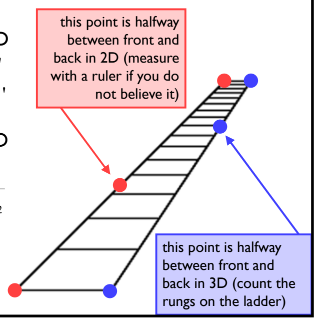
$$[(x_a', y_a'), z_a]$$

- 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

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



263

Interpolating depth values 3

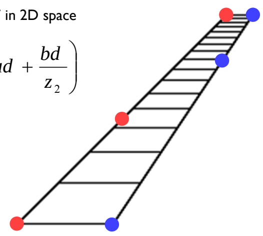
$$x = az + b$$ consider the projection onto the plane $y=0$

$$x' = x\frac{d}{z} = ad + \frac{bd}{z}$$ now project to $z=d$

$$x' = (1-t)x_1' + tx_2'$$ interpolate x' in 2D space

$$ad + \frac{bd}{z} = (1-t)\left(ad + \frac{bd}{z_1}\right) + t\left(ad + \frac{bd}{z_2}\right)$$

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



264

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

265

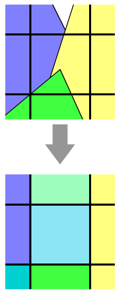
Putting it all together - a summary

- 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
  - either:
    - ordering the polygons so that they are drawn in the correct order
  - or:
    - calculating the *z* value at each pixel and using a depth-buffer

266

Anti-aliasing method 1: area averaging

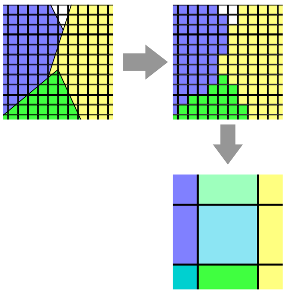
- average the contributions of all polygons to each pixel
  - e.g. assume pixels are square and we just want the average colour in the square
  - Ed Catmull developed an algorithm which does this:
    - works a scan-line at a time
    - clips all polygons to the scan-line
    - determines the fragment 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!



267

Anti-aliasing method 2: super-sampling

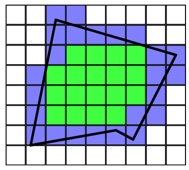
- sample on a finer grid, then average the samples in each pixel to produce the final colour
  - for an $n \times n$ sub-pixel grid, the algorithm would take roughly n^2 times as long as just taking one sample per pixel
- can simply average all of the sub-pixels in a pixel or can do some sort of weighted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



268

The A-buffer – efficient super-sampling

- 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



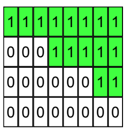
L. Carpenter, "The A-buffer: an antialiased hidden surface method", SIGGRAPH 84, 103–8

269

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

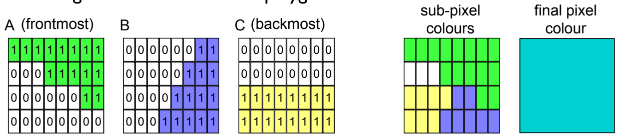
sampling is done at the centre of each of the sub-pixels

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.

270

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

271

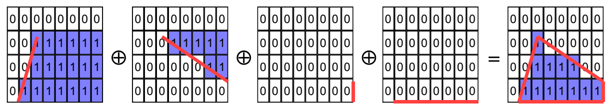
Making the A-buffer more efficient

- 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

272

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



273

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

274

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

275

276

277

Computer Graphics & Image Processing

- Background
- Simple rendering
- Graphics pipeline
- Underlying algorithms
- Colour and displays
  - Colour models for display and printing
  - Display technologies
  - Colour printing
- Image processing

278

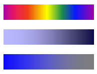
Representing colour

- 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

279

Munsell's colour classification system

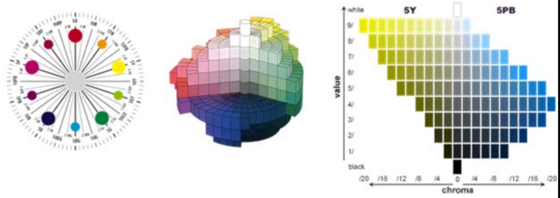
- three axes
  - hue ➤ the dominant colour
  - value ➤ bright colours/dark colours
  - chroma ➤ vivid colours/dull colours
- can represent this as a 3D graph



280

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



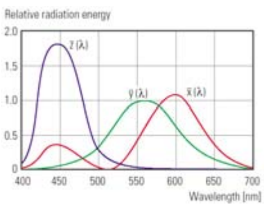
invented by Albert H. Munsell, an American artist, in 1905 in an attempt to systematically classify colours

281

XYZ colour space

FvDFH Sec 13.2.2

- 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)

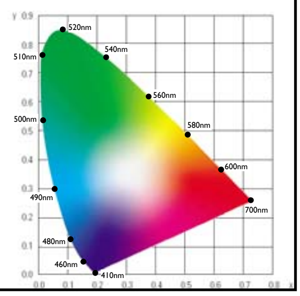
282

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



283

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*

284

XYZ is not perceptually uniform

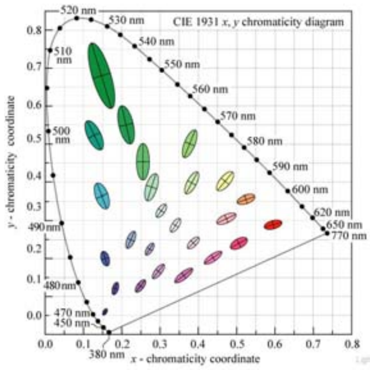


Fig. 17.5. MacAdam ellipses plotted in the CIE 1931 (x, y) chromaticity diagram. The axes of the ellipses are ten times their actual lengths (after MacAdam, 1943; Wright, 1943; MacAdam, 1993).

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

285

Luv was designed to be more uniform

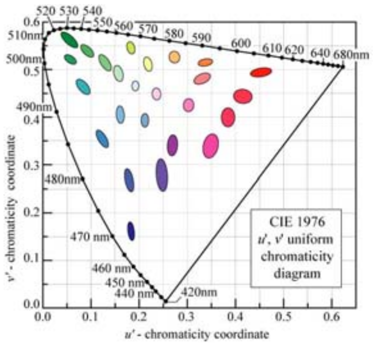
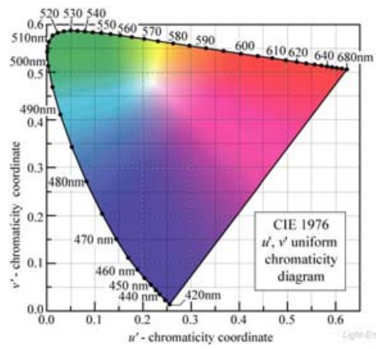


Fig. 17.7. MacAdam ellipses transformed to uniform CIE 1976 (u', v') chromaticity coordinates. For clarity, the axes of the transformed ellipses are ten times their actual lengths. Transformed ellipses are not ellipses in a strict mathematical sense, but their shapes closely resemble those of ellipses.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

286

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.

287

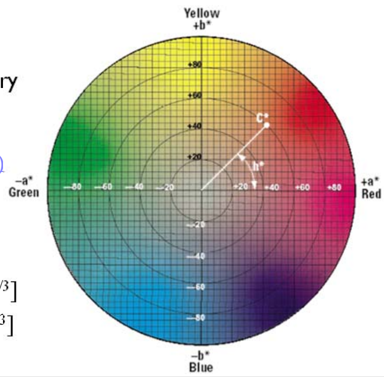
Lab space

- another CIE colour space
- based on complementary colour theory
  - see slide 206 (Colour signals sent to the brain)
- also aims to be perceptually uniform

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

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

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



288

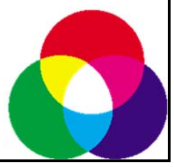
Lab space

- 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

289

RGB space

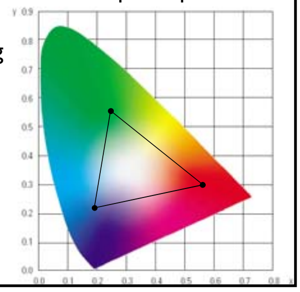
- all display devices which output light mix red, green and blue lights to make colour
  - televisions, CRT monitors, video projectors, LCD screens
- nominally, *RGB* space is a cube
- the device puts physical limitations on:
  - the range of colours which can be displayed
  - the brightest colour which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



290

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



291

CMY space

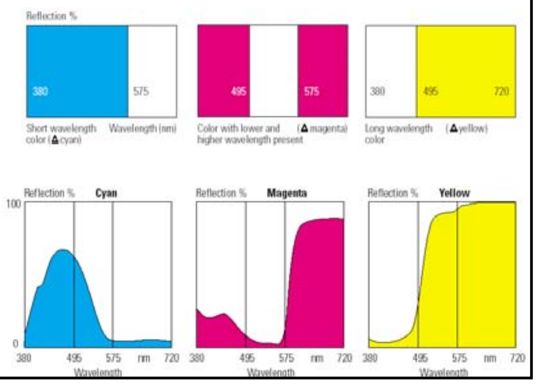
- printers make colour by mixing coloured inks
- the important difference between inks (*CMY*) and lights (*RGB*) is that, while lights *emit* light, inks *absorb* light
  - cyan absorbs red, reflects blue and green
  - magenta absorbs green, reflects red and blue
  - yellow absorbs blue, reflects green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

292

Ideal and actual printing ink reflectivities

ideal

actual

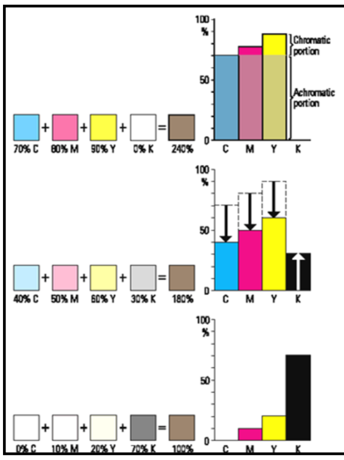


293

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

294



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%

295

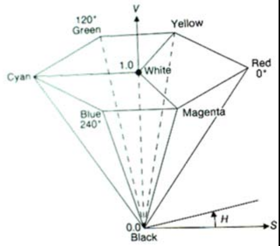
Colour spaces for user-interfaces

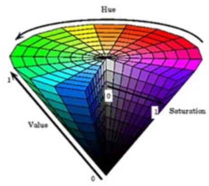
- *RGB* and *CMY* are based on the physical devices which produce the coloured output
- *RGB* and *CMY* are difficult for humans to use for selecting colours
- Munsell's colour system is much more intuitive:
  - hue — what is the principal colour?
  - value — how light or dark is it?
  - chroma — how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

296

HSV: hue saturation value

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



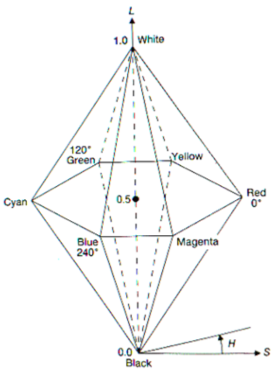


- designed by Alvy Ray Smith in 1978
- algorithm to convert *HSV* to *RGB* and back can be found in Foley et al., Figs 13.33 and 13.34

297

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



298

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* $\Rightarrow$ same intensity, different colour
    - same *X* and *Z*, change *Y* $\Rightarrow$ 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*: $Y = 0.299\,R + 0.587\,G + 0.114\,B$

299

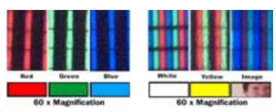
Image display

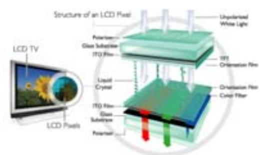
- a handful of technologies cover over 99% of all display devices
  - active displays
    - cathode ray tube — standard for late 20th century
    - liquid crystal display — most common today
    - plasma displays — briefly popular but power-hungry
    - digital mirror displays — increasing use in video projectors
  - printers (passive displays)
    - laser printers — the traditional office printer
    - ink jet printers — the traditional home printer
    - commercial printers — for high volume

300

Liquid crystal displays I

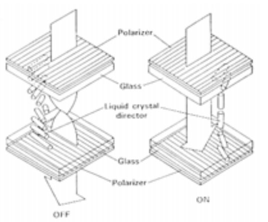
- liquid crystals can twist the polarisation of light
- basic control is by the voltage that is applied across the liquid crystal: either on or off, transparent or opaque
- greyscale can be achieved with some types of liquid crystal by varying the voltage
- colour is achieved with colour filters





301

Liquid crystal displays II



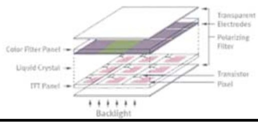
there are two polarizers at right angles to one another on either side of the liquid crystal: under normal circumstances these would block all light

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

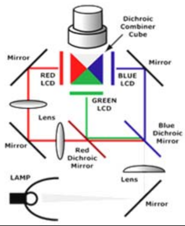
applying a voltage across the liquid crystal causes the molecules to stand on their ends, ruining the twisting phenomenon, so light cannot get through and the pixel is opaque



302

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

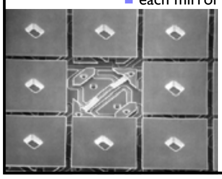


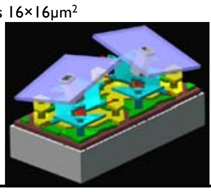
a three LCD video projector, with colour made by devoting one LCD panel to each of red, green and blue, and by splitting the light using dichroic mirrors which pass some wavelengths and reflect others

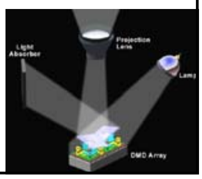
303

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 16×16μm^2



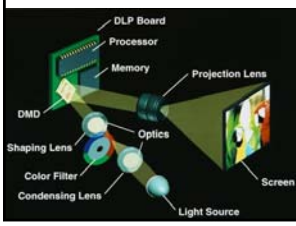


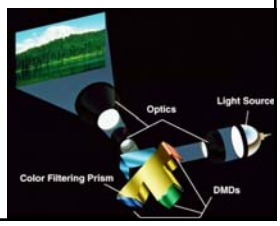


304

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





305

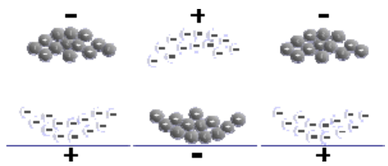
Electrophoretic displays I

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

306

Electrophoretic displays II

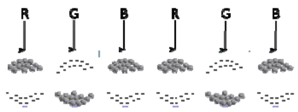
- transparent capsules ~40μ diameter
  - filled with dark oil
  - negatively charged 1μ titanium dioxide particles
- electrodes in substrate attract or repel white particles
- image persists with no power consumption



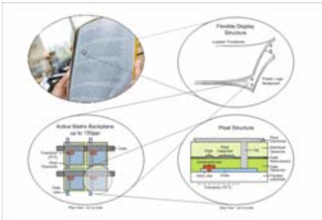
307

Electrophoretic displays III

- colour filters over individual pixels



- flexible substrate using plastic semiconductors (Plastic Logic)



308

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

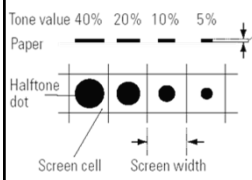
309

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)

310

What about greyscale?

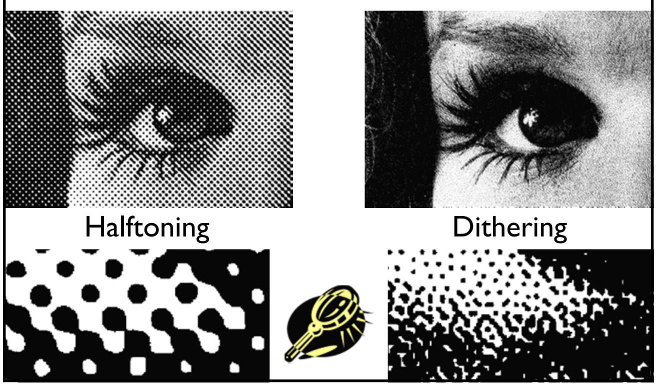


- achieved by halftoning
  - divide image into cells, in each cell draw a spot of the appropriate size for the intensity of that cell
  - on a printer each cell is $m \times m$ pixels, allowing m^2+1 different intensity levels
  - e.g. 300dpi with 4×4 cells ⇒ 75 cells per inch, 17 intensity levels
  - 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*

311

Halftoning & dithering examples



312

What about colour?

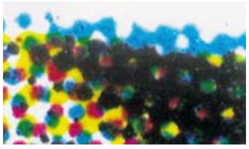
- generally use cyan, magenta, yellow, and black inks (CMYK)
- inks *aborb* colour
  - c.f. lights which *emit* colour
  - CMY is the inverse of RGB
- why is black (K) necessary?
  - inks are not perfect aborbers
  - mixing C + M + Y gives a muddy grey, not black
  - lots of text is printed in black: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

see slide 221 CMYK space

313

How do you produce halftoned colour?

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

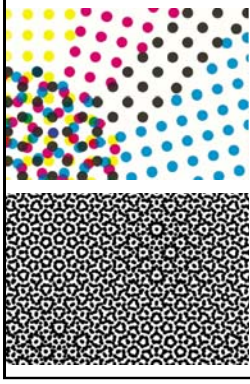


150 lpi × 16 dots per cell = 2400 dpi phototypesetter (16×16 dots per cell = 256 intensity levels)

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

314

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

315

Range of printable colours

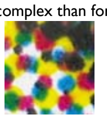
a: colour photography (diapositive)

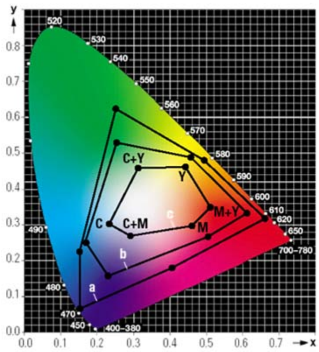
b: high-quality offset printing

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





316

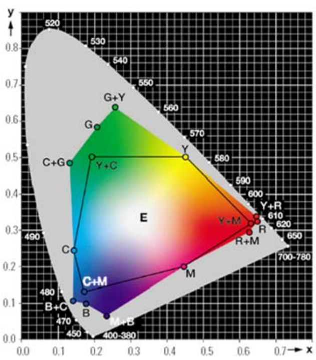
Beyond four colour printing

- 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

317

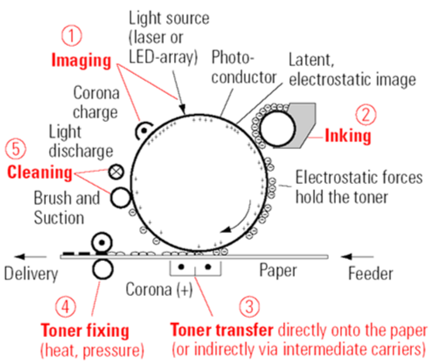
The extra range of colour

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



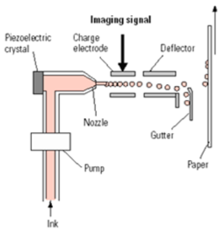
318

Laser printer

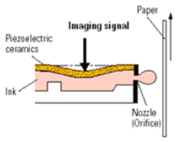


319

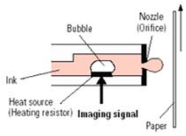
Ink jet printers



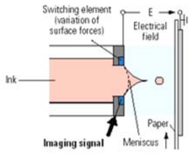
continuous ink jet (left)



piezo ink jet (right)



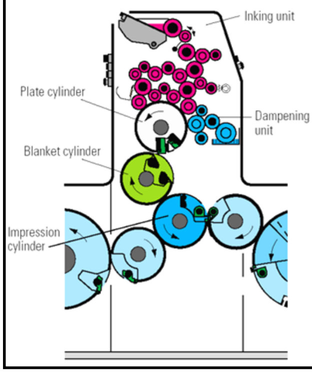
thermal ink jet *or* bubble jet (left)



electrostatic ink jet (right)

320

Commercial offset printing



- 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

321

322

323

324

325

Computer Graphics & Image Processing

- Background
- Simple rendering
- Graphics pipeline
- Underlying algorithms
- Colour and displays
- Image processing
  - Point processing
  - Area processing
  - Rendering

326

Point processing

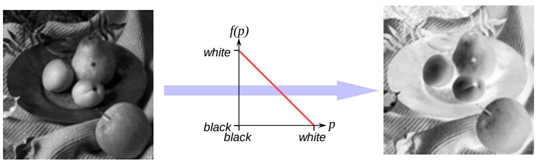
- each pixel's value is modified
- the modification function only takes that pixel's value into account

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

  - 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

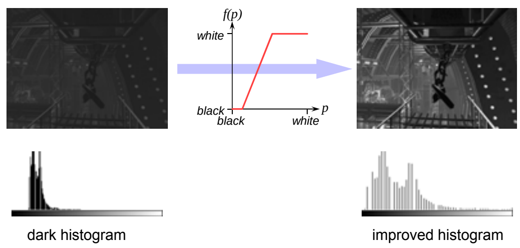
327

Point processing
inverting an image



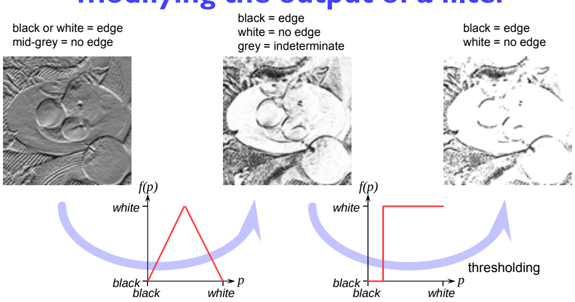
328

Point processing
improving an image's contrast



329

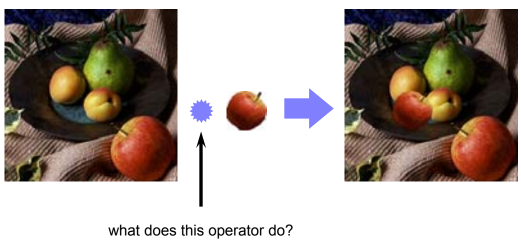
Point processing
modifying the output of a filter



330

Image compositing

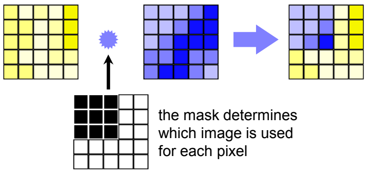
- merging two or more images together



331

Simple compositing

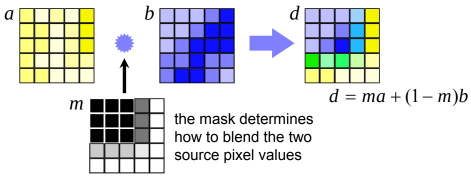
- copy pixels from one image to another
  - only copying the pixels you want
  - use a mask to specify the desired pixels



332

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



333

Differencing – an 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

334

Differencing – an example

take the difference between the two images

$$d = |a - b|$$

where 1 = white and 0 = black

black = no difference
white = large difference

335

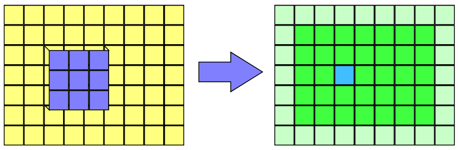
336

337

338

Filtering

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



339

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:

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

340

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

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

341

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

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

342

Example filter - horizontal edge detection

Horizontal edge detection filter

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

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

=

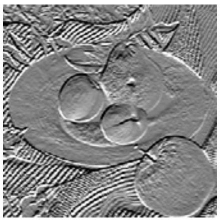
Result

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

343

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

344

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

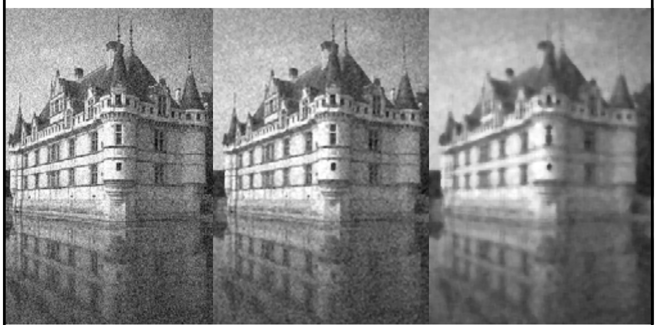
(16,20,22,23, 25, 25,36,37,39) →

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

sort into order and take median

345

Median filter - example



Original noisy image

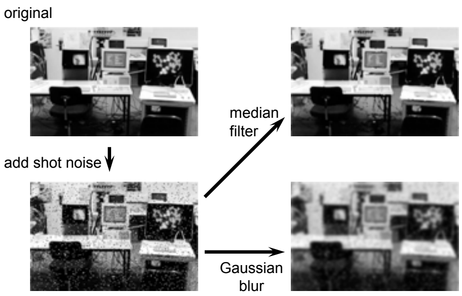
Small median filter reduces the noise

Large median filter reduces noise but blurs

346

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

347

Median filter – as an artistic effect

30×30 median filter

348

Filtering based on local image properties

Photoshop "Crystallize" filter with cell size 20

349

Filtering based on local image properties

Photoshop "Wind" filter

350

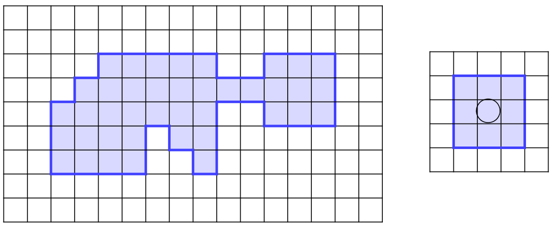
Filtering based on global image properties

Photoshop "Auto Colour" adjustment

351

Morphological image processing

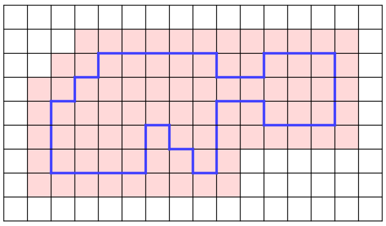
- Consider images as sets of binary pixels
  - Image $I \subseteq \mathbb{Z}^n$ (with $n = 2$ for images)
  - Structuring element $S \subseteq \mathbb{Z}^n$



352

Dilation

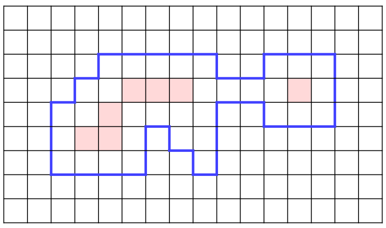
- $I \oplus S = \{i + s | i \in I, s \in S\}$
- expands image by structuring element



353

Erosion

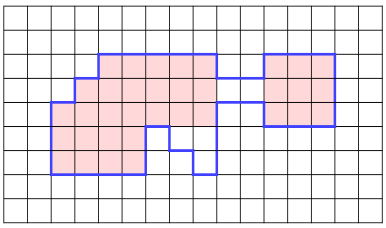
- $I \ominus S = \{x | \forall s \in S. x + s \in I\}$
- set of points where S can be centred to lie entirely inside I



354

Opening

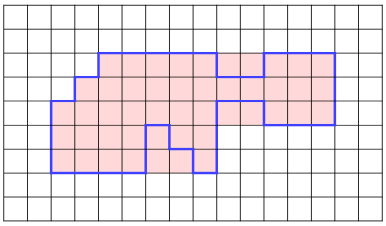
- $I \circ S = (I \ominus S) \oplus S$
- Smooths outlines and breaks narrow links



355

Closing

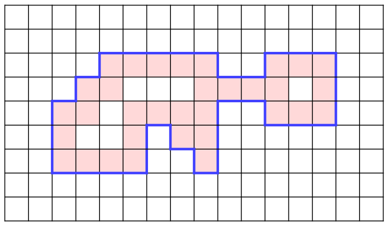
- $I \odot S = (I \oplus S) \ominus S$
- Smooths outlines and joins narrow breaks



356

Boundary

- $\beta(I) = I - (I \ominus B)$
- where B is a solid 3×3 template



357

Morphology with grey scales

- Consider images as functions $f: \mathbb{Z}^2 \rightarrow \mathbb{R}$
  - still with structuring element $S \subseteq \mathbb{Z}^2$
- Dilation: $(f \oplus S)(p) = \max_{s \in S} f(p + s)$
  - largest value in S-shaped region
- Erosion: $(f \ominus S)(p) = \min_{s \in S} f(p + s)$
  - smallest value in S-shaped region
- Same opening and closing

358

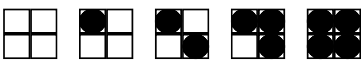
Halftoning & dithering

- 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

359

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



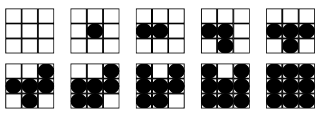
0-51 52-102 103-153 154-204 205-255

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

360

Halftoning dither matrix

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



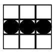
- these patterns can be represented by the *dither matrix*:

7	9	5
2	1	4
6	3	8

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

361

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)

362

Ordered dither

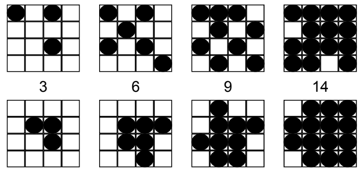
- halftone prints and photocopies well, at the expense of large dots
- an ordered dither matrix produces a nicer visual result than a halftone dither matrix

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

363

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

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

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

364

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

365

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	$f_{i,j} - 255$

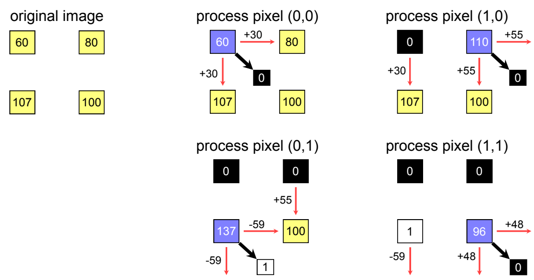
  - each 8-bit value is calculated from pixel and error values:

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

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

366

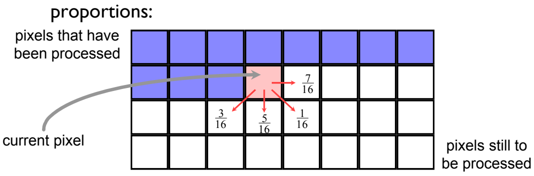
Error diffusion - example (2)



367

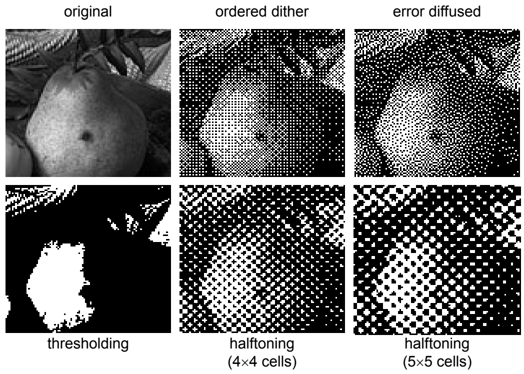
Error diffusion

- 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:



368

Halftoning & dithering — examples



369

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

370

Course review

- Background
- Simple rendering
- Graphics pipeline
- Underlying algorithms
- Colour and displays
- Image processing

371

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

372

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/