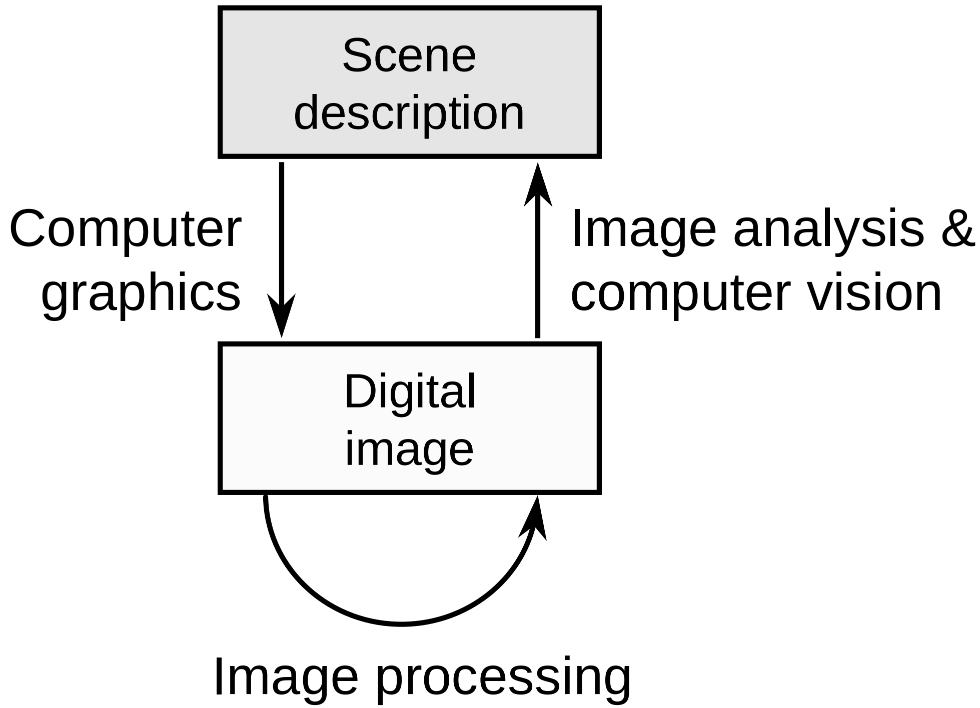
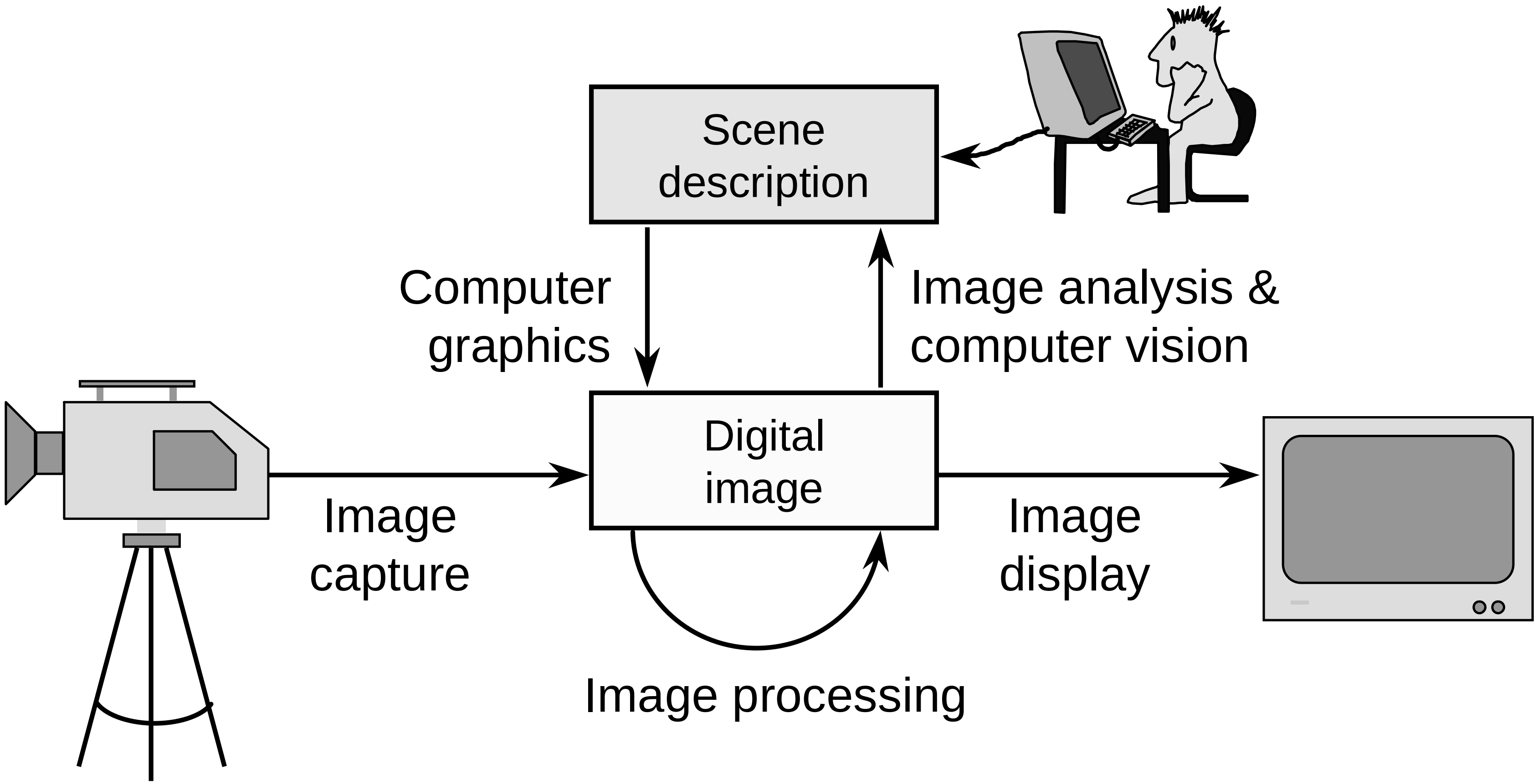
Computer Graphics & Image Processing

- How many lectures are there?
  - Four this term
  - Twelve next term
- Who gets examined on it?
  - Part IB
  - Part II (General)
  - Diploma

What are Computer Graphics & Image Processing?



What are Computer Graphics & Image Processing?



Why bother with CG & IP?

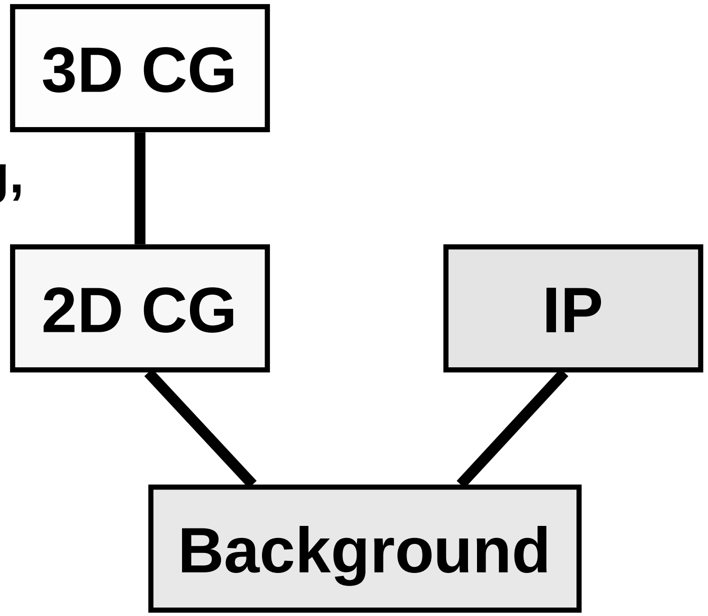
- ***all*** **visual computer output depends on Computer Graphics**
  - **printed output**
  - **monitor (CRT/LCD/whatever)**
  - **all visual computer output consists of real images generated by the computer from some internal digital image**

What are CG & IP used for?

- 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

- background
  - images, human vision, displays
- 2D computer graphics
  - lines, curves, clipping, polygon filling, transformations
- 3D computer graphics
  - projection (3D→2D), surfaces, clipping, transformations, lighting, filling, ray tracing, texture mapping
- image processing
  - filtering, compositing, half-toning, dithering, encoding, compression



Course books

Computer Graphics

- ***Computer Graphics: Principles & Practice***
  - **Foley, van Dam, Feiner & Hughes [1Y919]**
    Addison-Wesley, 1990
    - *Fundamentals of Interactive Computer Graphics*
      Foley & van Dam [1Y104], Addison-Wesley, 1982

Image Processing

- ***Digital Image Processing***
  - **Gonzalez & Woods [U242]**
    Addison-Wesley, 1992
    - *Digital Image Processing*, Gonzalez & Wintz [U135]
    - *Digital Picture Processing*, Rosenfeld & Kak

Past exam questions

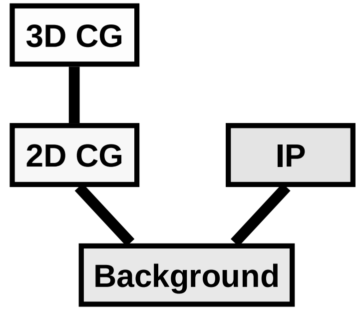
◆ 98/4/10✓	98/5/4✓	98/6/4✓		(11/11,12/4,13/4)
◆ 97/4/10✓	97/5/2?	97/6/4✓		(11/11,12/2,13/4)
◆ 96/4/10?	96/5/4✓	96/6/4✓		(11/11,12/4,13/4)
◆ 95/4/8✓	95/5/4✓	95/6/4✓		(11/9,12/4,13/4)
◆ 94/3/8?	94/4/8?	94/5/4?	94/6/4?	(10/9,11/8,12/4,13/4)
◆ 93/3/9✗	93/4/9?	93/5/4✗	93/6/4✗	(10/9,11/9,12/4,13/4)

✓ could be set exactly as it is

? could be set in a different form or parts could be set

✗ would not be set

- N.B. Diploma/Part II(G) questions in parentheses are identical to the corresponding Part IB questions



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?
- how do displays & printers work?
  - how do we fool the human eye into seeing what we want it to see?

What is an image?

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

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

Image capture

- a variety of devices can be used
  - scanners
    - line CCD in a flatbed scanner
    - spot detector in a drum scanner
  - cameras
    - area CCD

Image capture example

A real image

103 59 12 80 56 12 34 30 1 78 79 21 145 156 52 136 143 65 115 129 41 128 143 50 85
106 11 74 96 14 85 97 23 66 74 23 73 82 29 67 76 21 40 48 7 33 39 9 94 54 19
42 27 6 19 10 3 59 60 28 102 107 41 208 88 63 204 75 54 197 82 63 179 63 46 158 62
46 146 49 40 52 65 21 60 68 11 40 51 17 35 37 0 28 29 0 83 50 15 2 0 1 13 14
8 243 173 161 231 140 69 239 142 89 230 143 90 210 126 79 184 88 48 152 69 35 123 51
27 104 41 23 55 45 9 36 27 0 28 28 2 29 28 7 40 28 16 13 13 1 224 167 112 240
174 80 227 174 78 227 176 87 233 177 94 213 149 78 196 123 57 141 72 31 108 53 22 121
62 22 126 50 24 101 49 35 16 21 1 12 5 0 14 16 11 3 0 0 237 176 83 244 206 123
241 236 144 238 222 147 221 190 108 215 170 77 190 135 52 136 93 38 76 35 7 113 56 26
156 83 38 107 52 21 31 14 7 9 6 0 20 14 12 255 214 112 242 215 108 246 227 133 239
232 152 229 209 123 232 193 98 208 162 64 179 133 47 142 90 32 29 19 27 89 53 21 171
116 49 114 64 29 75 49 24 10 9 5 11 16 9 237 190 82 249 221 122 241 225 129 240 219
126 240 199 93 218 173 69 188 135 33 219 186 79 189 184 93 136 104 65 112 69 37 191 153
80 122 74 28 80 51 19 19 37 47 16 37 32 223 177 83 235 208 105 243 218 125 238 206
103 221 188 83 228 204 98 224 220 123 210 194 109 192 159 62 150 98 40 116 73 28 146 104
46 109 59 24 75 48 18 27 33 33 47 100 118 216 177 98 223 189 91 239 209 111 236 213
117 217 200 108 218 200 100 218 206 104 207 175 76 177 131 54 142 88 41 108 65 22 103
59 22 93 53 18 76 50 17 9 10 2 54 76 74 108 111 102 218 194 108 228 203 102 228 200
100 212 180 79 220 182 85 198 158 62 180 138 54 155 106 37 132 82 33 95 51 14 87 48
15 81 46 14 16 15 0 11 6 0 64 90 91 54 80 93 220 186 97 212 190 105 214 177 86 208
165 71 196 150 64 175 127 42 170 117 49 139 89 30 102 53 12 84 43 13 79 46 15 72 42
14 10 13 4 12 8 0 69 104 110 58 96 109 130 128 115 196 154 82 196 148 66 183 138 70
174 125 56 169 120 54 146 97 41 118 67 24 90 52 16 75 46 16 58 42 19 13 7 9 10 5
0 18 11 3 66 111 116 70 100 102 78 103 99 57 71 82 162 111 66 141 96 37 152 102 51
130 80 31 110 63 21 83 44 11 69 42 12 28 8 0 7 5 10 18 4 0 17 10 2 30 20 10
58 88 96 53 88 94 59 91 102 69 99 110 54 80 79 23 69 85 31 34 25 53 41 25 21 2
0 8 0 0 17 10 4 11 0 0 34 21 13 47 35 23 38 26 14 47 35 23

A digital image

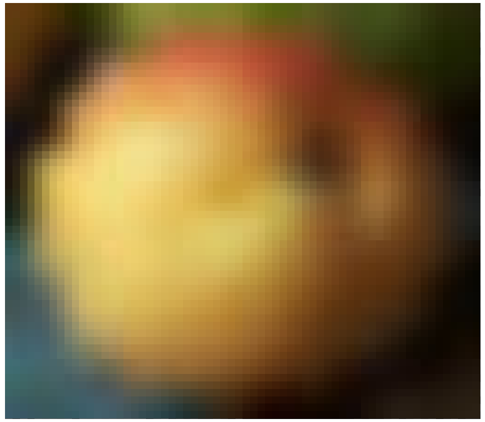
Image display

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

Image display example

103 59 12 80 56 12 34 30 1 78 79 21 145 156 52 136 143 65 115 129 41 128 143 50 85
106 11 74 96 14 85 97 23 66 74 23 73 82 29 67 76 21 40 48 7 33 39 9 94 54 19
42 27 6 19 10 3 59 60 28 102 107 41 208 88 63 204 75 54 197 82 63 179 63 46 158 62
46 146 49 40 52 65 21 60 68 11 40 51 17 35 37 0 28 29 0 83 50 15 2 0 1 13 14
8 243 173 161 231 140 69 239 142 89 230 143 90 210 126 79 184 88 48 152 69 35 123 51
27 104 41 23 55 45 9 36 27 0 28 28 2 29 28 7 40 28 16 13 13 1 224 167 112 240
174 80 227 174 78 227 176 87 233 177 94 213 149 78 196 123 57 141 72 31 108 53 22 121
62 22 126 50 24 101 49 35 16 21 1 12 5 0 14 16 11 3 0 0 237 176 83 244 206 123
241 236 144 238 222 147 221 190 108 215 170 77 190 135 52 136 93 38 76 35 7 113 56 26
156 83 38 107 52 21 31 14 7 9 6 0 20 14 12 255 214 112 242 215 108 246 227 133 239
232 152 229 209 123 232 193 98 208 162 64 179 133 47 142 90 32 29 19 27 89 53 21 171
116 49 114 64 29 75 49 24 10 9 5 11 16 9 237 190 82 249 221 122 241 225 129 240 219
126 240 199 93 218 173 69 188 135 33 219 186 79 189 184 93 136 104 65 112 69 37 191 153
80 122 74 28 80 51 19 19 37 47 16 37 32 223 177 83 235 208 105 243 218 125 238 206
103 221 188 83 228 204 98 224 220 123 210 194 109 192 159 62 150 98 40 116 73 28 146 104
46 109 59 24 75 48 18 27 33 33 47 100 118 216 177 98 223 189 91 239 209 111 236 213
117 217 200 108 218 200 100 218 206 104 207 175 76 177 131 54 142 88 41 108 65 22 103
59 22 93 53 18 76 50 17 9 10 2 54 76 74 108 111 102 218 194 108 228 203 102 228 200
100 212 180 79 220 182 85 198 158 62 180 138 54 155 106 37 132 82 33 95 51 14 87 48
15 81 46 14 16 15 0 11 6 0 64 90 91 54 80 93 220 186 97 212 190 105 214 177 86 208
165 71 196 150 64 175 127 42 170 117 49 139 89 30 102 53 12 84 43 13 79 46 15 72 42
14 10 13 4 12 8 0 69 104 110 58 96 109 130 128 115 196 154 82 196 148 66 183 138 70
174 125 56 169 120 54 146 97 41 118 67 24 90 52 16 75 46 16 58 42 19 13 7 9 10 5
0 18 11 3 66 111 116 70 100 102 78 103 99 57 71 82 162 111 66 141 96 37 152 102 51
130 80 31 110 63 21 83 44 11 69 42 12 28 8 0 7 5 10 18 4 0 17 10 2 30 20 10
58 88 96 53 88 94 59 91 102 69 99 110 54 80 79 23 69 85 31 34 25 53 41 25 21 2
0 8 0 0 17 10 4 11 0 0 34 21 13 47 35 23 38 26 14 47 35 23

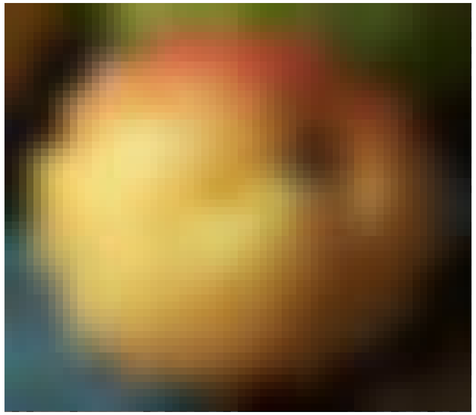
The image data



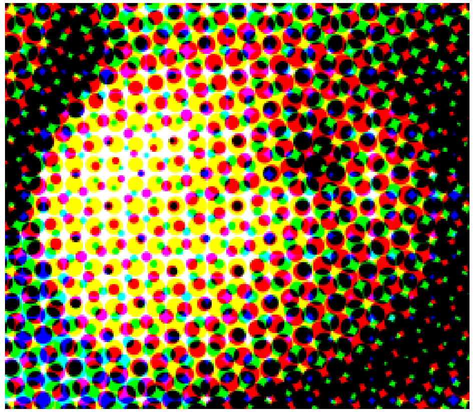
Displayed on a CRT

Different ways of displaying the same digital image

Nearest-neighbour
e.g. LCD



Gaussian
e.g. CRT



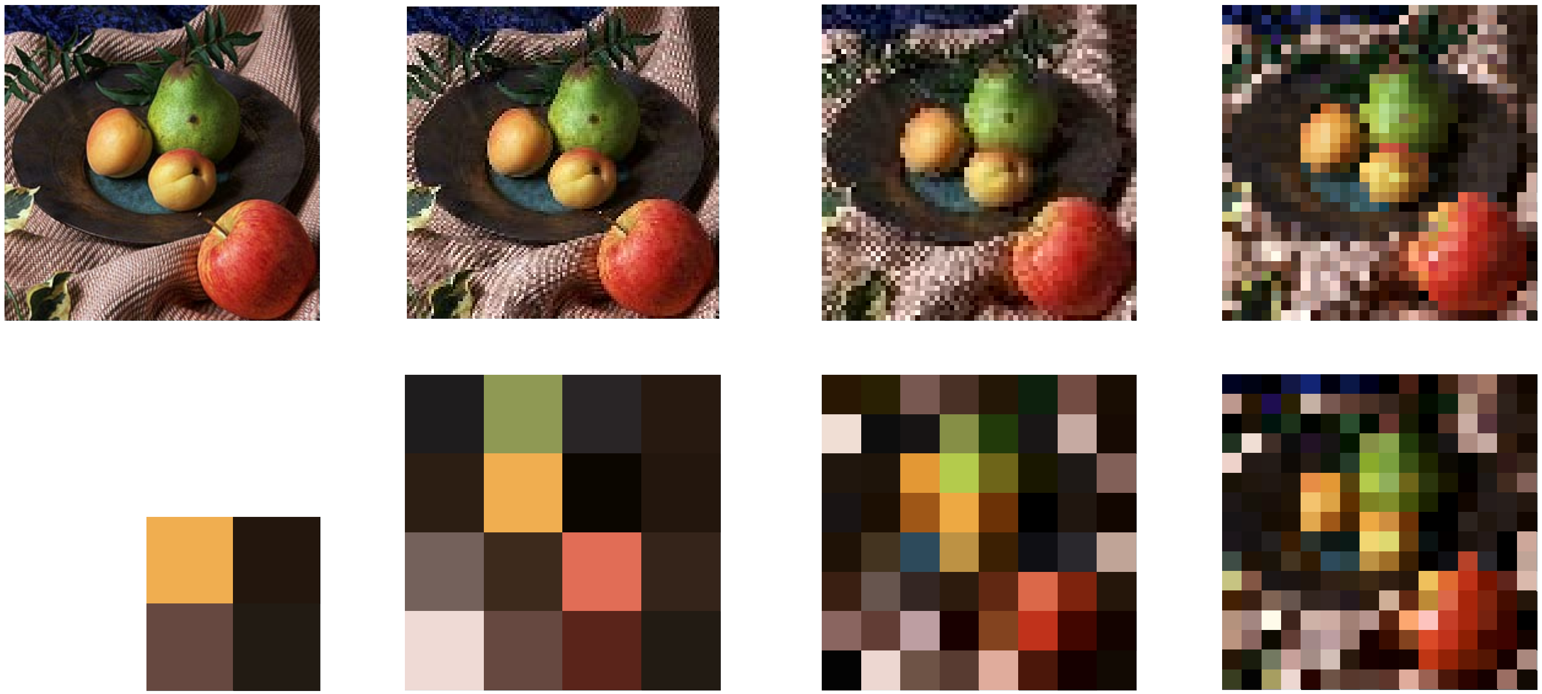
Half-toning
e.g. laser printer

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 printers have resolutions between 300 and 1200 ppi

Sampling resolution

256×256 128×128 64×64 32×32



2×2 4×4 8×8 16×16

Quantisation

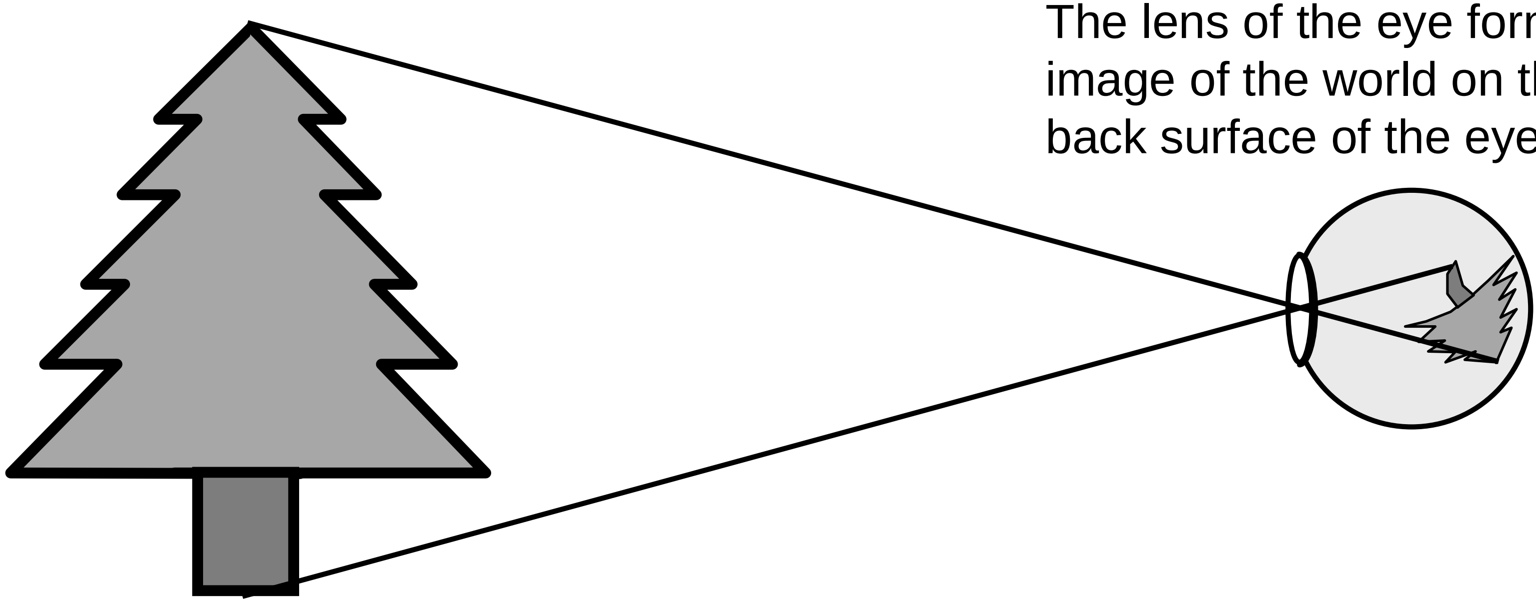
- each intensity value is a number
- for digital storage the intensity values must be quantised
  - limits the number of different intensities that can be stored
  - limits the brightest intensity that can be stored
- how many intensity levels are needed for human consumption
  - 8 bits usually sufficient
  - some applications use 10 or 12 bits

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)

Human visual system



The lens of the eye forms an image of the world on the back surface of the eye

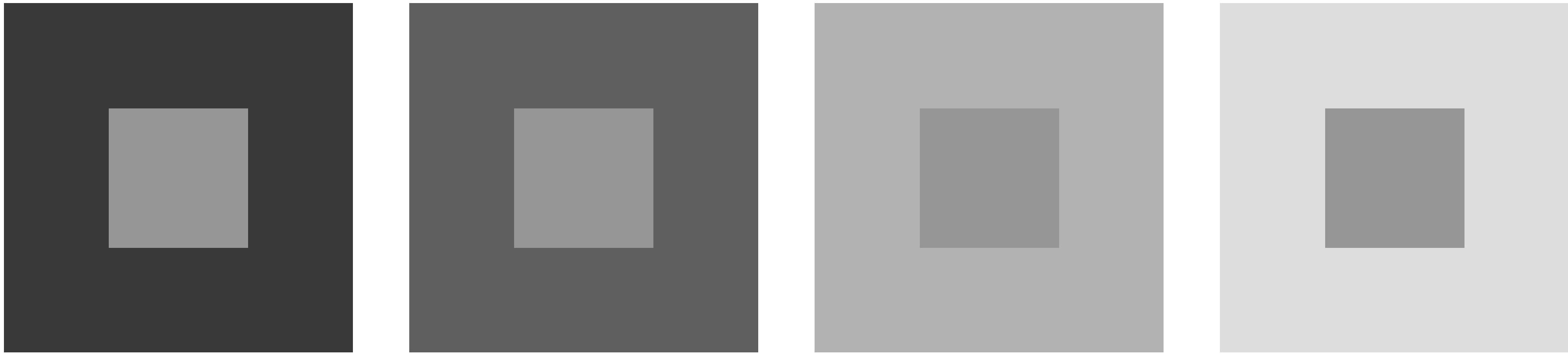
GW Fig 2.1, 2.2; Sec 2.1.1
FLS Fig 35-2

Things eyes do

- **discrimination**
  - **discriminates between different intensities and colours**
- **adaptation**
  - **adapts to changes in illumination level and colour**
- **simultaneous contrast**
  - **adapts locally within an image**
- **persistence**
  - **integrates light over a period of about 1/30 second**
- **edge enhancement**
  - **causes Mach banding effects**

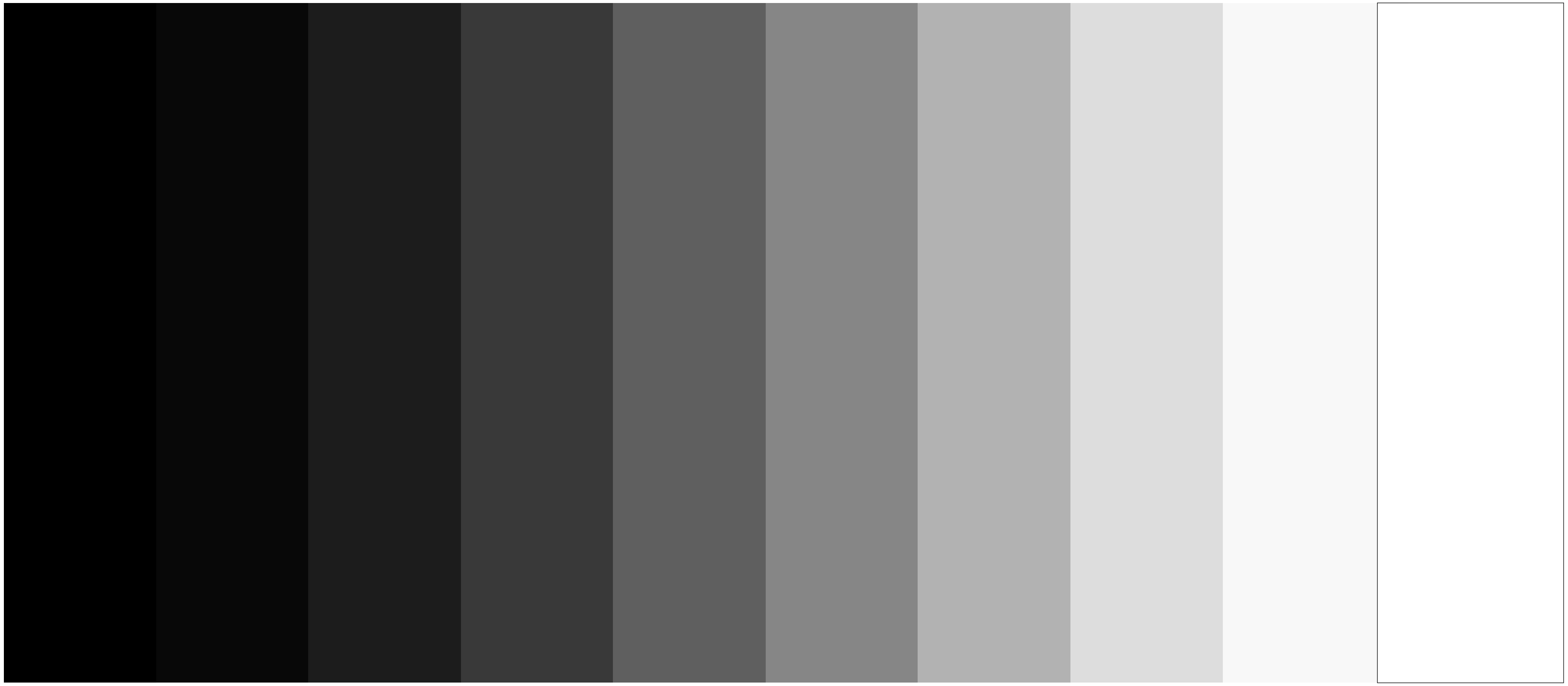
GLA Fig 1.17
GW Fig 2.4

Simultaneous contrast



The centre square is the same intensity in all four cases

Mach bands



Each of the nine rectangles is a constant colour

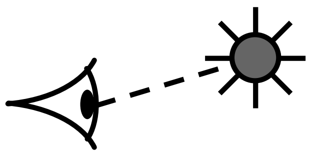
Foveal vision

- 150,000 cones per square millimetre in the fovea
  - high resolution
  - colour
- outside fovea: mostly rods
  - lower resolution
  - monochromatic
  - peripheral vision
    - allows you to keep the high resolution region in context
    - allows you to avoid being hit by passing branches

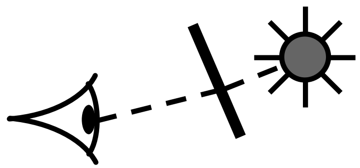
GW Fig 2.1, 2.2

What is required for vision?

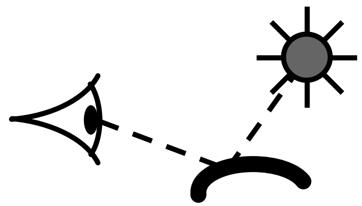
- **illumination**
  - **some source of light**
- **objects**
  - **which reflect (or transmit) the light**
- **eyes**
  - **to capture the light as an image**



direct viewing



transmission



reflection

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

MIN Fig 22a

- 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

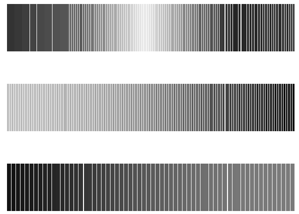
MIN Examples 1 & 2

Classifying colours

- we want some way of classifying colours and, preferably, quantifying them
- 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

Munsell's colour classification system

- ✦ three axes
  - ■ hue ➢ the dominant colour
  - ■ lightness ➢ bright colours/dark colours
  - ■ saturation ➢ vivid colours/dull colours



  - ◆ can represent this as a 3D graph
- ✦ any two adjacent colours are a standard "perceptual" distance apart
  - ◆ worked out by testing it on people
- ✦ but how does the eye *actually* see colour?

MIN Fig 4
Colour plate 1

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

Colour vision

- ✦ three types of cone
  - ◆ each responds to a different spectrum
    - ■ roughly can be defined as red, green, and blue
    - ■ each has a response function $r(\lambda)$, $g(\lambda)$, $b(\lambda)$
  - ◆ different sensitivities to the different colours
    - ■ roughly 40:20:1
    - ■ so cannot see fine detail in blue
  - ◆ overall intensity response of the eye can be calculated
    - ■ $y(\lambda) = r(\lambda) + g(\lambda) + b(\lambda)$
    - ■ $y = k \int P(\lambda)\, y(\lambda)\, d\lambda$ is the perceived *luminance*

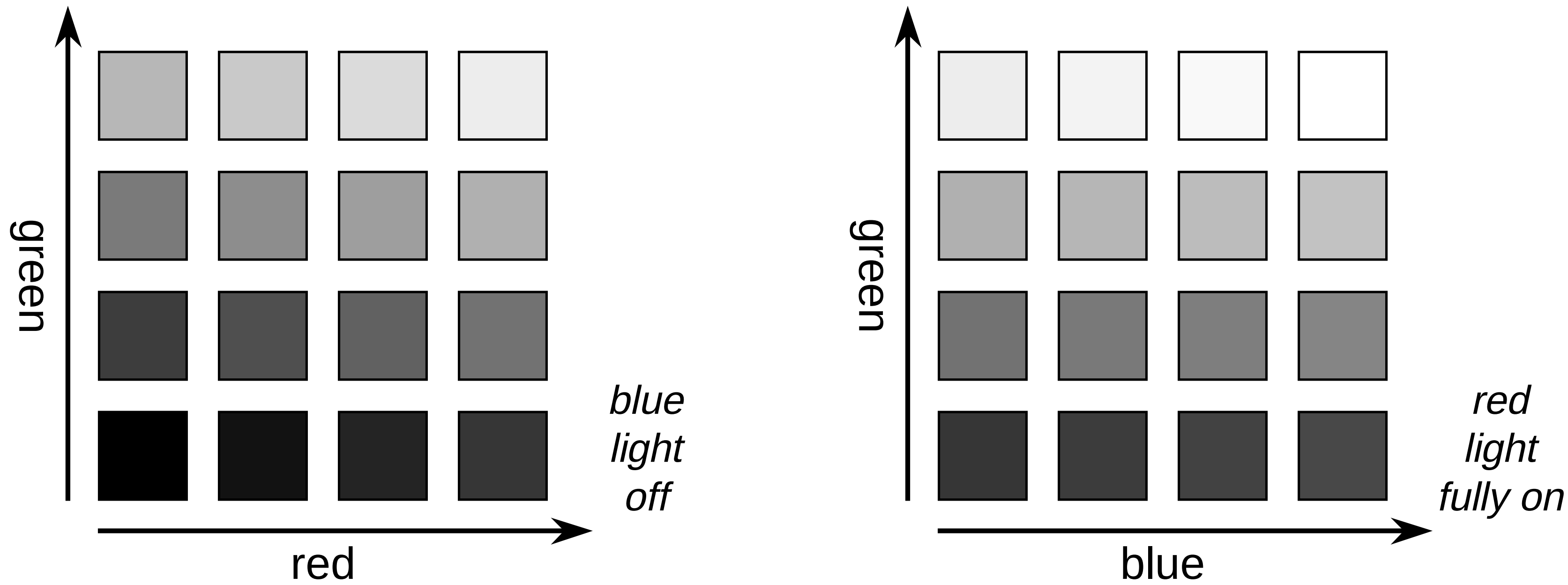
JMF Fig 20b

Chromatic metamerism

- **many different spectra will induce the same response in our cones**
  - **the values of the three perceived values can be calculated as:**
    - $r = k \int P(\lambda)\, r(\lambda)\, d\lambda$
    - $g = k \int P(\lambda)\, g(\lambda)\, d\lambda$
    - $b = k \int P(\lambda)\, b(\lambda)\, d\lambda$
  - **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 r, g, b**
  - **we can thus fool the eye into seeing (almost) any colour by mixing correct proportions of some small number of lights**

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



XYZ colour space

FvDFH Sec 13.2.2
Figs 13.20, 13.22, 13.23

- **not every wavelength can be represented as a mix of red, green, and blue**
- **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)

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**

FvDFH Fig 13.24
Colour plate 2

- **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**

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 purples, blues, or greens

FvDFH Figs 13.26, 13.27

Colour spaces

- CIE ***XYZ***, ***Yxy***
- Pragmatic
  - used because they relate directly to the way that the hardware works
  - ***RGB***, ***CMY***, ***YIQ***, ***YUV***

FvDFH Fig 13.28

YIQ and YUV are used in broadcast television

- Munsell-like
  - considered by many to be easier for people to use than the pragmatic colour spaces
  - ***HSV***, ***HLS***

FvDFH Figs 13.30, 13,35

- Uniform
  - equal steps in any direction make equal perceptual differences
  - ***L*a*b****, ***L*u*v****

GLA Figs 2.1, 2.2; Colour plates 3 & 4

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
- **some other systems use three colour co-ordinates**
  - **luminance can then be derived as some function of the three**
    - e.g. in RGB: $Y = 0.299\,R + 0.587\,G + 0.114\,B$

Implications of vision on resolution

- in theory you can see about 600dpi, 30cm from your eye
- in practice, opticians say that 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

- resolution decreases as contrast decreases
- colour resolution is much worse than intensity resolution
  - hence YIQ and YUV for TV broadcast

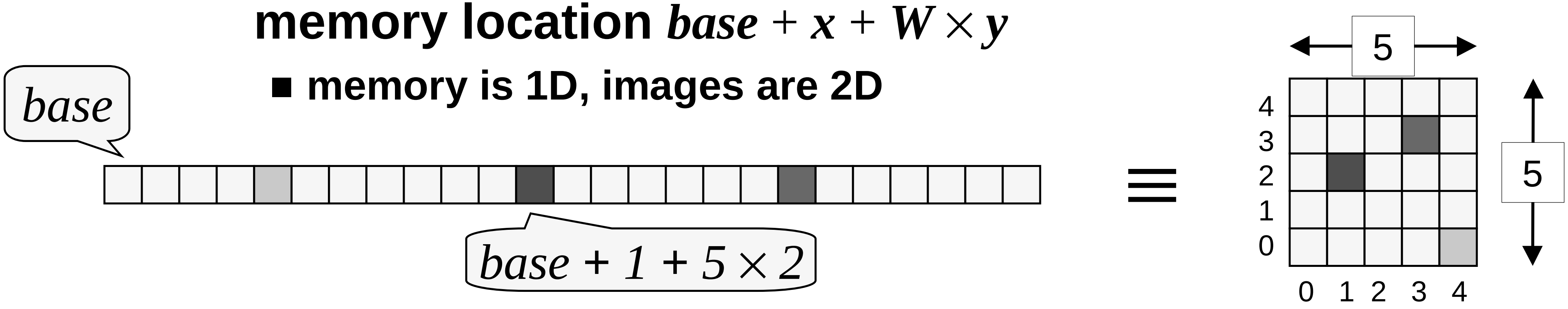
Implications of vision on quantisation

- humans can distinguish, at best, about a 2% change in intensity
  - not so good at distinguishing colour differences
- for TV $\Rightarrow$ 10 bits of intensity information
  - 8 bits is usually sufficient
    - why use only 8 bits? why is it usually acceptable?
  - for movie film $\Rightarrow$ 14 bits of intensity information

> for TV the brightest white is about 25x as bright as the darkest black
>
> movie film has about 10x the contrast ratio of TV

Storing images in memory

- 8 bits has become a *de facto* standard for greyscale images
  - 8 bits = 1 byte
  - an image of size $W \times H$ can therefore 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

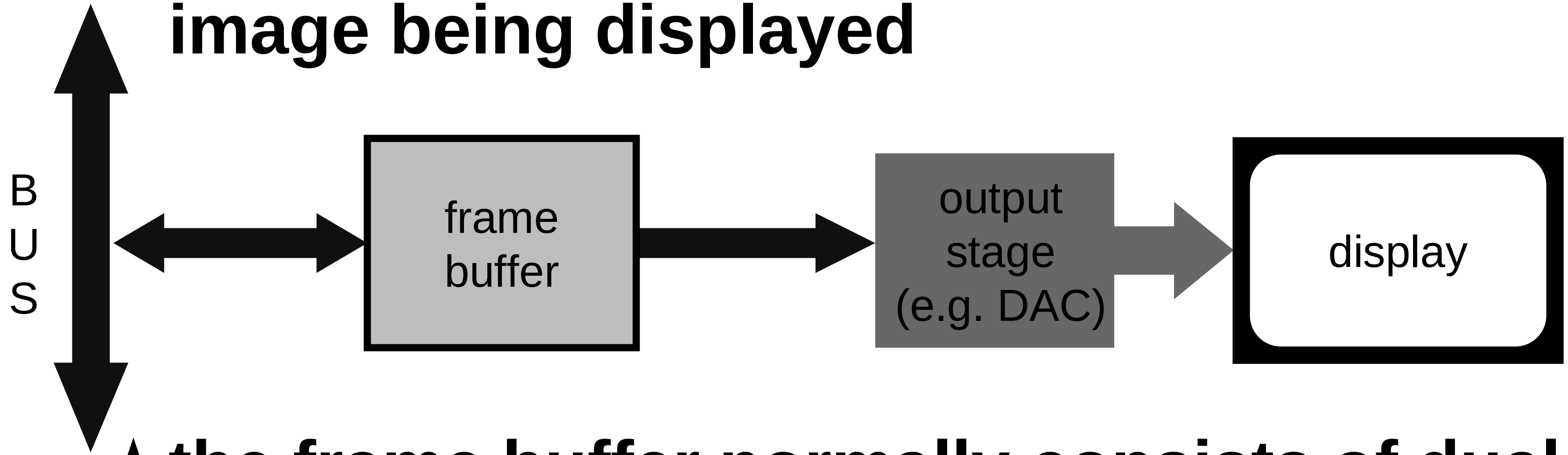


Colour images

- **tend to be 24 bits per pixel**
  - **3 bytes: one red, one green, one blue**
- **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*)**

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)

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

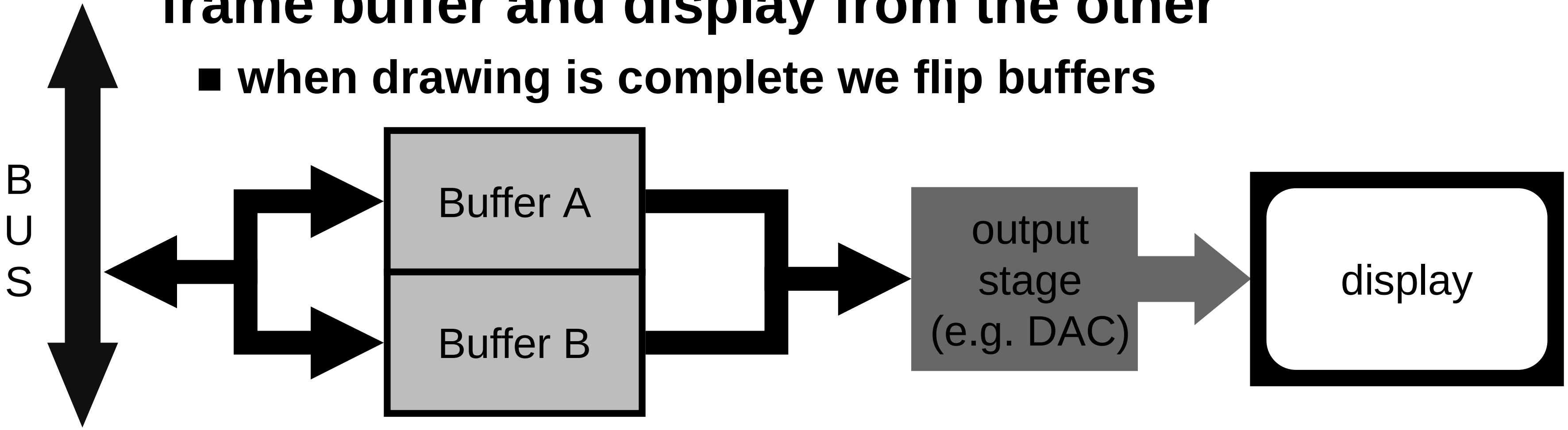


Image display

- three technologies cover over 99% of all display devices
  - cathode ray tube
  - liquid crystal display
  - printer

Liquid crystal display

- liquid crystal can twist the polarisation of light
- control is by the voltage that is applied across the liquid crystal
  - either on or off: transparent or opaque
- greyscale can be achieved in some liquid crystals by varying the voltage
- colour is achieved with colour filters
- low power consumption but image quality not as good as cathode ray tubes

JMF Figs 90, 91

Cathode ray tubes

- focus an electron gun on a phosphor screen
  - produces a bright spot
- scan the spot back and forth, up and down to cover the whole screen
- vary the intensity of the electron beam to change the intensity of the spot
- repeat this fast enough and humans see a continuous picture

CRT slides in handout

How fast do CRTs need to be?

- speed at which entire screen is updated is called the “refresh rate”
- 50Hz (PAL TV, used in most of Europe)
  - many people can see a slight flicker
- 60Hz (NTSC TV, used in USA and Japan)
  - better
- 80-90Hz
  - 99% of viewers see no flicker, even on very bright displays
- 100HZ (newer “flicker-free” PAL TV sets)
  - practically no-one can see the image flickering

Flicker/resolution trade-off

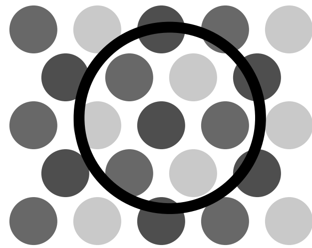
PAL 50Hz
768x576

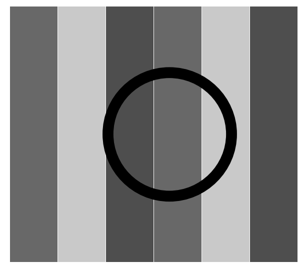
NTSC 60Hz
640x480

Colour CRTs: shadow masks

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

FvDFH Fig 4.14





CRT *vs* LCD

		CRT	***LCD***
Physical	Screen size	Excellent	Fair
	Depth	Poor	Excellent
	Weight	Poor	Excellent
	Brightness	Excellent	Fair/Good
Intensity & colour	Contrast	Good/Excellent	Fair
	Intensity levels	Excellent	Fair
	Viewing angle	Excellent	Poor
	Colour capability	Excellent	Good
Financial	Cost	Low	Low
	Power consumption	Fair	Excellent

Printers

- many types of printer
  - ink jet
    - sprays ink onto paper
  - dot matrix
    - pushes pins against an ink ribbon and onto the 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
- all make marks on paper
  - essentially binary devices: mark/no mark

Printer resolution

- ink jet
  - up to 360dpi
- laser printer
  - up to 1200dpi
  - generally 600dpi
- phototypesetter
  - up to about 3000dpi
- bi-level devices: each pixel is either black or white

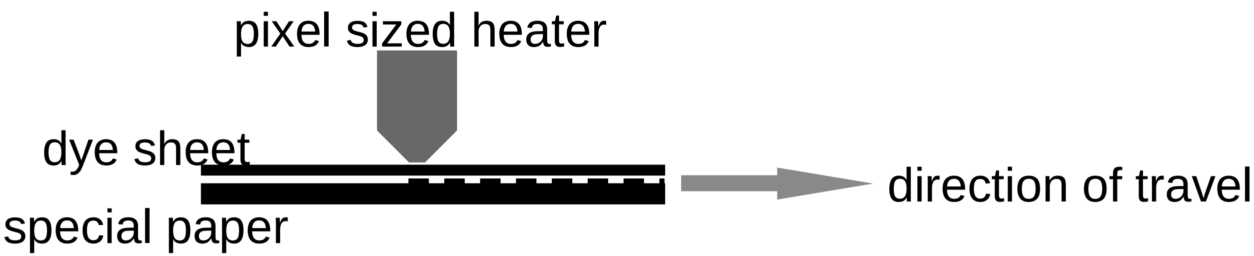
What about greyscale?

- **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 $\Rightarrow$ 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*

Dye sublimation printers: true greyscale

- **dye sublimation gives true greyscale**



- **dye sublimes off dye sheet and onto paper in proportion to the heat level**

- **colour is achieved by using four different coloured dye sheets in sequence — the heat mixes them**

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

JMF Fig 9b

How do you produce halftoned colour?

- **print four halftone screens, one in each colour**

Colour plate 5

- **carefully angle the screens to prevent interference (moiré) patterns**

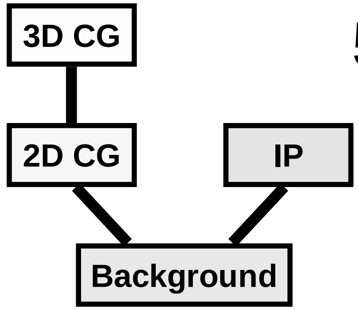
Standard angles	
Magenta	45°
Cyan	15°
Yellow	90°
Black	75°

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 leavlets
200 lpi	very smooth, expensive paper very high quality publication

2D Computer Graphics



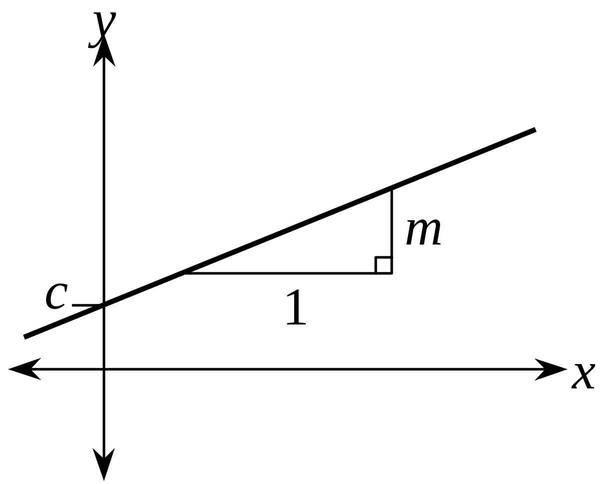
- **lines**
  - **how do I draw a straight line?**
- **curves**
  - **how do I specify curved lines?**
- **clipping**
  - **what about lines that go off the edge of the screen?**
- **filled areas**
- **transformations**
  - **scaling, rotation, translation, shearing**
- **applications**

Drawing a straight line

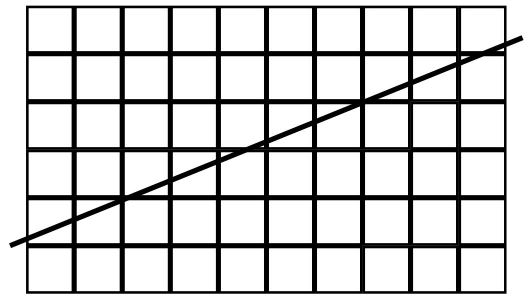
- **a straight line can be defined by:**

$$y = mx + c$$

the slope of the line

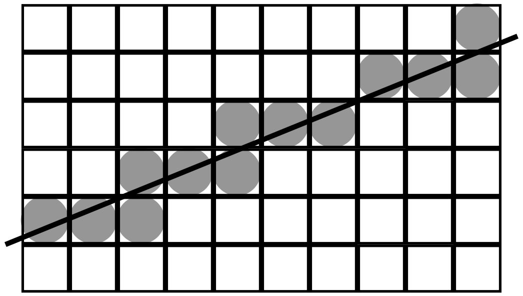


- **a mathematical line is "length without breadth"**
- **a computer graphics line is a set of pixels**
- **which pixels do we need to turn on to draw a given line?**



Which pixels do we use?

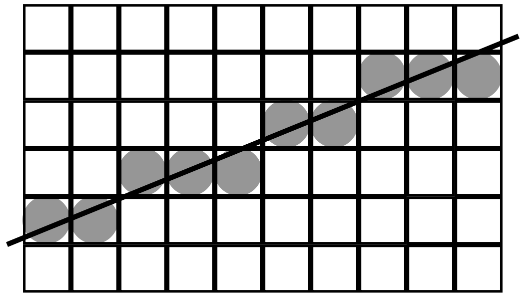
- **there are two reasonably sensible alternatives:**



every pixel through which
the line passes

(can have either one or two
pixels in each column)

✘



the “closest” pixel to the
line in each column

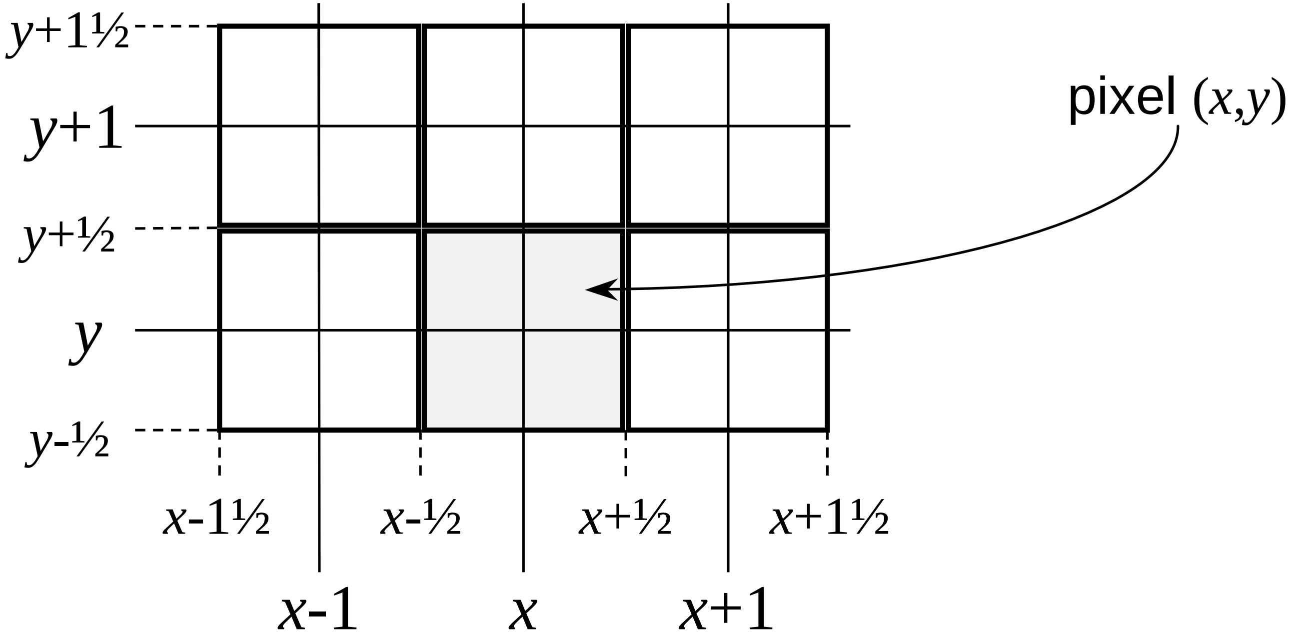
(always have just one pixel
in every column)

✓

- **in general, use this**

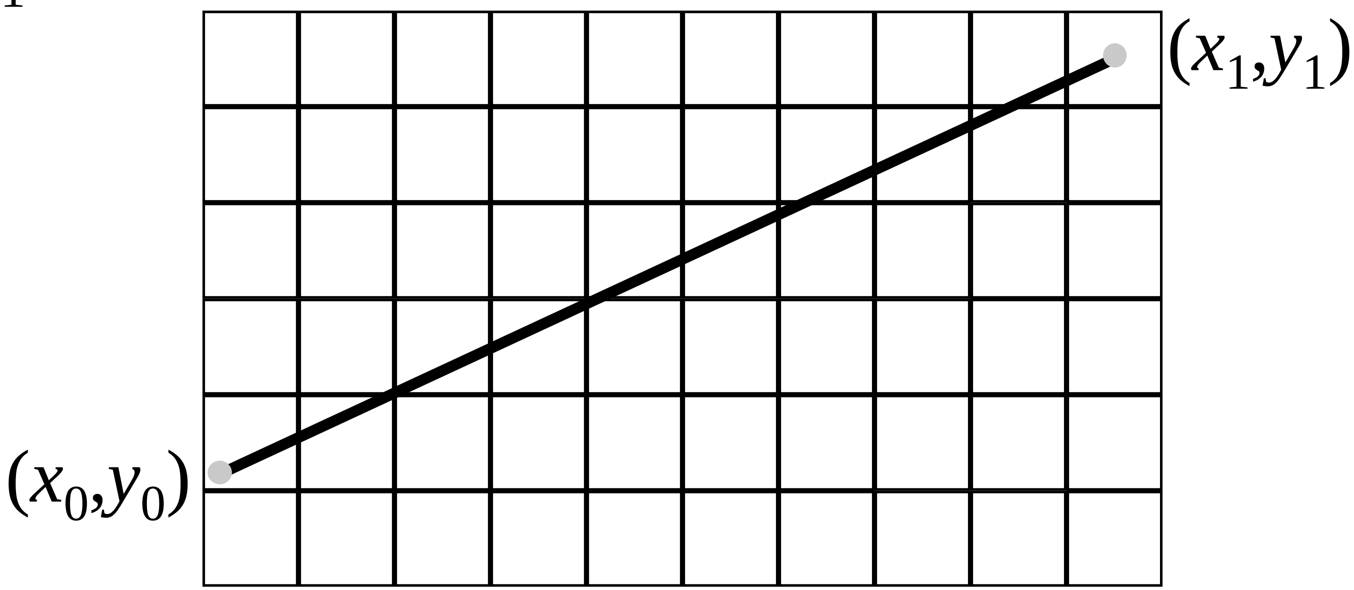
A line drawing algorithm - preparation 1

- **pixel** (x,y) **has its centre at real co-ordinate** (x,y)
  - **it thus stretches from** $(x-\frac{1}{2}, y-\frac{1}{2})$ **to** $(x+\frac{1}{2}, y+\frac{1}{2})$



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** $(0 \leq m \leq 1)$
- $x_0 < x_1$



Bresenham's line drawing algorithm 1

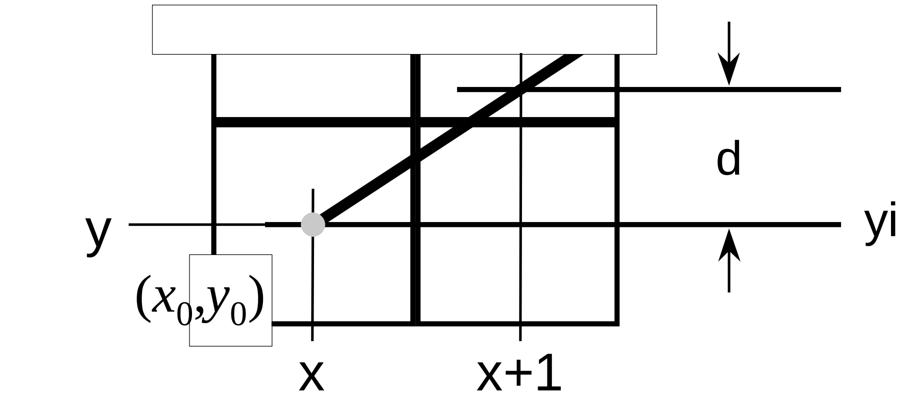
Initialisation

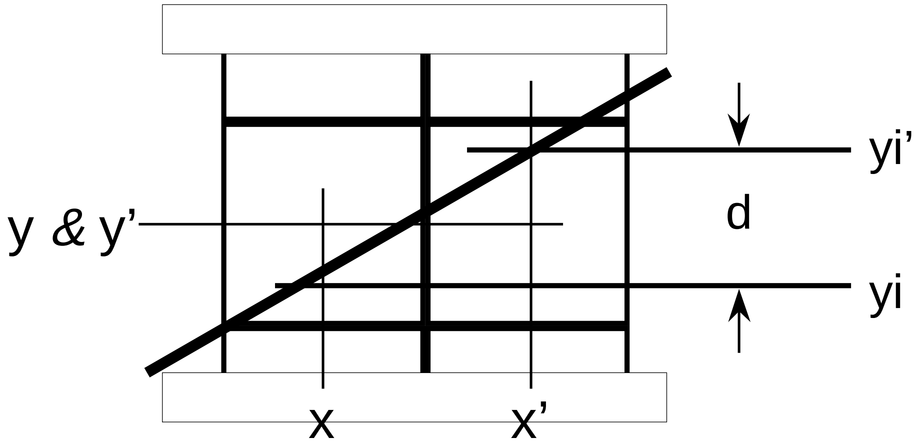
d = (y_1 - y_0) / (x_1 - x_0)
x = x_0
yi = y_0
y = y_0
DRAW(x,y)

Iteration

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

assumes integer end points





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

Bresenham's line drawing algorithm 2

- naïve algorithm involves floating point arithmetic & rounding inside the loop
  $\Rightarrow$ slow
- Speed up A:
  - separate integer and fractional parts of yi (into y and yf)
  - replace rounding by an IF
    - removes need to do rounding

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

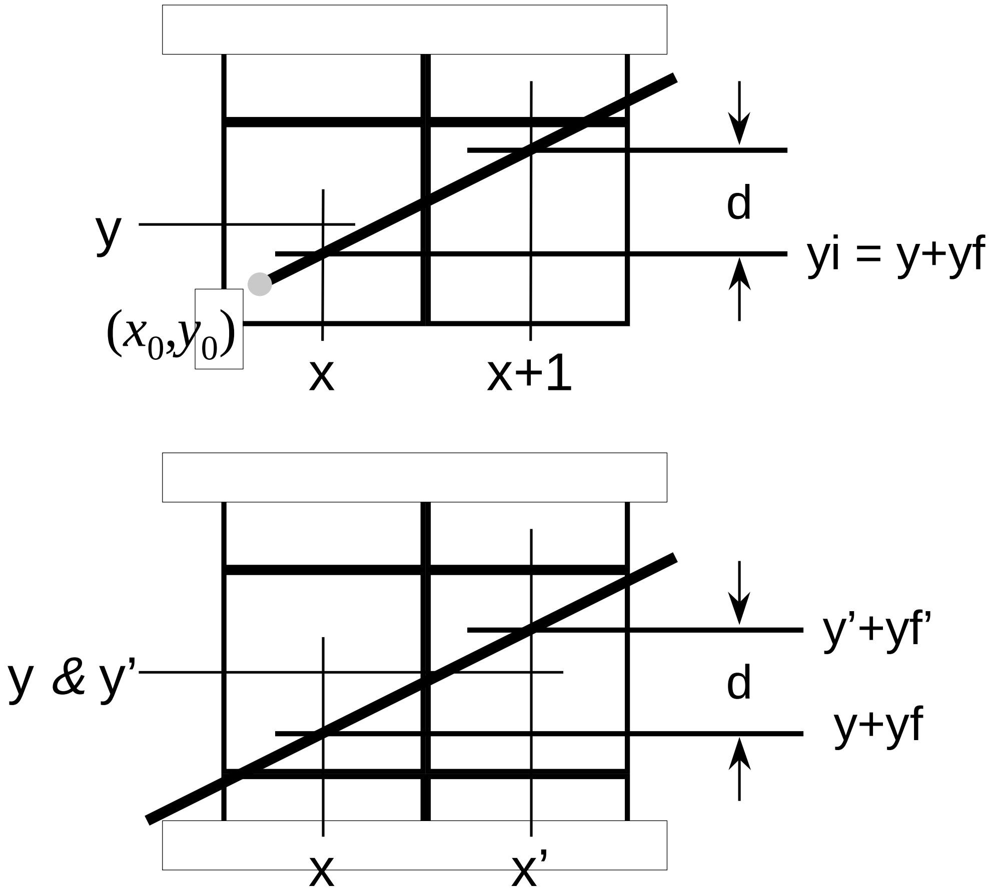
Bresenham's line drawing algorithm 3

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

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

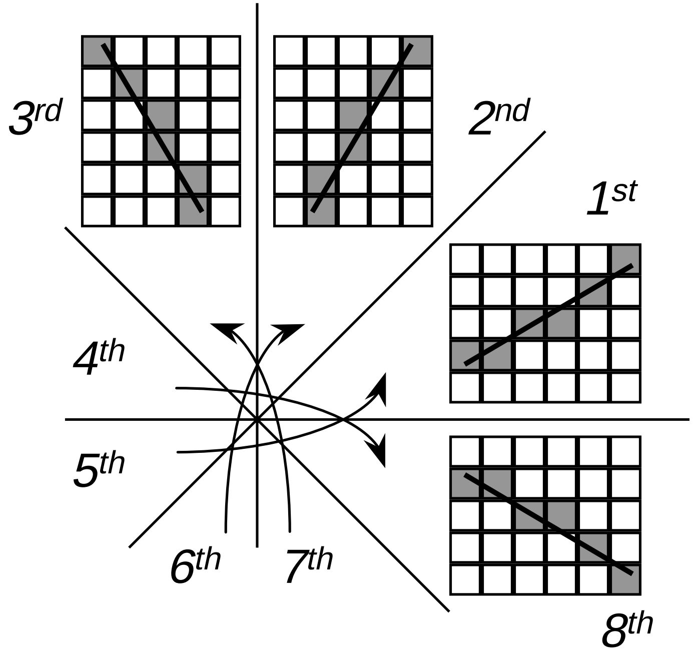
Bresenham's algorithm for floating point end points


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



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



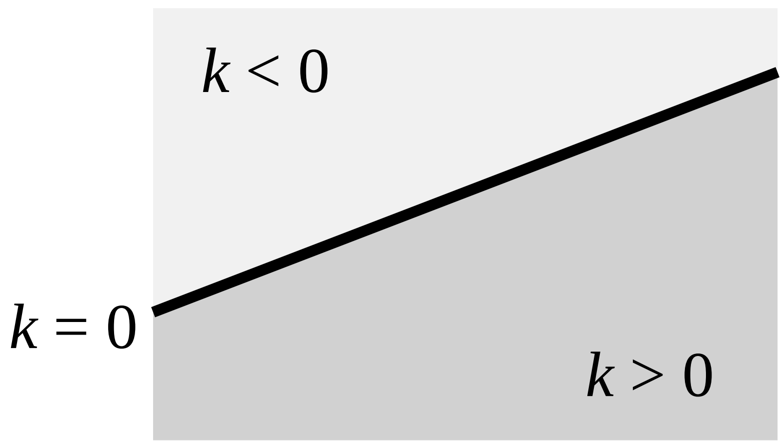
Exercise: work out what changes need to be made to the algorithm for it to work in each of the other three octants

A second line drawing algorithm

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

$$k = ax + by + c$$

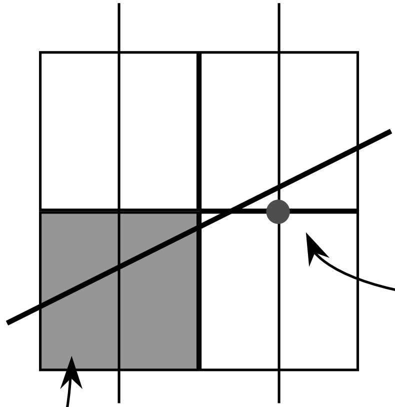
- this divides the plane into three regions:
  - above the line $k < 0$
  - below the line $k > 0$
  - on the line $k = 0$



Midpoint line drawing algorithm 1

- 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



Evaluate the decision variable at this point

if ≥ 0 then go NE

if < 0 then go E

This is the current pixel

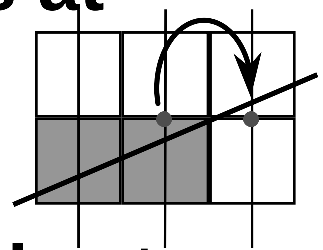
Midpoint line drawing algorithm 2

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

$$d = a(x+1) + b(y+½) + c$$

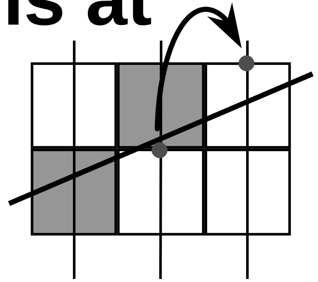
- if go E then the new decision variable is at $(x+2, y+½)$

$$d' = a(x+2) + b(y+½) + c$$
$$= d + a$$



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

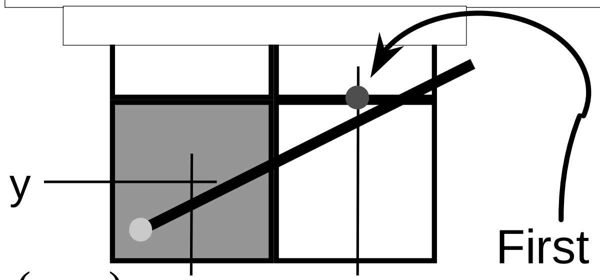
$$d' = a(x+2) + b(y+1½) + c$$
$$= d + a + b$$



Midpoint line drawing algorithm 3

Initialisation

a = $(y_1 - y_0)$
b = $-(x_1 - x_0)$
c = $x_1 y_0 - x_0 y_1$
x = ROUND(x_0)
y = ROUND(y_0-(x- x_0)(a / b))
d = a * (x+1) + b * (y+½) + c
DRAW(x,y)



Iteration

WHILE x < $(x_1$ - ½) DO
x = x + 1
IF d < 0 THEN
d = d + a
ELSE
d = d + a + b
y = y + 1
END IF
DRAW(x,y)
END WHILE

E case
just increment x

NE case
increment x & y

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

Midpoint - comments

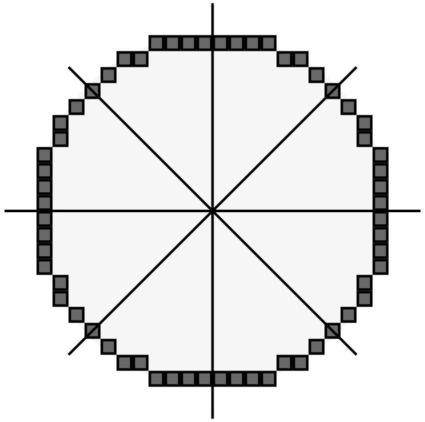
- this version only works for lines in the first octant
  - extend to other octants as for Bresenham
- Sproull has proven that Bresenham and Midpoint give identical results
- Midpoint algorithm can be generalised to draw arbitary circles & ellipses
  - Bresenham can only be generalised to draw circles with integer radii

Curves

- circles & ellipses
- Bezier cubics
  - Pierre Bézier, worked in CAD for Citroën
  - widely used in Graphic Design
- Overhauser cubics
  - Overhauser, worked in CAD for Ford
- NURBS
  - Non-Uniform Rational B-Splines
  - more powerful than Bezier & now more widely used
  - consider these next year

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**

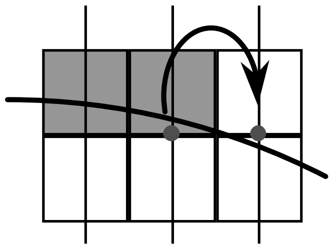


Midpoint circle algorithm 2

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

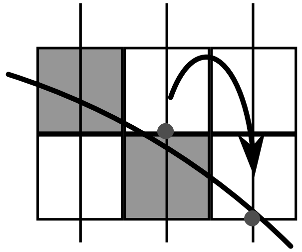
$$d = (x+1)^2 + (y - ½)^2 - r^2$$

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



$$d' = (x+2)^2 + (y - ½)^2 - r^2$$
$$= d + 2x + 3$$

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

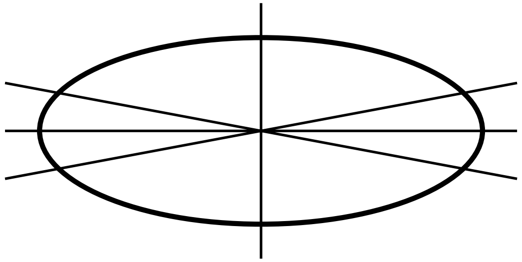


$$d' = (x+2)^2 + (y - 1½)^2 - r^2$$
$$= d + 2x - 2y + 5$$

Exercise: complete the circle algorithm for the second octant

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

Are circles & ellipses enough?

- simple drawing packages use ellipses & segments of ellipses
- for graphic design & CAD need something with more flexibility
  - use cubic polynomials

Why cubics?

- lower orders cannot:
  - have a point of inflection
  - match both position and slope at both ends of a segment
  - be non-planar in 3D
- higher orders:
  - can wiggle too much
  - take longer to compute

Hermite cubic

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

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

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

Charles Hermite, mathematician, 1822–1901

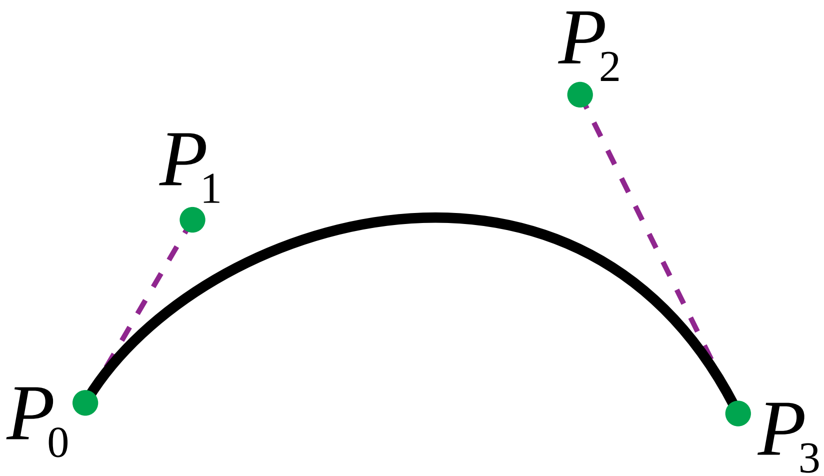
Bezier cubic

- difficult to think in terms of tangent vectors

✦ **Bezier 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: $P_i \equiv (x_i, y_i)$



Pierre Bézier worked for Citroën in the 1960s

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

Types of curve join

- each curve is smooth within itself
- joins at endpoints can be:
  - C_0 – continuous in position, tangent vectors have different directions
    - “corner”
  - C_1 – continuous in both position and tangent vector
    - smooth join
  - G_1 – continuous in postion, tangent vectors have same direction but not same magnitude
  - discontinuous in position

Drawing a Bezier cubic – naïve method

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

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

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

Drawing a Bezier cubic – sensible method

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

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

how do we do this? see the next slide…

Subdividing a Bezier cubic into two halves

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

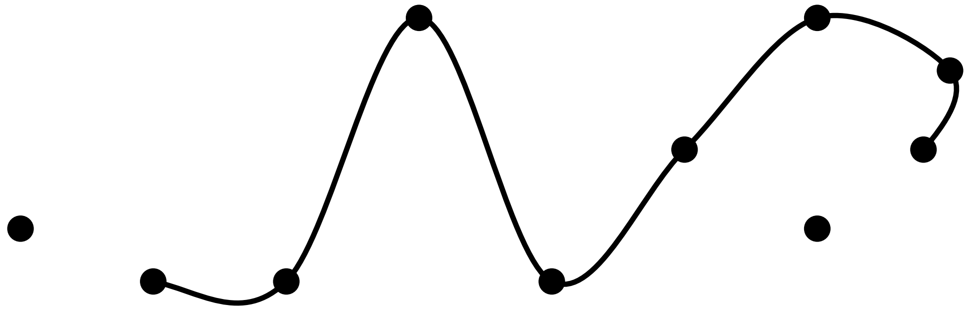
$$Q_0 = P_0$$
$$Q_1 = \tfrac{1}{2}P_0 + \tfrac{1}{2}P_1$$
$$Q_2 = \tfrac{1}{4}P_0 + \tfrac{1}{2}P_1 + \tfrac{1}{4}P_2$$
$$Q_3 = \tfrac{1}{8}P_0 + \tfrac{3}{8}P_1 + \tfrac{3}{8}P_2 + \tfrac{1}{8}P_3$$

$$R_0 = \tfrac{1}{8}P_0 + \tfrac{3}{8}P_1 + \tfrac{3}{8}P_2 + \tfrac{1}{8}P_3$$
$$R_1 = \tfrac{1}{4}P_1 + \tfrac{1}{2}P_2 + \tfrac{1}{4}P_3$$
$$R_2 = \tfrac{1}{2}P_2 + \tfrac{1}{2}P_3$$
$$R_3 = P_3$$

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

What if we have no tangent vectors?

- base each cubic piece on the four surrounding data points



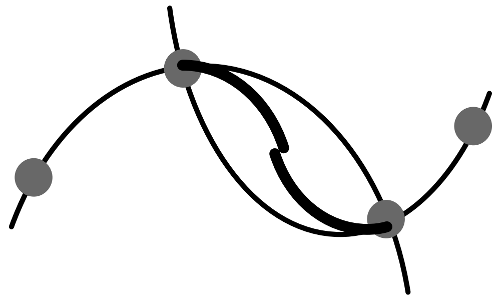
- at each data point the curve must depend solely on the three surrounding data points

Why?

  - there is a unique quadratic passing through any three points
  - use this to define the tangent at each 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

Overhauser's cubic

- **method**
  - **linearly interpolate between two quadratics**



$$P(t)=(1-t)Q(t)+tR(t)$$

- **(potential) problem**
  - **moving a single point modifies the surrounding four curve segments** (c.f. Bezier where moving a single point modifies just the two segments connected to that point)
- **good for control of movement in animation**

Overhauser worked for the Ford motor company in the 1960s

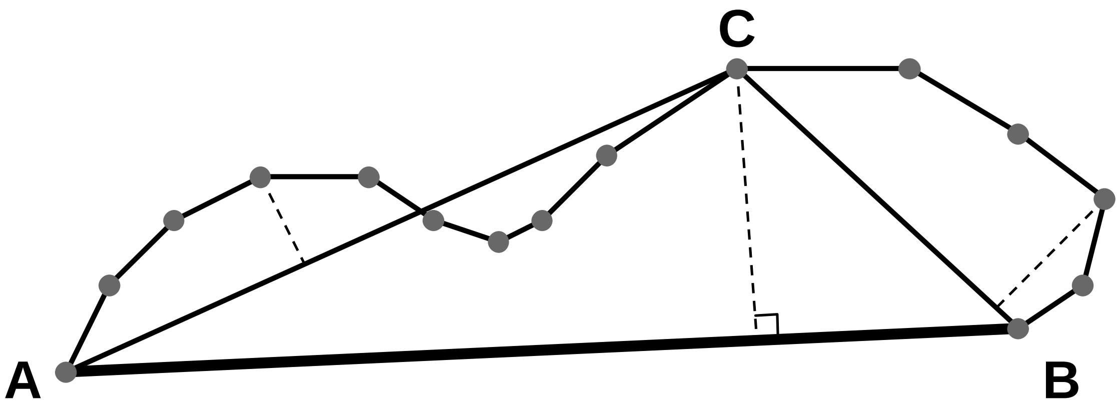
Simplifying line chains

- **the problem: you are given a chain of line segments at a very high resolution, how can you reduce the number of line segments without compromising the quality of the line**
  - **e.g. given the coastline of Britain defined as a chain of line segments at 10m resolution, draw the entire outline on a 1280$\times$1024 pixel screen**
- **the solution: Douglas & Pücker’s line chain simplification algorithm**

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

Douglas & Pücker’s algorithm

- find point, C, at greatest distance from line 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

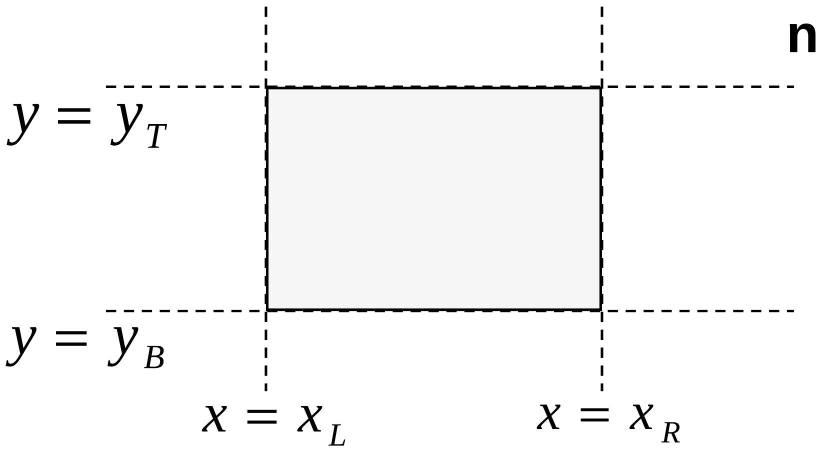


Exercise: what special cases need to be considered?
how should they be handled?

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

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 four inequalities:

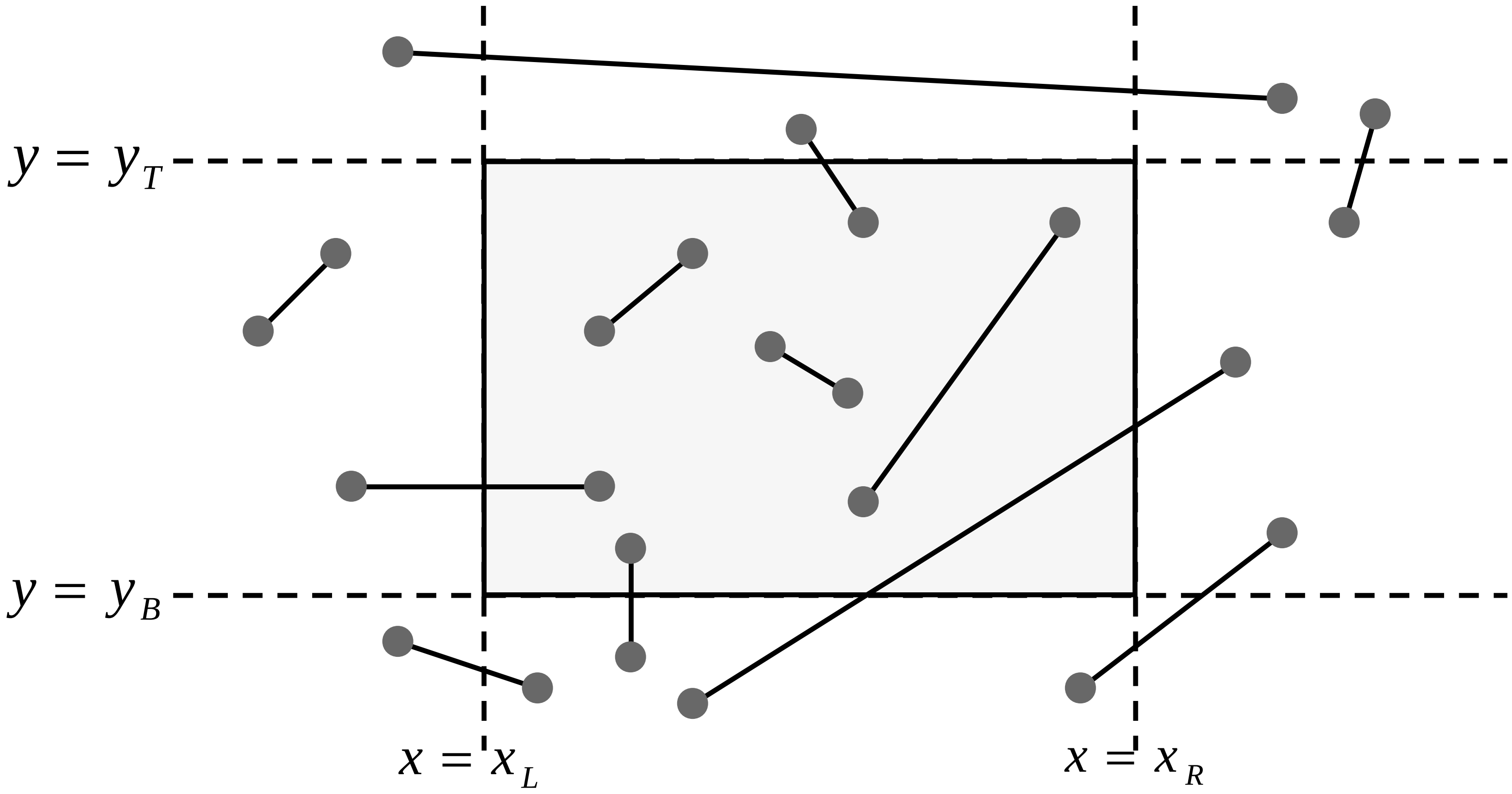
$$x \geq x_L$$

$$x \leq x_R$$

$$y \geq y_B$$

$$y \leq y_T$$

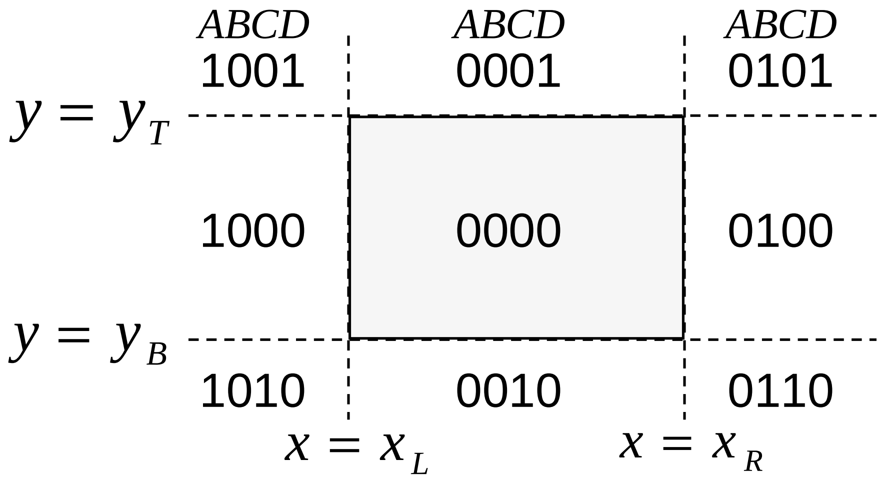
Clipping lines against a rectangle



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**

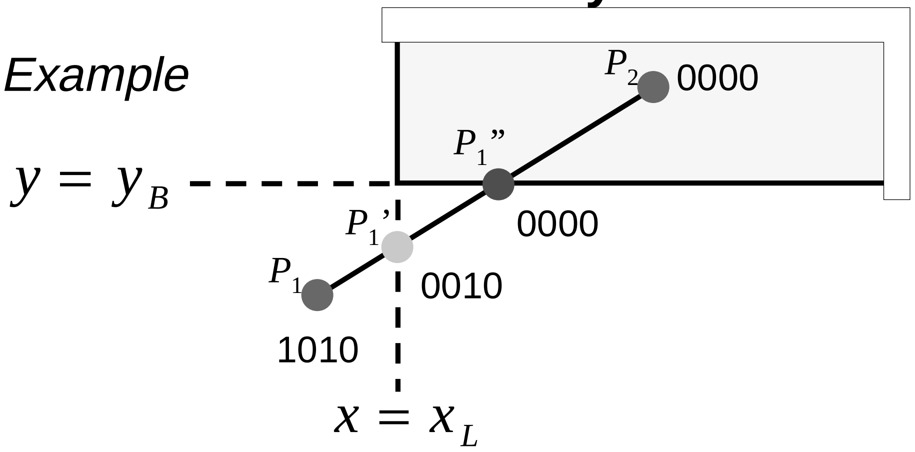
$$Q_1 = A_1 B_1 C_1 D_1 \quad Q_2 = A_2 B_2 C_2 D_2$$

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

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**
  - **the 1 bits tell you which edge to clip against**

Example



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

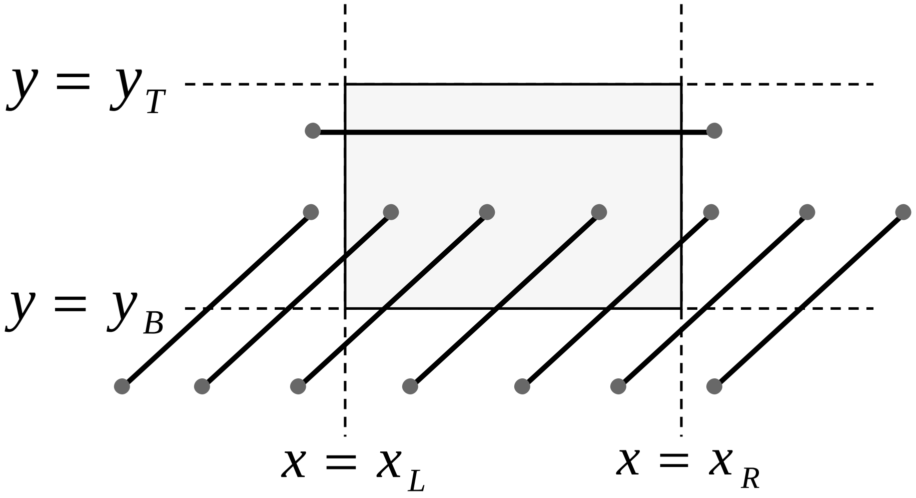
$$y_1'' = y_B \qquad x_1'' = x_1' + (x_2 - x_1')\frac{y_B - y_1'}{y_2 - y_1'}$$

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
- need a line drawing algorithm that can cope with floating-point endpoint co-ordinates

Why not?

Why?

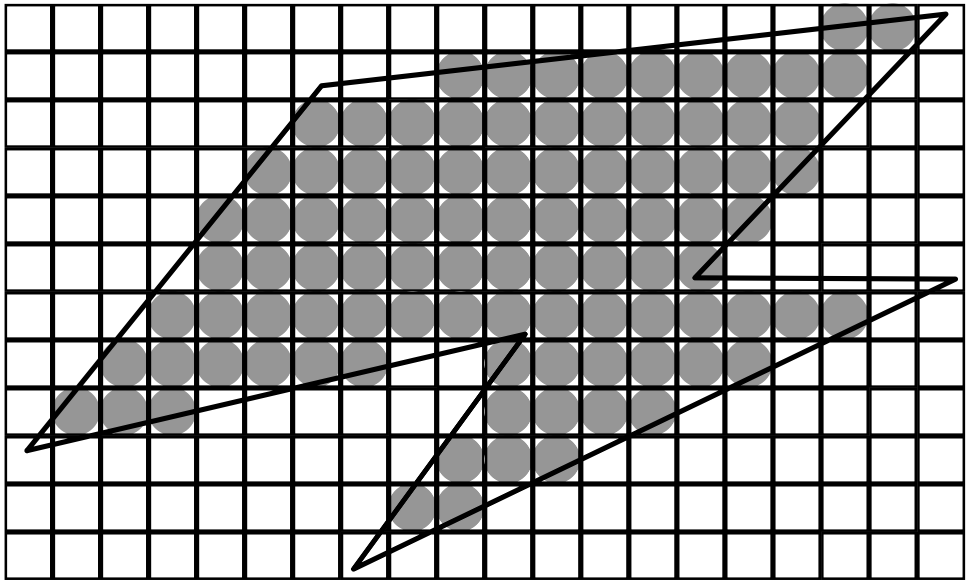


Exercise: what happens in each of the cases at left?
[Assume that, where there is a choice, the algorithm always trys to intersect with x_L or x_R before y_B or y_T.]

Try some other cases of your own devising.

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

Scanline polygon fill algorithm

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

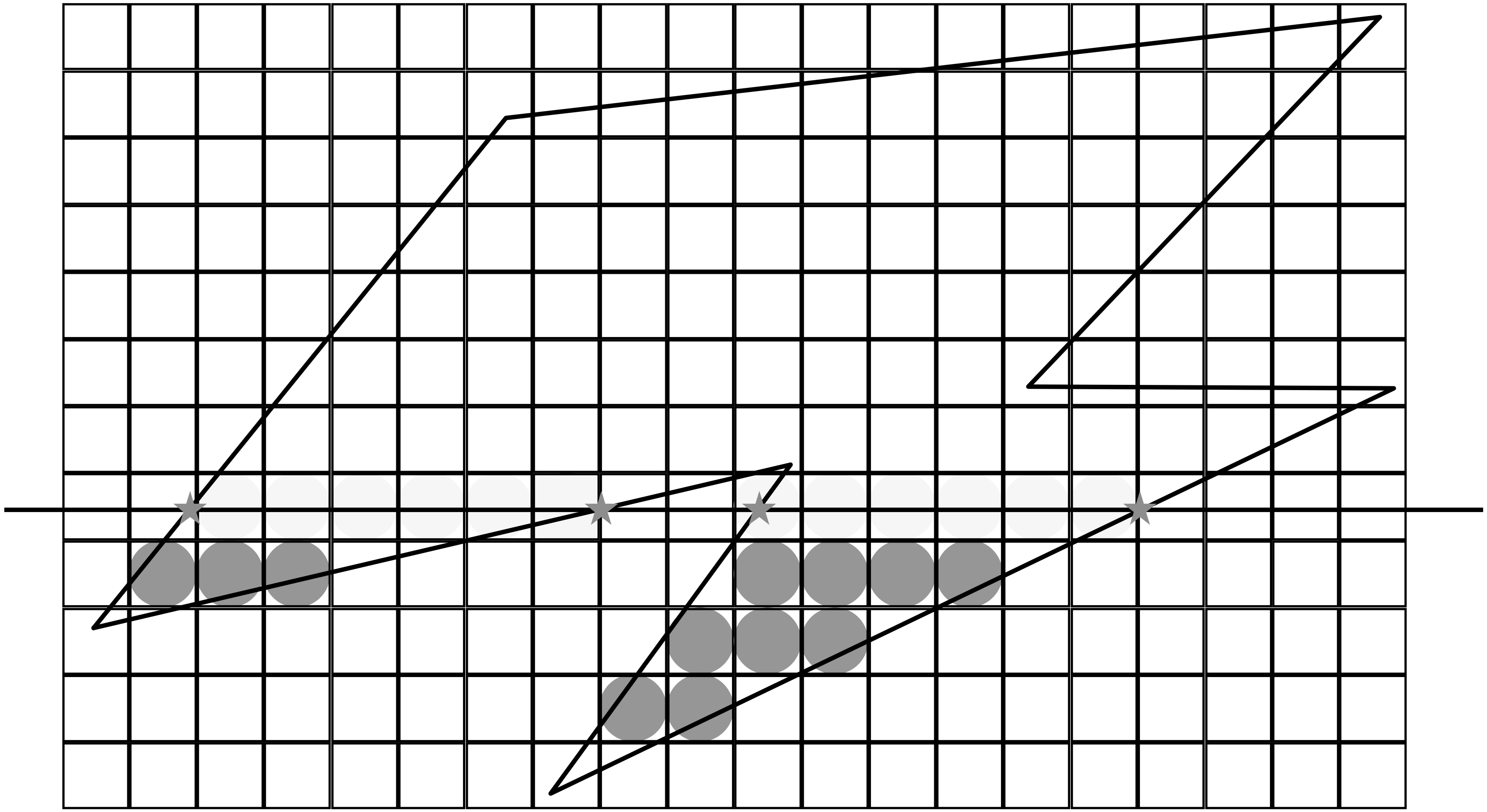
❷ start with the first scanline that intersects the polygon, get all edges which intersect that scan line and move them to an *active edge list* (*AEL*)

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

❹ fill between pairs of intersection points

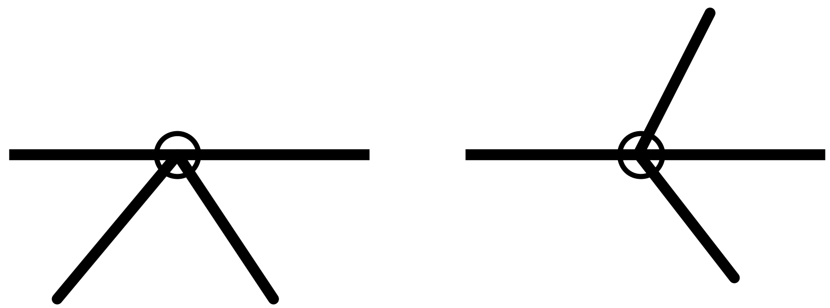
❺ move to the next scanline (increment y); remove edges from the AEL if endpoint $< y$; move new edges from EL to AEL if start point $\leq y$; if any edges remain in the AEL go back to step ❸

Scanline polygon fill example

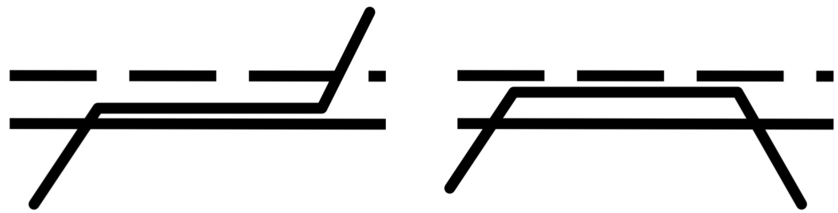


Scanline polygon fill details

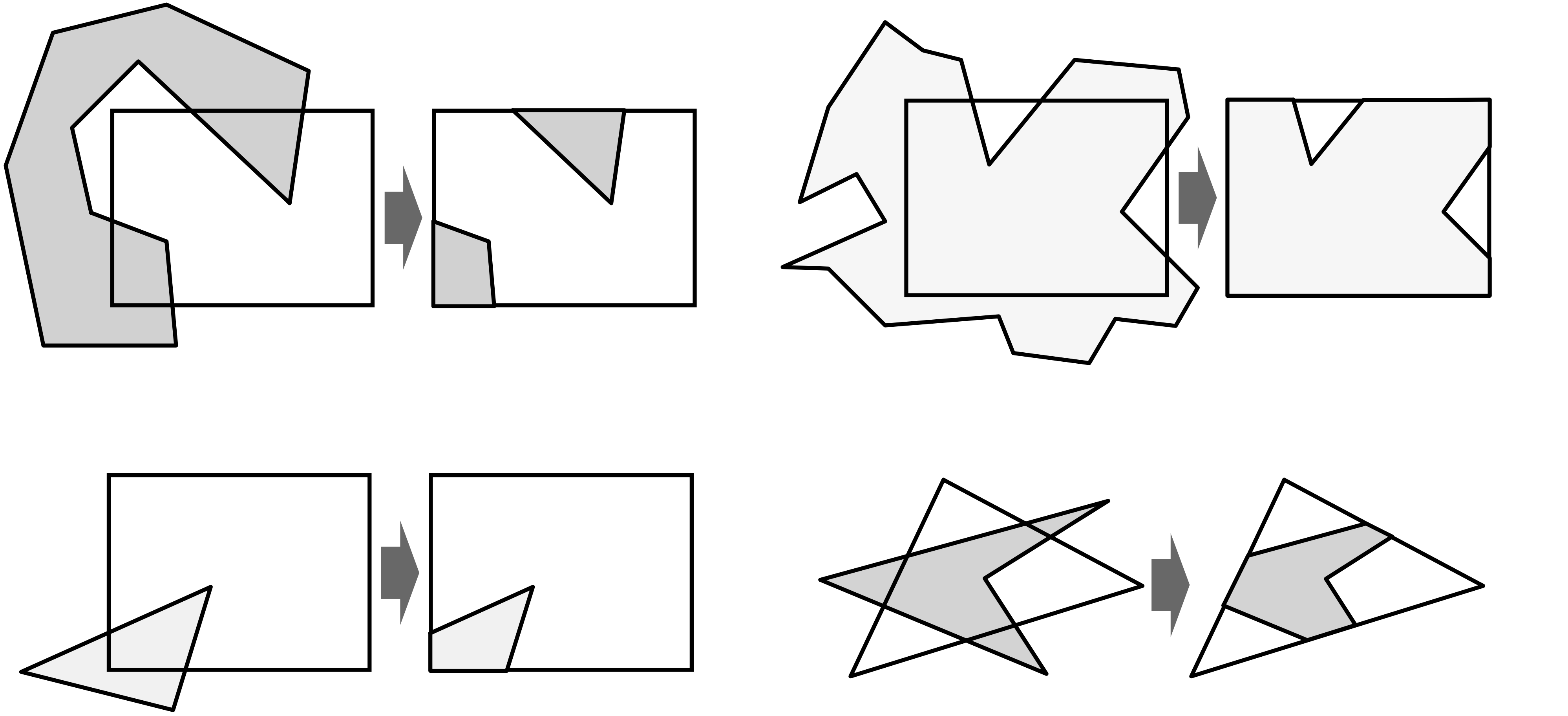
- **how do we efficiently calculate the intersection points?**
  - **use a line drawing algorithm to do incremental calculation**
- **what if endpoints exactly intersect scanlines?**
  - **need to cope with this, e.g. add a tiny amount to the y co-ordinate to ensure that they don't exactly match**



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

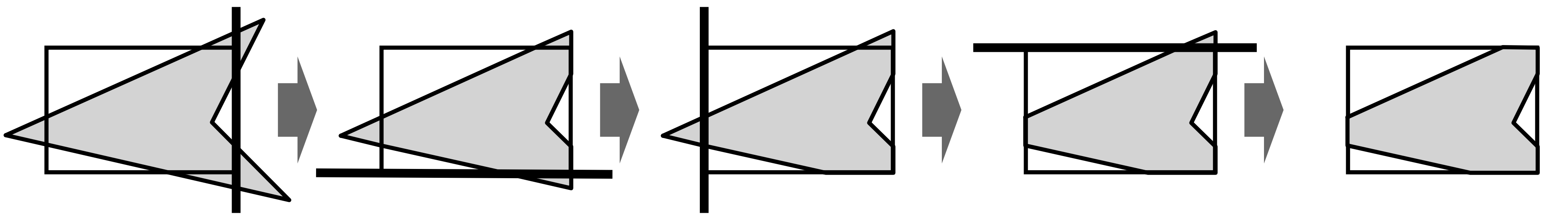


Clipping polygons



Sutherland-Hodgman polygon clipping 1

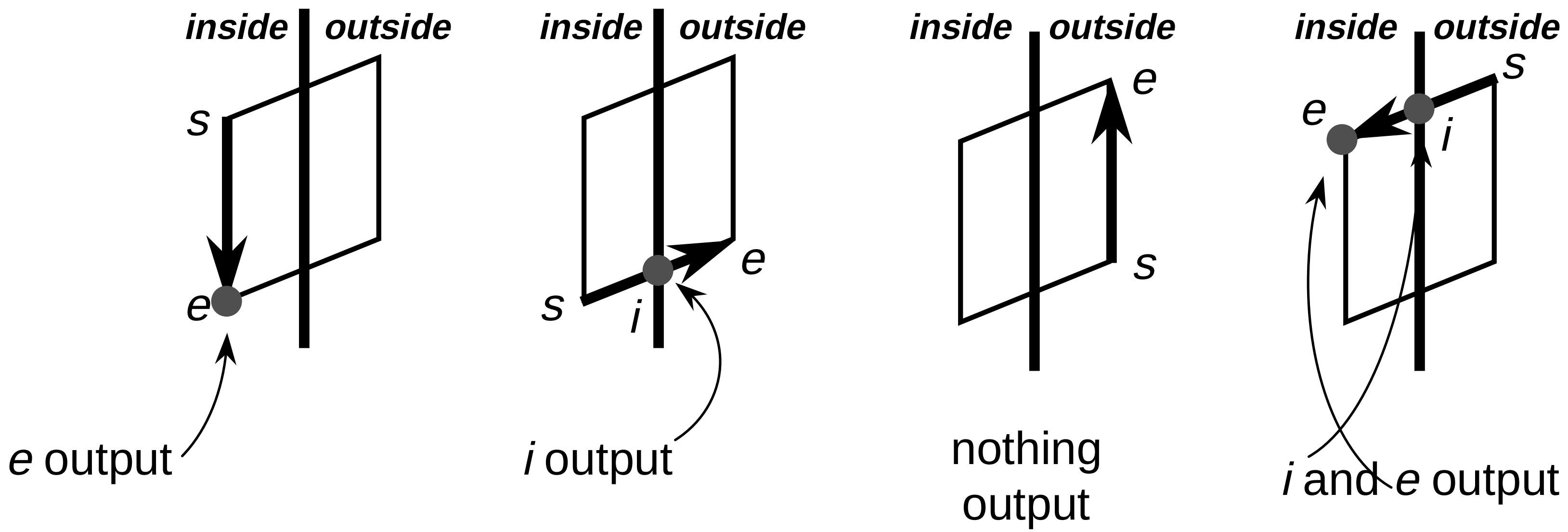
- clips an arbitrary polygon against an arbitrary *convex* polygon
  - basic algorithm clips an arbitrary polygon against a single infinite clip edge
  - the polygon is clipped against one edge at a time, passing the result on to the next stage



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

Sutherland-Hodgman polygon clipping 2

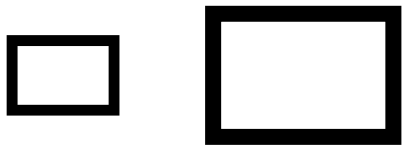
- **the algorithm progresses around the polygon checking if each edge crosses the clipping line and outputting the appropriate points**



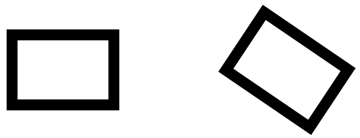
Exercise: the Sutherland-Hodgman algorithm may introduce new edges along the edge of the clipping polygon — when does this happen and why?

2D transformations

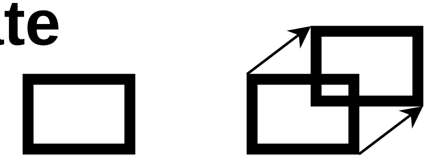
- scale



- 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

Basic 2D transformations

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

$$x' = mx$$
$$y' = my$$

- **rotate**
  - **about origin**
  - **by angle** θ

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

- **translate**
  - **along vector** (x_o, y_o)

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

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

$$x' = x + ay$$
$$y' = y$$

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

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

Homogeneous 2D co-ordinates

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

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

- 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:

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

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

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

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

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

Concatenating transformations

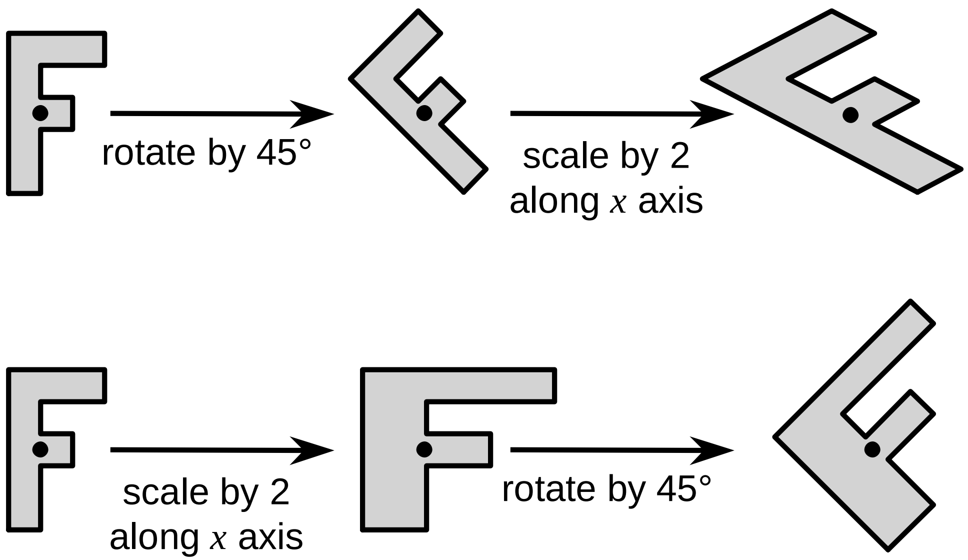
- often necessary to perform more than one transformation on the same object
- can concatenate transformations by multiplying their matrices
  e.g. a shear followed by a scaling:

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

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

Concatenation is not commutative

- be careful of the order in which you concatenate transformations



rotate then scale

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

scale

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

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

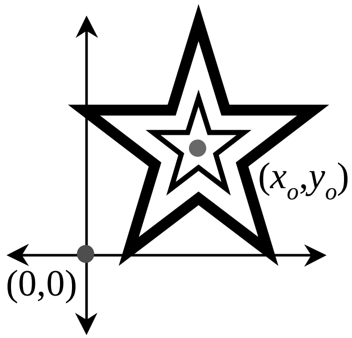
scale then rotate

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

rotate

Scaling about an arbitrary point

- **scale by a factor** m **about point** (x_o,y_o)
  - ❶**translate point** (x_o,y_o) **to the origin**
  - ❷**scale by a factor** m **about the origin**
  - ❸**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 $$\begin{bmatrix} x'' \\ y'' \\ w'' \end{bmatrix} = \begin{bmatrix} m & 0 & 0 \\ 0 & m & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x' \\ y' \\ w' \end{bmatrix}$$

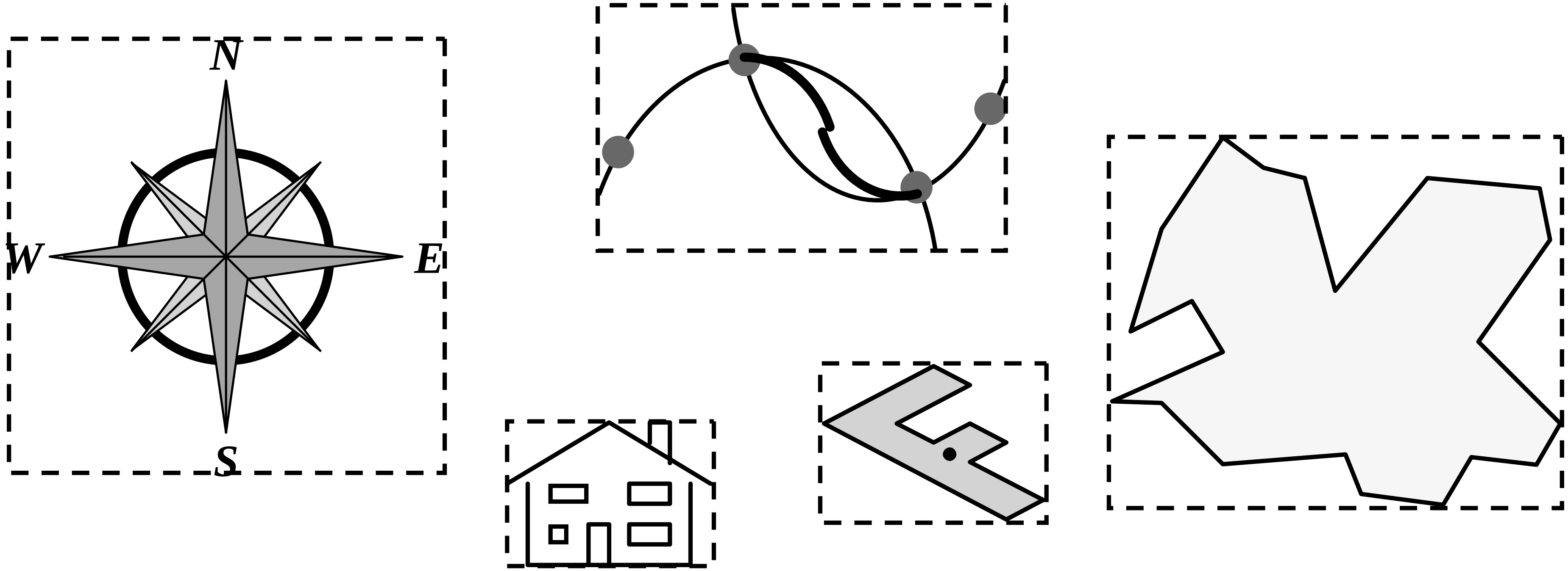
❸ $$\begin{bmatrix} x''' \\ y''' \\ w''' \end{bmatrix} = \begin{bmatrix} 1 & 0 & x_o \\ 0 & 1 & y_o \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x'' \\ y'' \\ w'' \end{bmatrix}$$

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

Exercise: show how to perform rotation about an arbitrary point

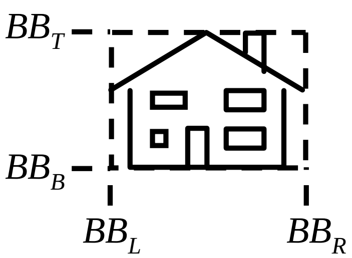
Bounding boxes

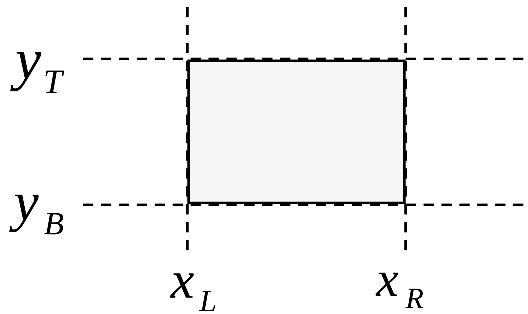
- when working with complex objects, bounding boxes can be used to speed up some operations

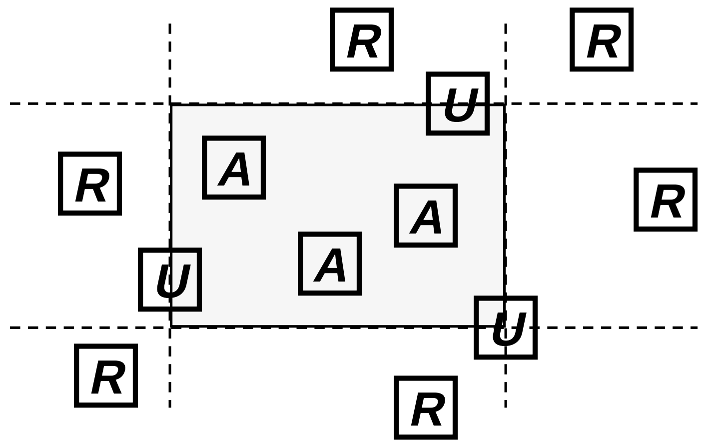


Clipping with bounding boxes

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







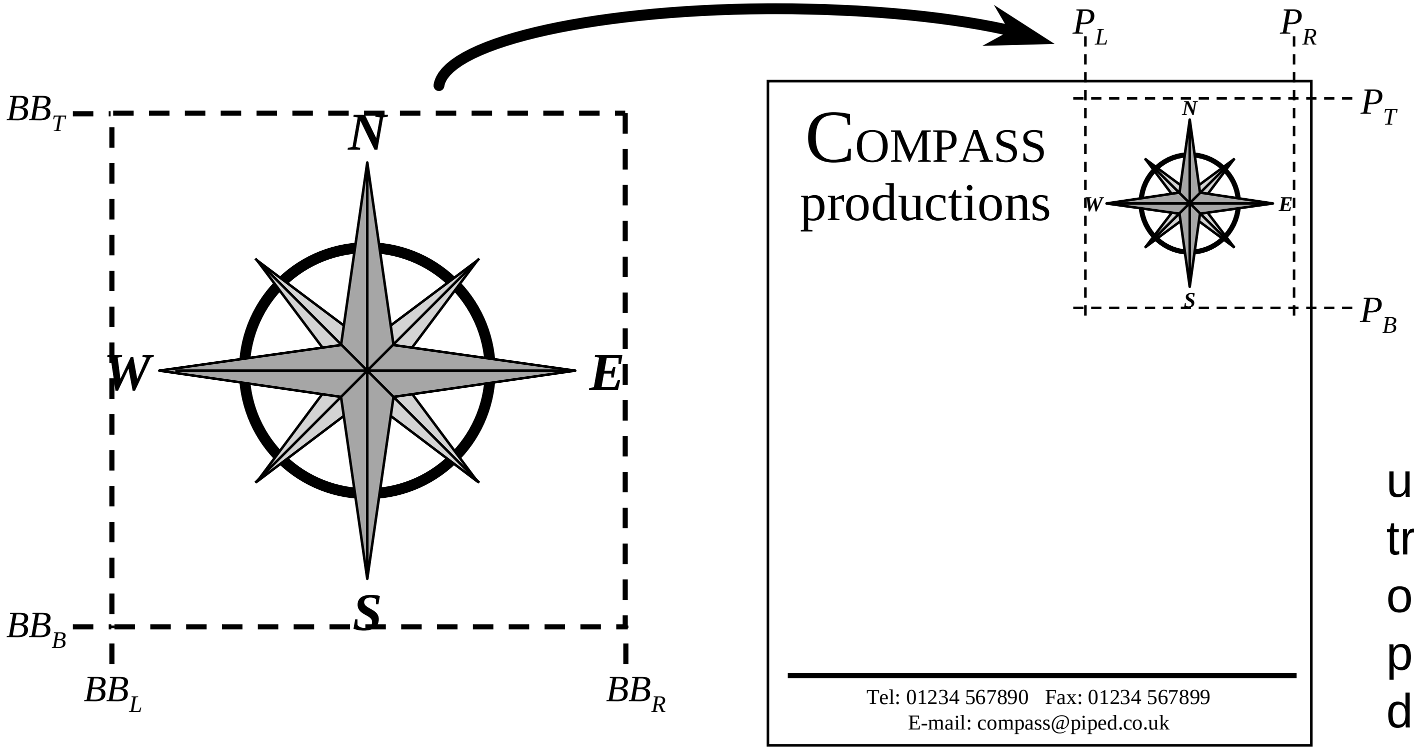
$$BB_L > x_R \vee BB_R < x_L \vee BB_B > x_T \vee BB_T < x_B \quad \Rightarrow \textit{REJECT}$$

$$BB_L \geq x_L \wedge BB_R \leq x_R \wedge BB_B \geq x_B \wedge BB_T \leq x_T \quad \Rightarrow \textit{ACCEPT}$$

otherwise $\Rightarrow$ clip at next higher level of detal

Object inclusion with bounding boxes

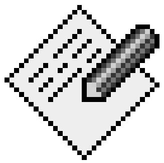
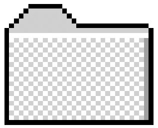
- **including one object (e.g. a graphics) file inside another can be easily done if bounding boxes are known and used**

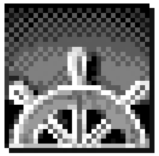
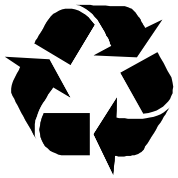


use the eight values to translate and scale the original to the appropriate position in the destination document

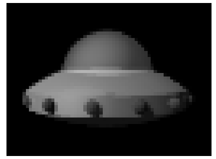
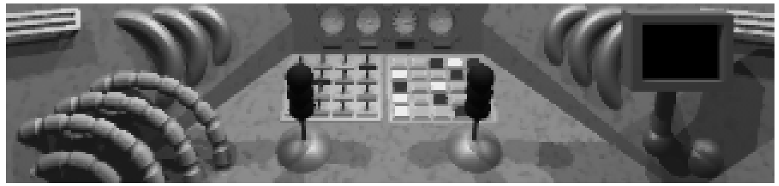
Bit block transfer (*BitBlT*)

- 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

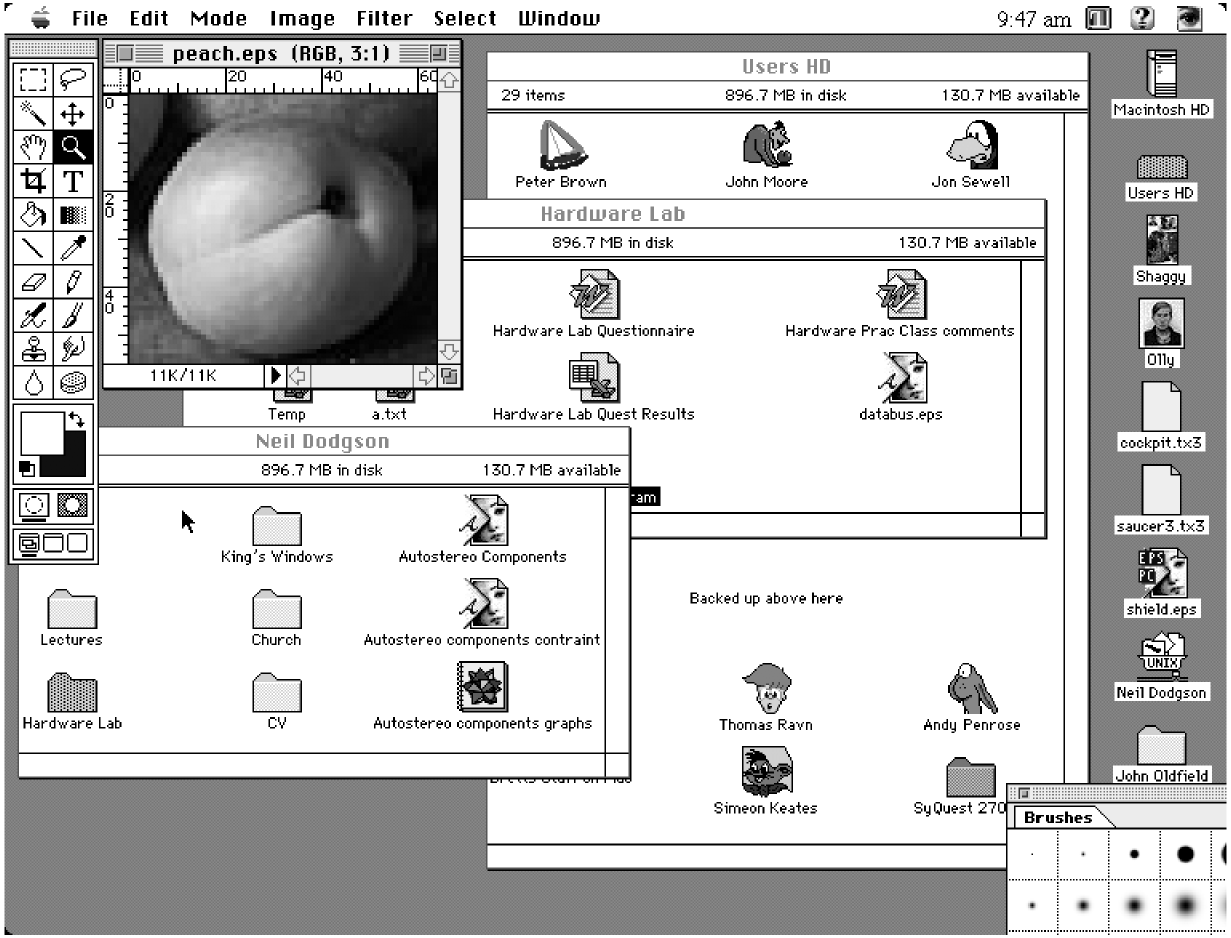
- copying an image from place to place is essentially a memory operation
  - can be made very fast
  - e.g. 32×32 pixel icon can be copied, say, 8 adjacent pixels at a time, if there is an appropriate memory copy operation

XOR drawing

- generally we draw objects in the appropriate colours, overwriting what was already there
- sometimes, usually in HCI, we want to draw something temporarily, with the intention of wiping it out (almost) immediately e.g. when drawing a rubber-band line
- if we bitwise XOR the object's colour with the colour already in the frame buffer we will draw an object of the correct shape (but wrong colour)
- if we do this twice we will restore the original frame buffer
- saves drawing the whole screen twice

Application 1: user interface

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

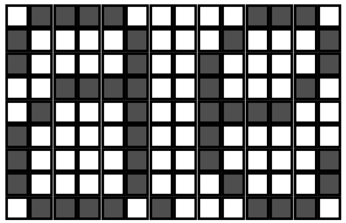


Application 2: typography

- **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*** - Helvetica abcd *efgh* **ijkl** ***mnop*** - Times

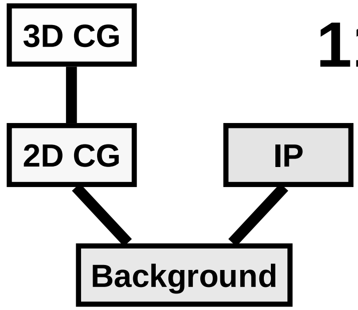
- **two forms of typeface used in computer graphics**
  - **pre-rendered bitmaps**
    - single resolution (don't scale well)
    - use BitBlT to put into frame buffer



  - **outline definitions**
    - multi-resolution (can scale)
    - need to render (fill) to put into frame buffer

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
  - halftoning
  - can define your own functions in the language

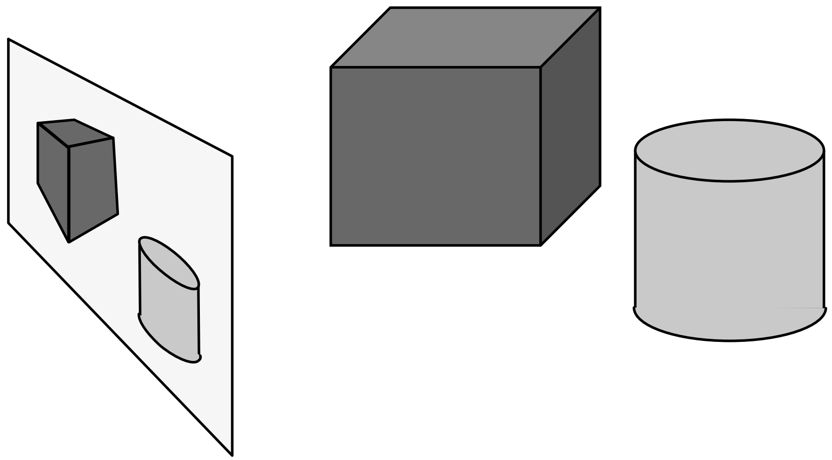


3D Computer Graphics

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

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



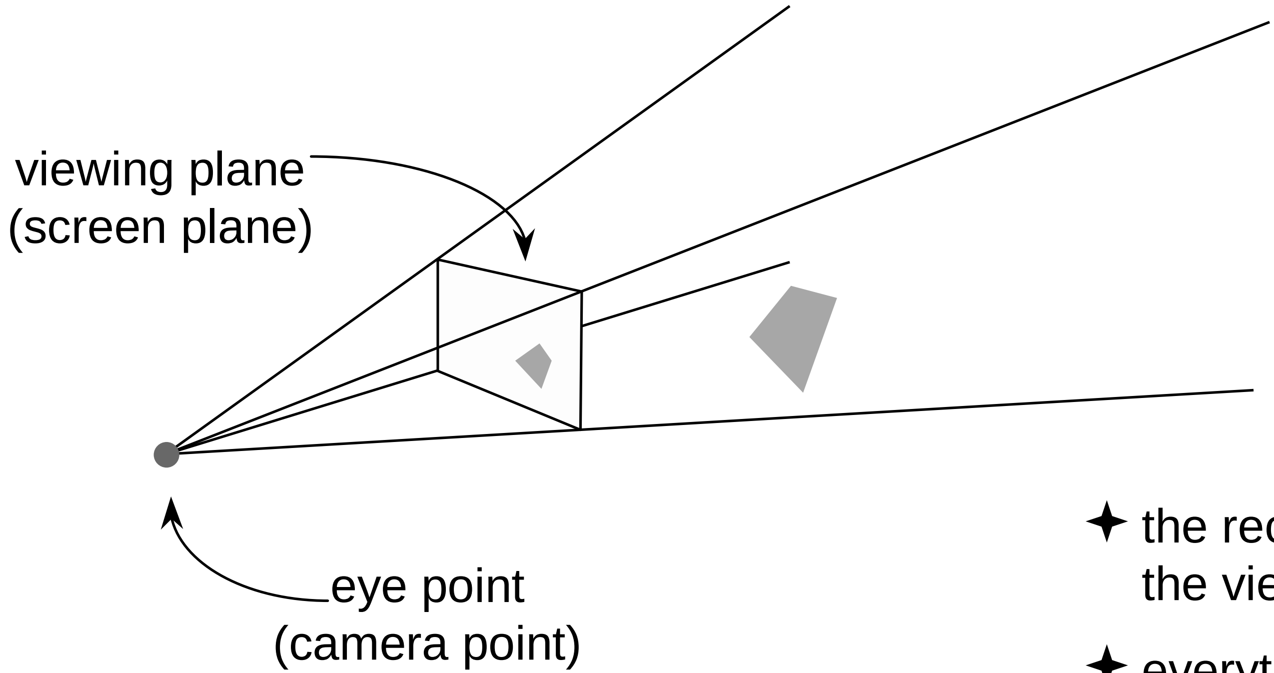
The 3D world is described as a set of (mathematical) objects

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

Types of projection

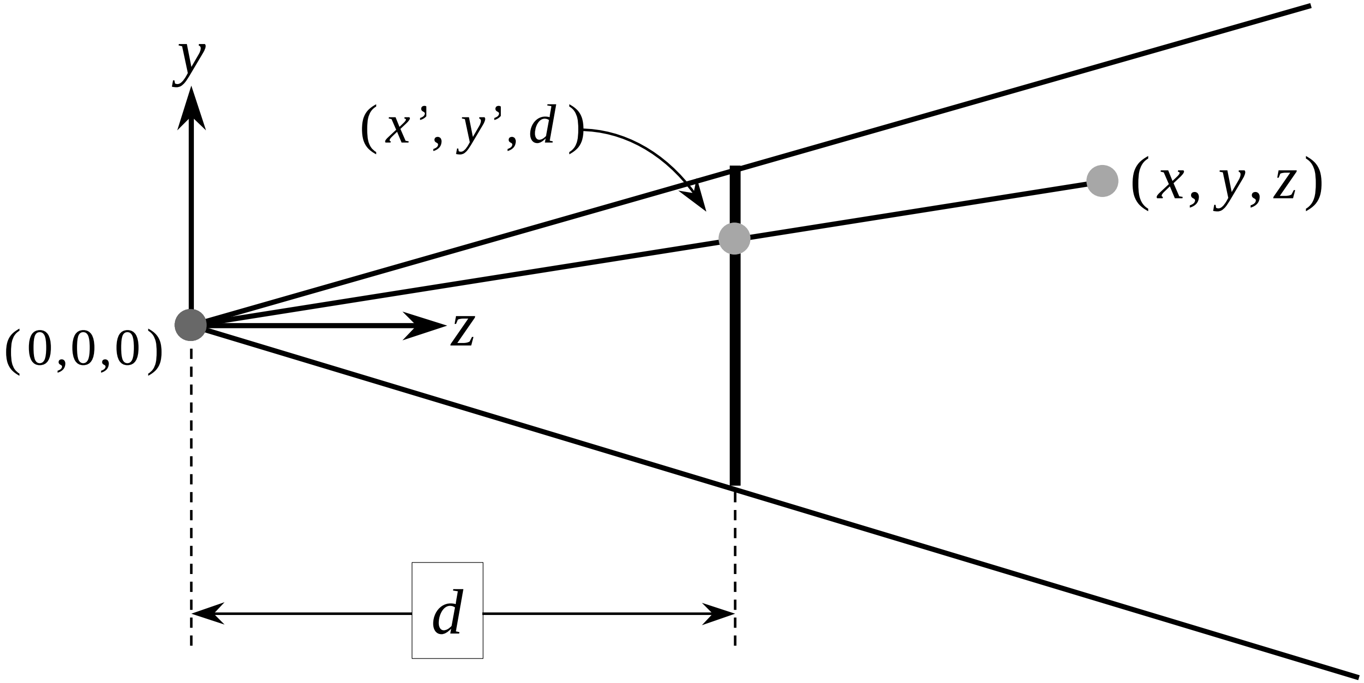
- parallel
  - e.g. $(x, y, z) \rightarrow (x, y)$
  - useful in CAD, architecture, etc
  - looks unrealistic
- perspective
  - e.g. $(x, y, z) \rightarrow (\frac{x}{z}, \frac{y}{z})$
  - things get smaller as they get farther away
  - looks realistic
    - this is how cameras work!

Viewing volume



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

Geometry of perspective projection



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

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

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**

3D transformations

- **3D homogeneous co-ordinates**

$$(x, y, z, w) \rightarrow (\tfrac{x}{w}, \tfrac{y}{w}, \tfrac{z}{w})$$

- **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

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

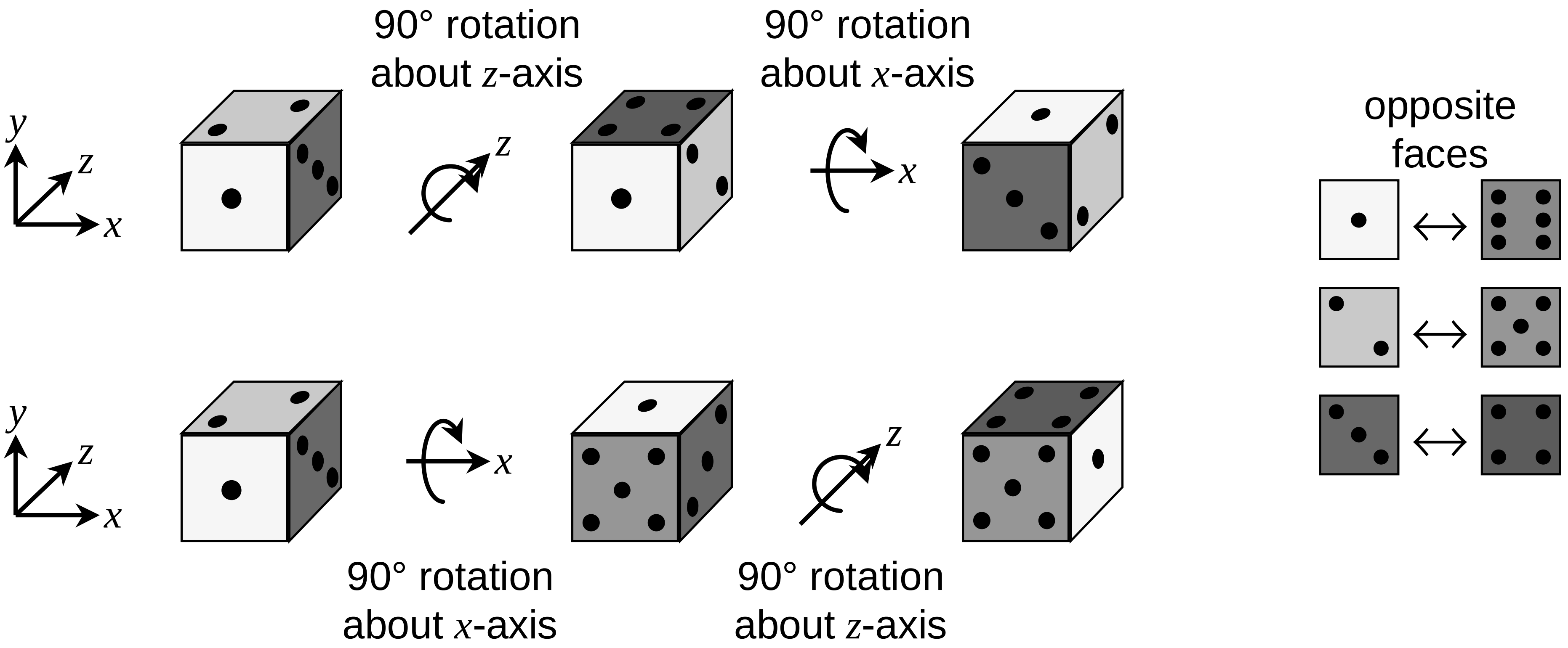
rotation about z-axis

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

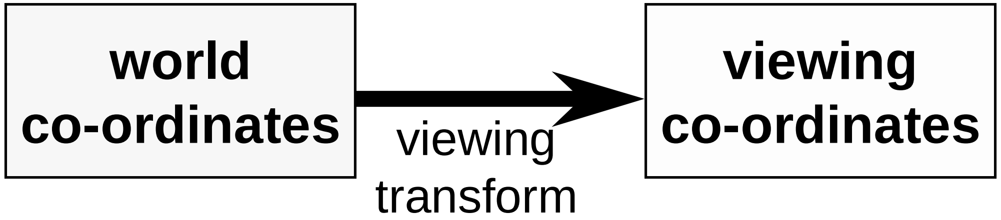
rotation about y-axis

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

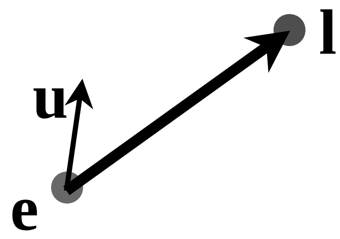
3D transformations are not commutative



Viewing transform 1



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



Viewing transform 2

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

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

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

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

Viewing transform 3

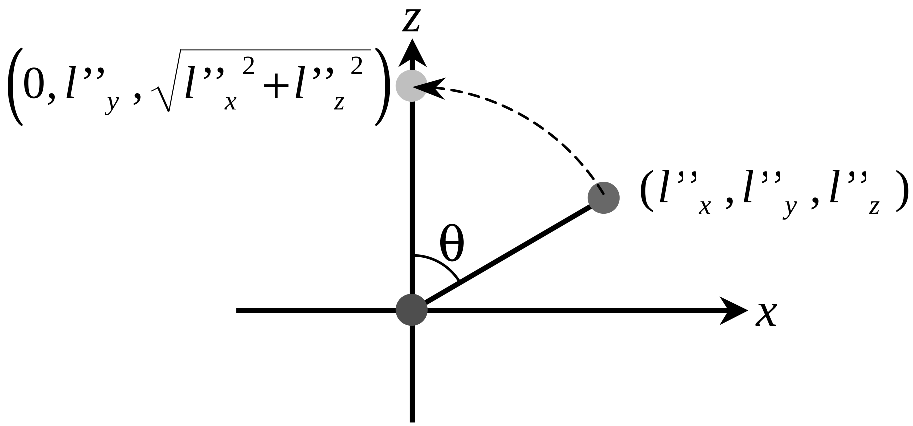
- need to align line $\overline{\mathbf{el}}$ with z-axis
  - first transform $\mathbf{e}$ and $\mathbf{l}$ into new co-ordinate system

$$\mathbf{e}'' = \mathbf{S} \times \mathbf{T} \times \mathbf{e} = \mathbf{0} \qquad \mathbf{l}'' = \mathbf{S} \times \mathbf{T} \times \mathbf{l}$$

  - then rotate $\mathbf{e}''\mathbf{l}''$ into yz-plane, rotating about y-axis

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

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



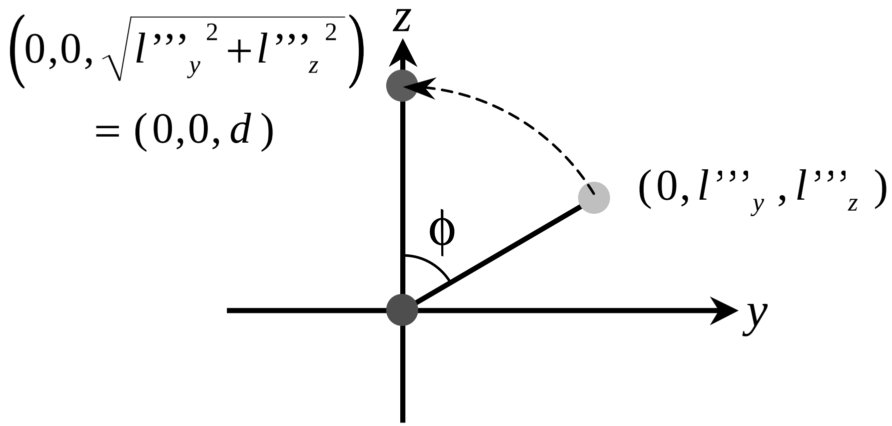
Viewing transform 4

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

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

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

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



Viewing transform 5

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

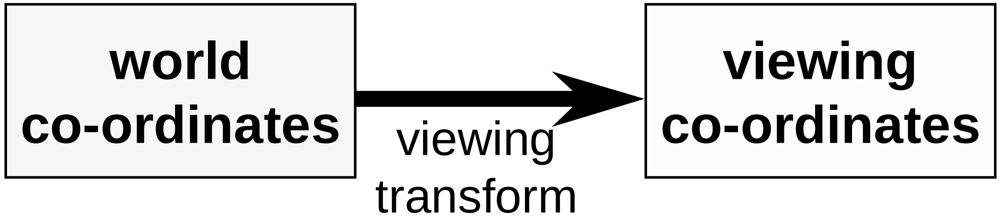
$$\mathbf{u}'''' = \mathbf{R}_2 \times \mathbf{R}_1 \times \mathbf{u}$$

why don't we need to multiply $\mathbf{u}$ by $\mathbf{S}$ or $\mathbf{T}$?

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

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

Viewing transform 6



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

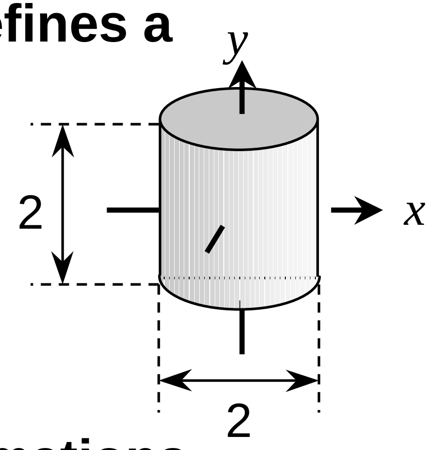
$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \mathbf{R}_3 \times \mathbf{R}_2 \times \mathbf{R}_1 \times \mathbf{S} \times \mathbf{T} \times \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

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

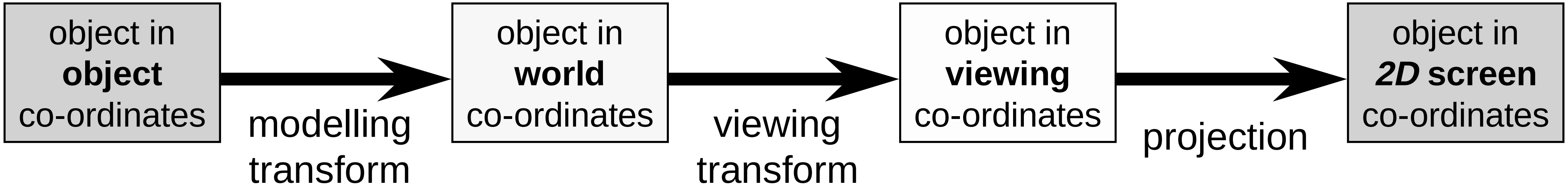
$$\mathbf{M} = \mathbf{R}_3 \times \mathbf{R}_2 \times \mathbf{R}_1 \times \mathbf{S} \times \mathbf{T}$$

Another transformation example

- a well known graphics package (Open Inventor) defines a cylinder to be:
  - centre at the origin, (0,0,0)
  - radius 1 unit
  - height 2 units, aligned along the *y*-axis
- this is the only cylinder that can be drawn, *but* the package has a complete set of 3D transformations
- we want to draw a cylinder of:
  - radius 2 units
  - the centres of its two ends located at (1,2,3) and (2,4,5)
    - ❖ its length is thus 3 units
- what transforms are required?
  and in what order should they be applied?



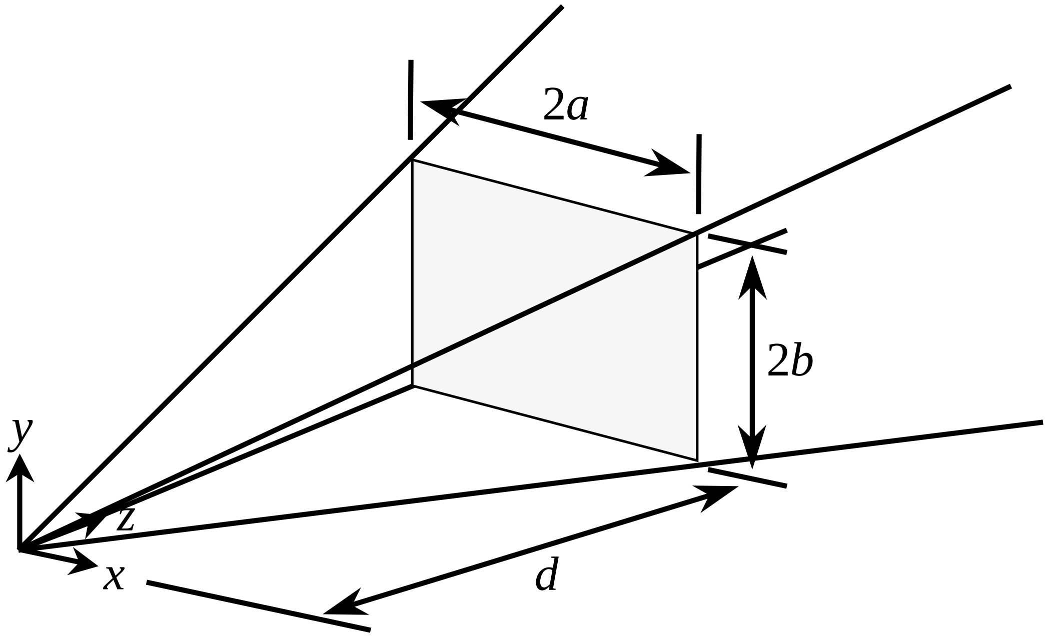
A variety of transformations



- **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**

Clipping in 3D

✦ **clipping against a volume in viewing co-ordinates**



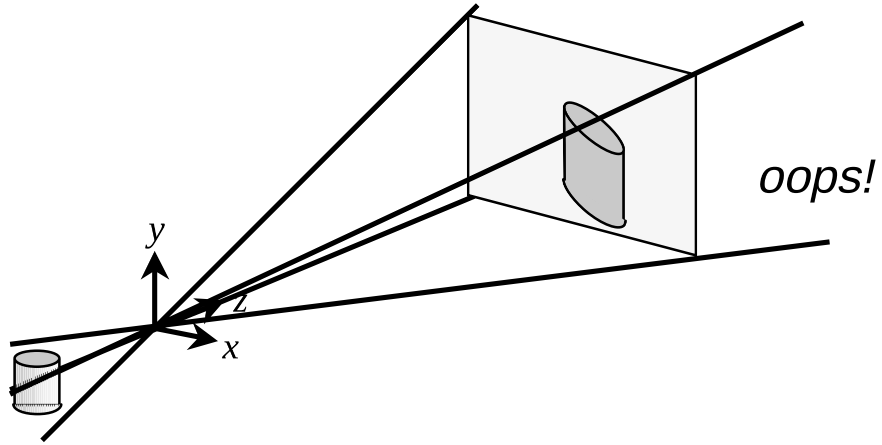
a point (x,y,z) can be clipped against the pyramid by checking it against four planes:

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

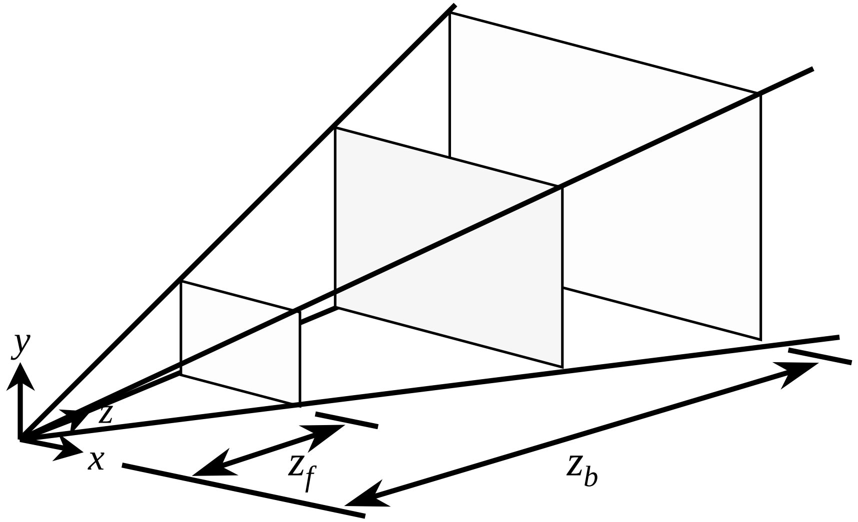
$$y > -z\frac{b}{d} \qquad y < z\frac{b}{d}$$

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*



Clipping in 3D — two methods

which is best?

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

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

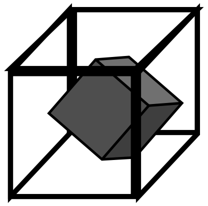
- **project to 2D (retaining z) and clip against the axis-aligned cuboid**
  - **still need to clip against six planes**

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

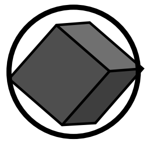
    - **these are simpler planes against which to clip**
    - **this is equivalent to clipping in 2D with two extra clips for z**

Bounding volumes & clipping

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



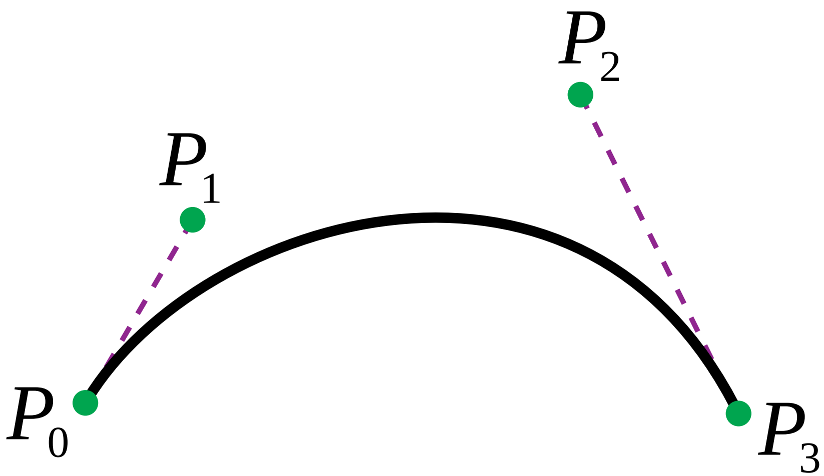
  - sphere



Curves in 3D

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

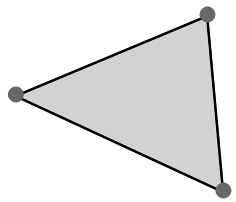
$$
\begin{aligned}
P(t) = {} & (1-t)^3 P_0 \\
& + 3t(1-t)^2 P_1 \\
& + 3t^2(1-t) P_2 \\
& + t^3 P_3
\end{aligned}
$$

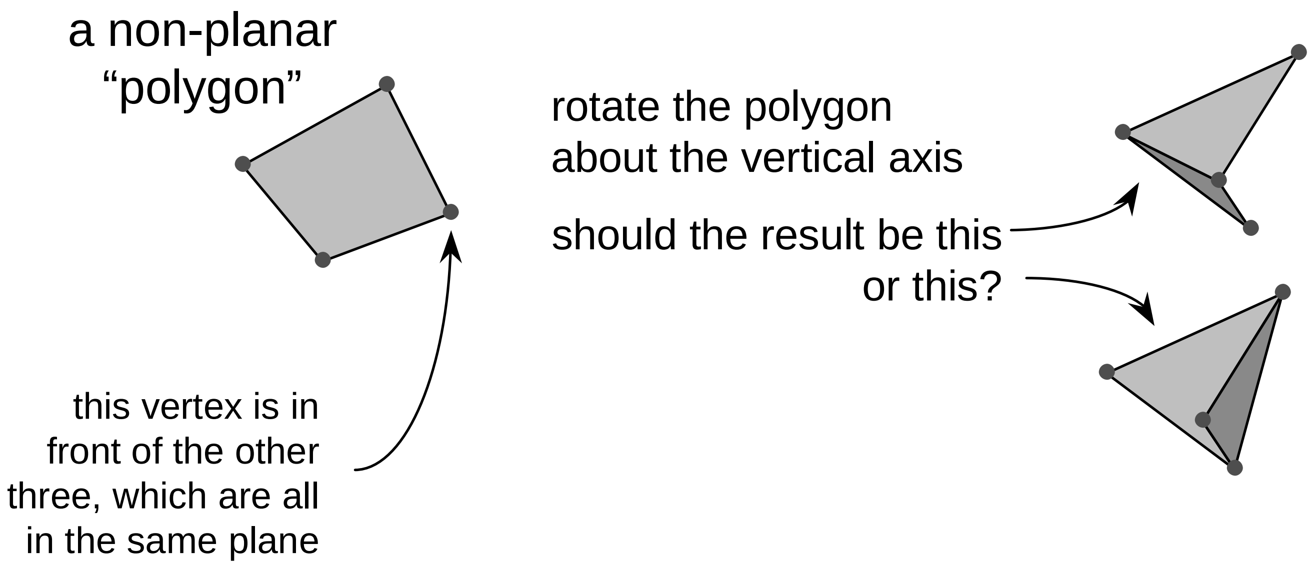


where: $P_i \equiv (x_i, y_i, z_i)$

Surfaces in 3D: polygons

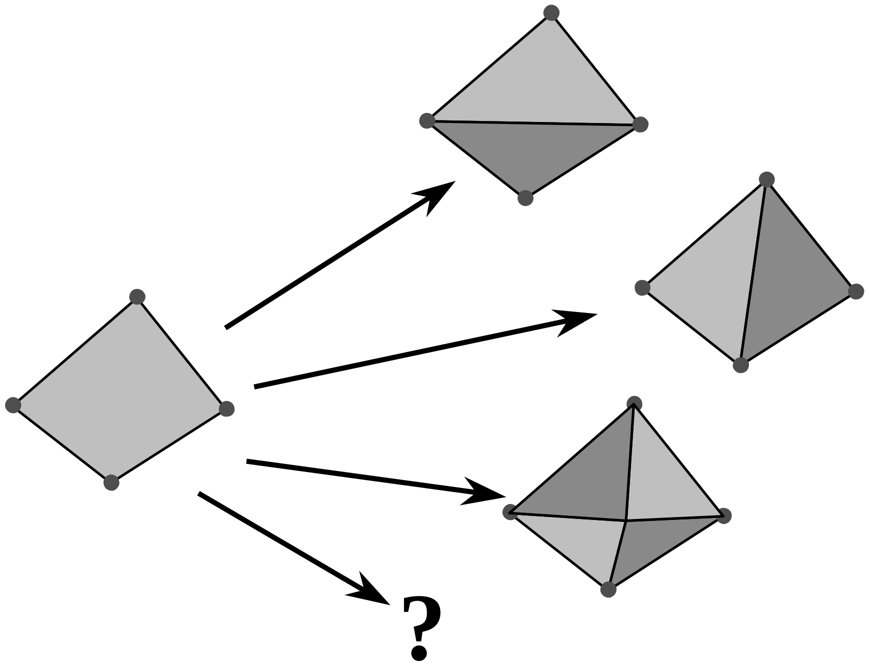
- **lines generalise to planar polygons**
  - **3 vertices (triangle) must be planar**
  - **> 3 vertices, not necessarily planar**





Splitting polygons into triangles

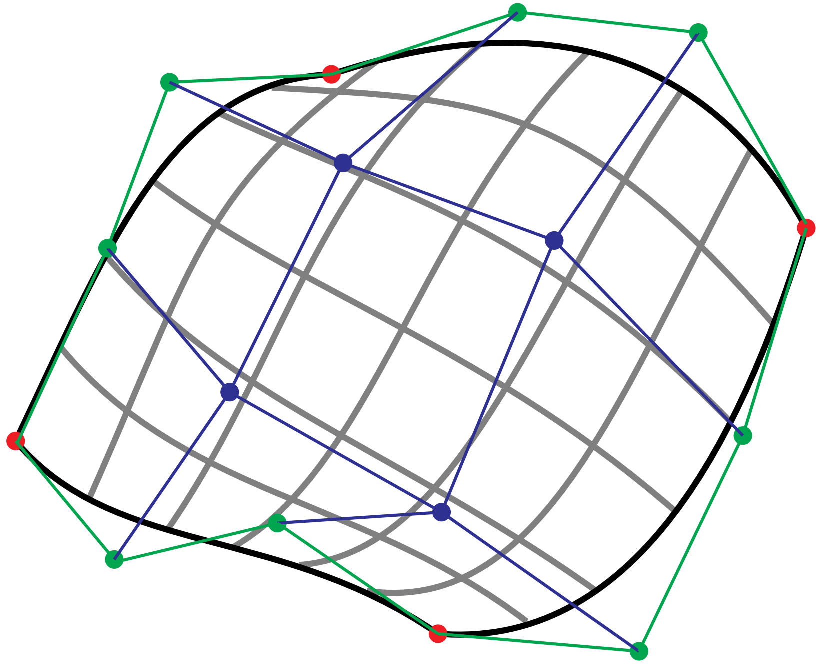
- some graphics processors accept only triangles
- an arbitrary polygon with more than three vertices isn't guaranteed to be planar; a triangle is



which is preferable?

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



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:**

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

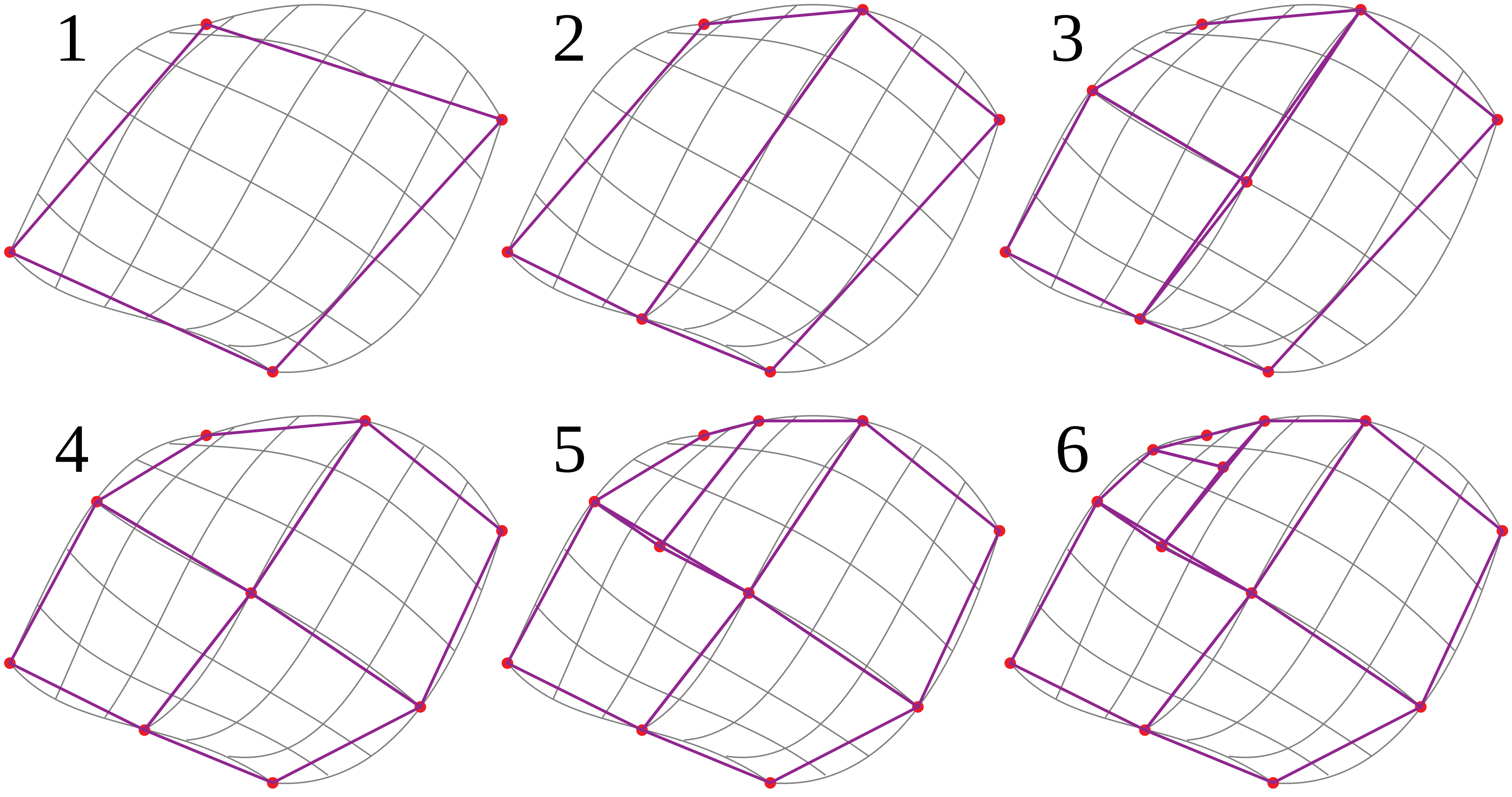
Continuity between Bezier patches

- each patch is smooth within itself
- ensuring continuity in 3D:
  - C_0 – continuous in position
    - the four edge control points must match
  - C_1 – continuous in both position and tangent vector
    - the four edge control points must match
    - the two control points on either side of each of the four edge control points must be co-linear with both the edge point and each another and be equidistant from the edge point
  - G_1
    - slightly less rigorous than C_1

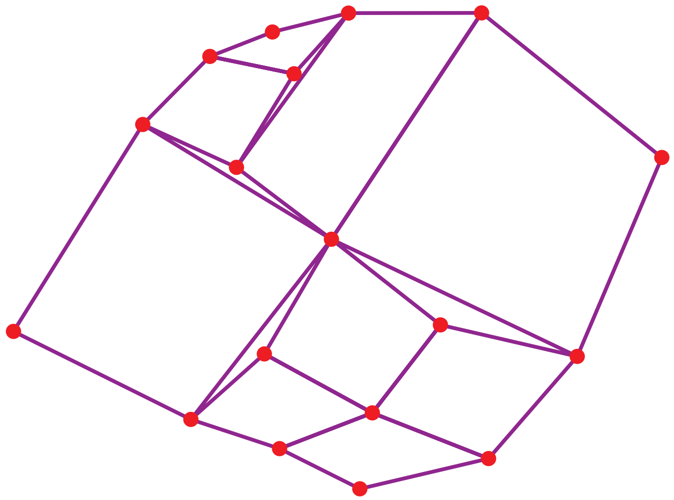
Drawing Bezier patches

- in a similar fashion to Bezier curves, Bezier patches can be drawn by approximating them with planar polygons
- 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!

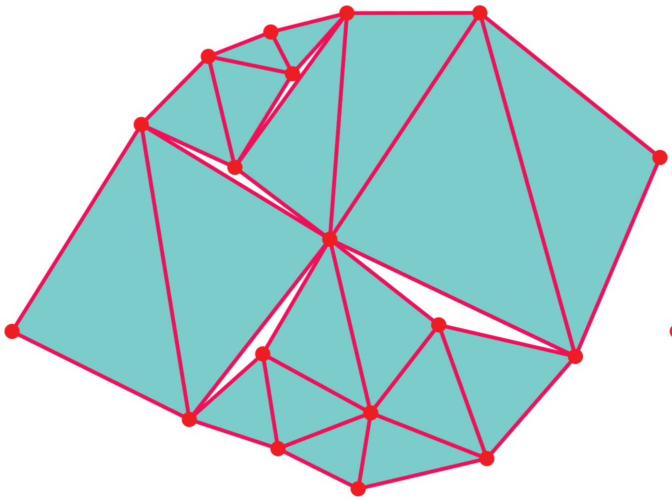
Subdividing a Bezier patch - example



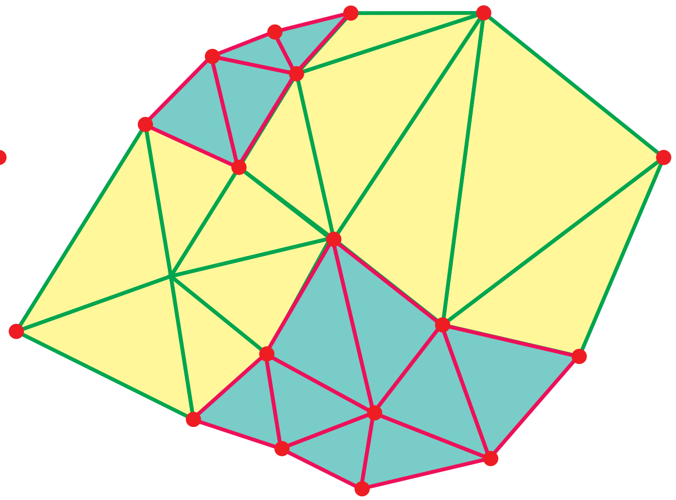
Triangulating the subdivided patch



Final quadrilateral mesh



Naïve triangulation



More intelligent triangulation

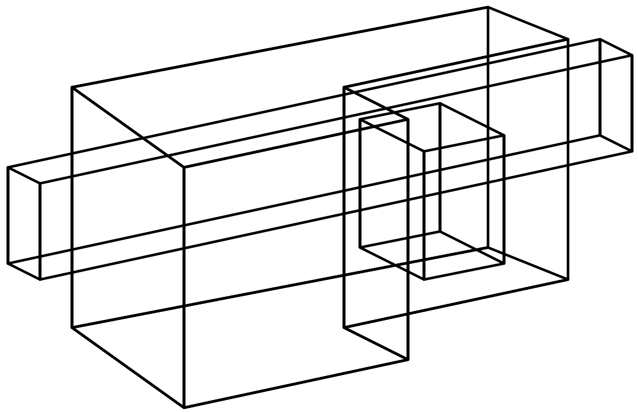
- need to be careful not to generate holes
- need to be equally careful when subdividing connected patches

3D scan conversion

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

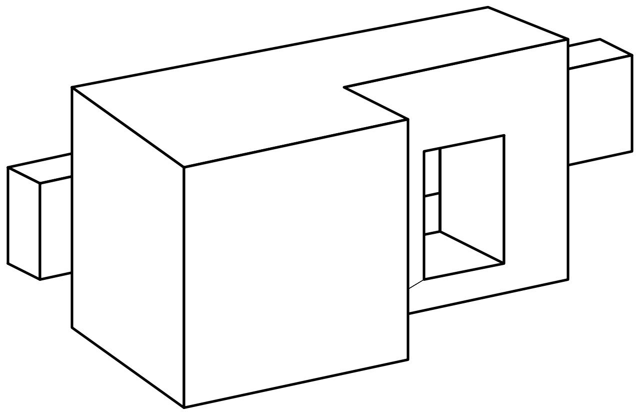
3D line drawing

- 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



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



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**

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

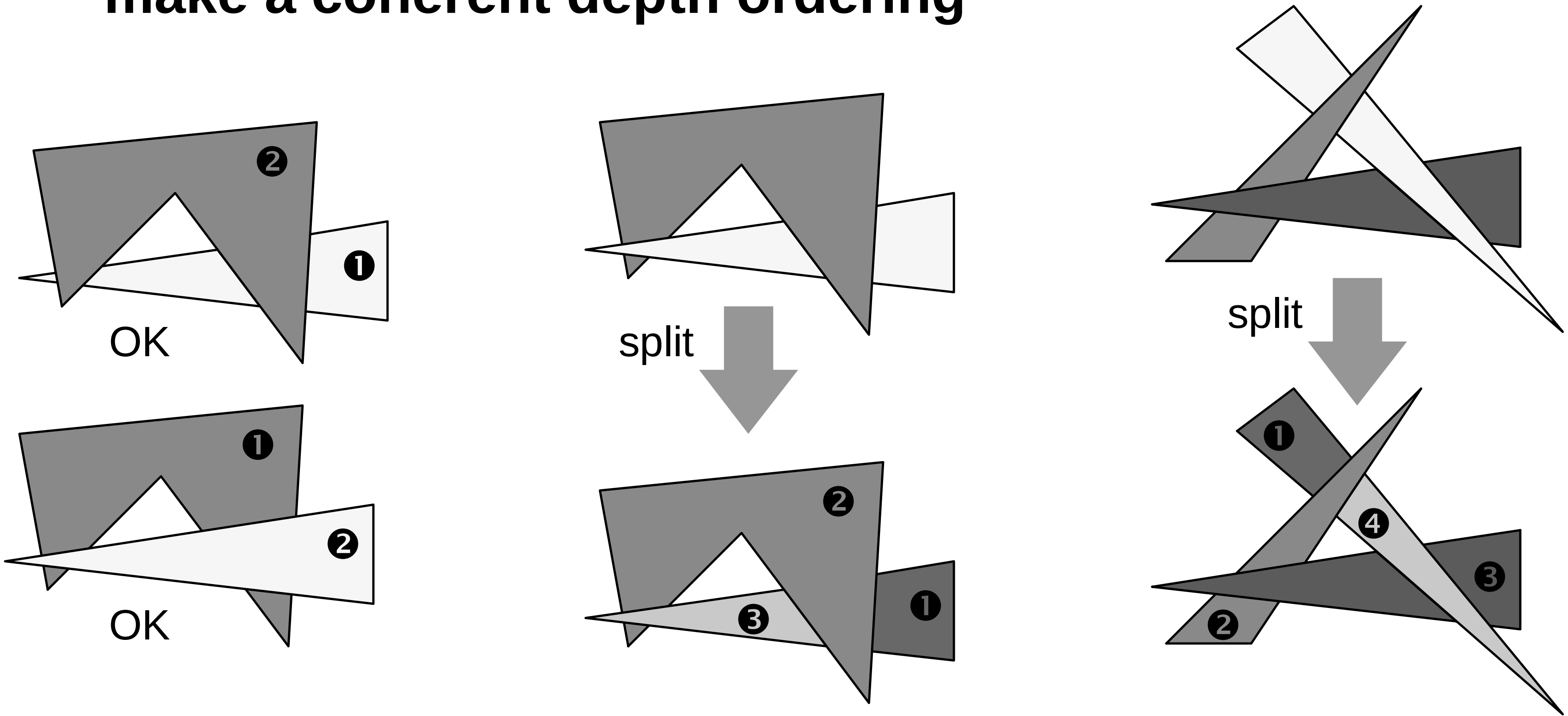
❹ draw the polygons in depth order from back to front

- "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?**
    - **❺ do the projections of the two polygons into the xy plane not overlap?**

tests get more expensive

  - **if all 5 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**

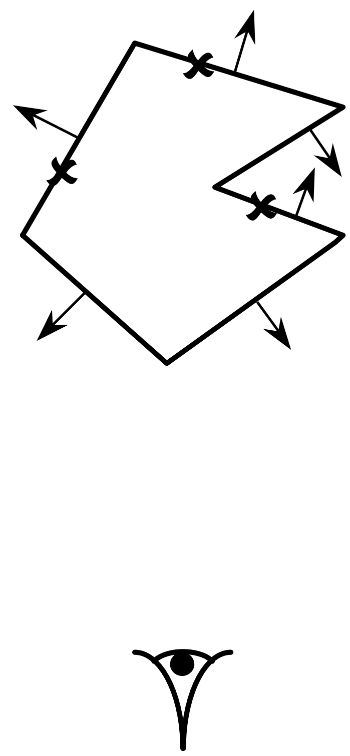
Depth sort: comments

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

- the ordering is only valid from one particular viewpoint

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**

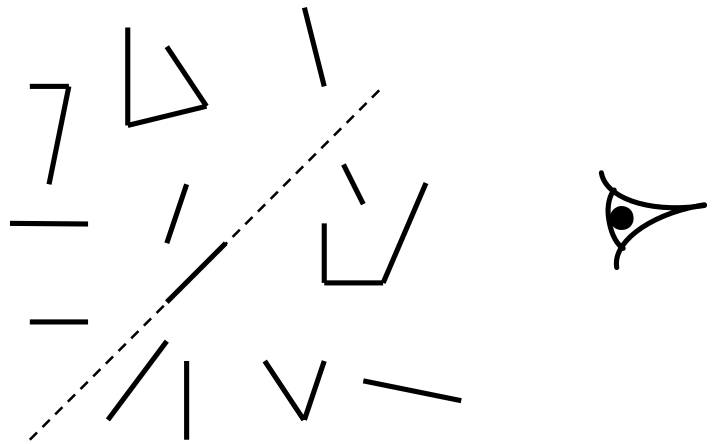


Binary Space-Partitioning trees

- 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

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



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

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**

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**

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

z-buffer basics

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

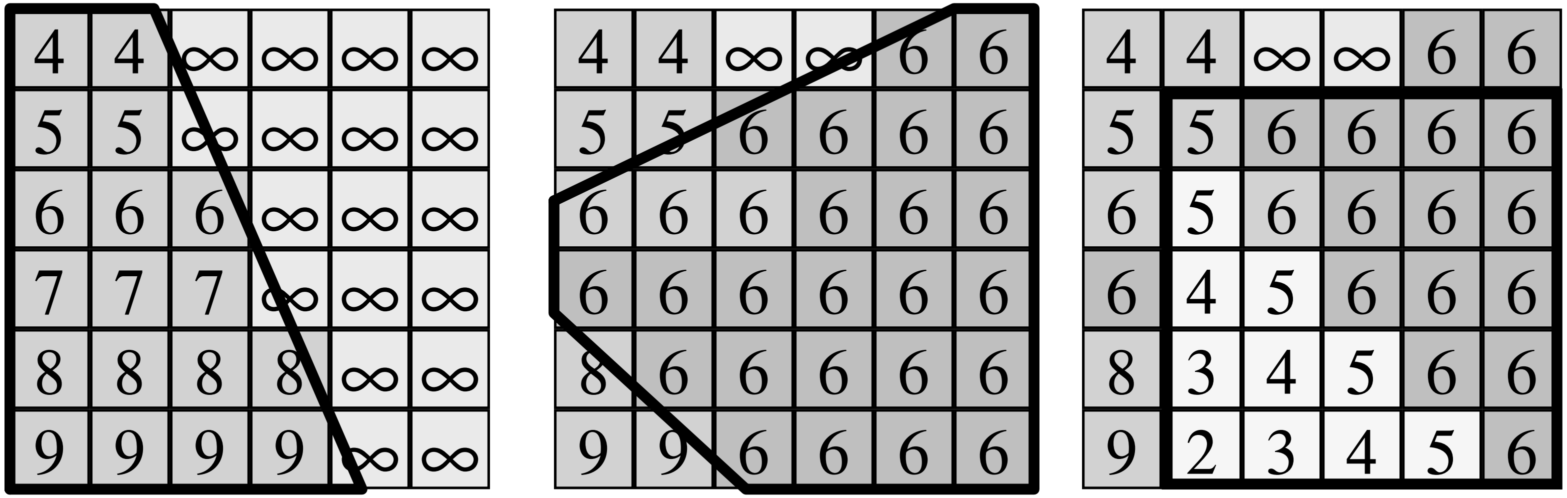
z-buffer algorithm

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

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

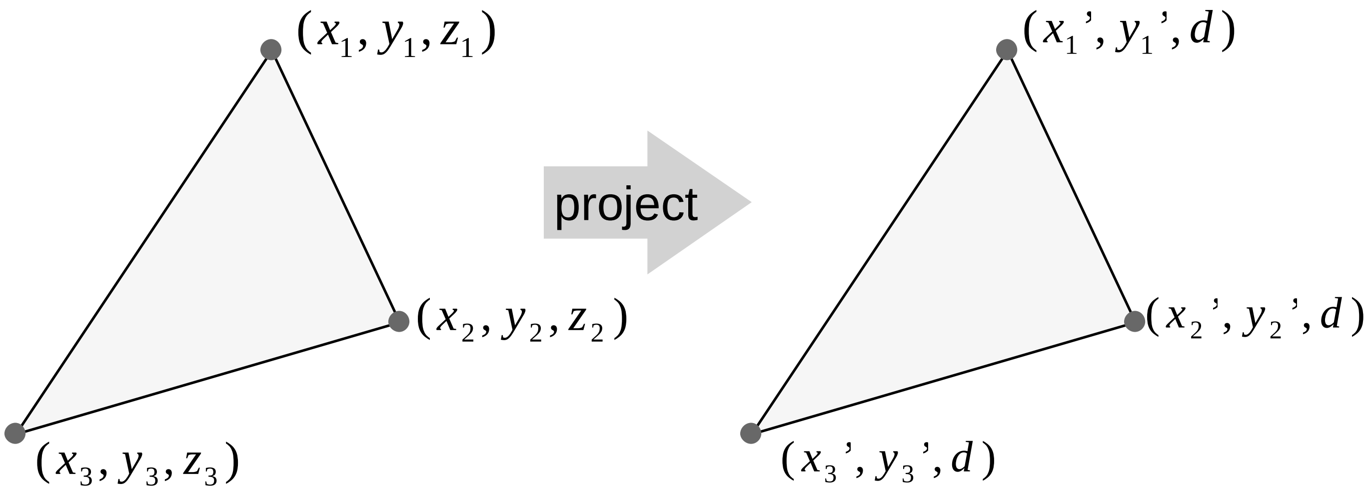
This is essentially the 2D polygon scan conversion algorithm with depth calculation and depth comparison added.

z-buffer example



Interpolating depth values 1

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



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

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

Interpolating depth values 2

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

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

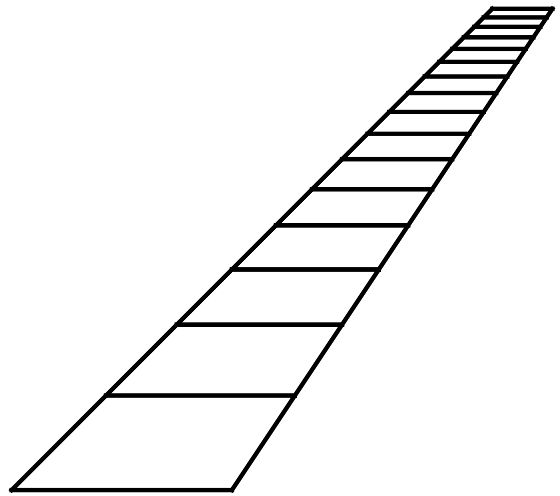
- we can interpolate x and y in 2D

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

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

- but z must be interpolated in 3D

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



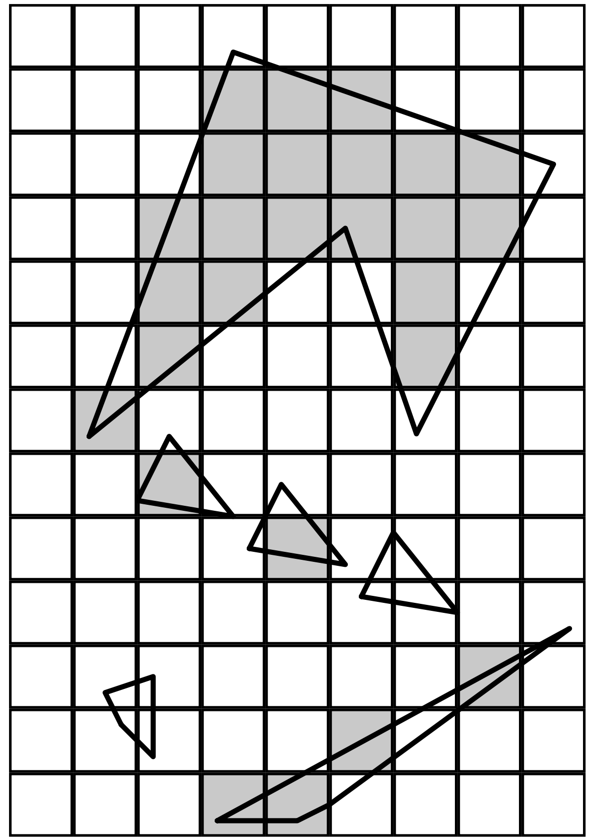
Comparison of methods

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

Sampling

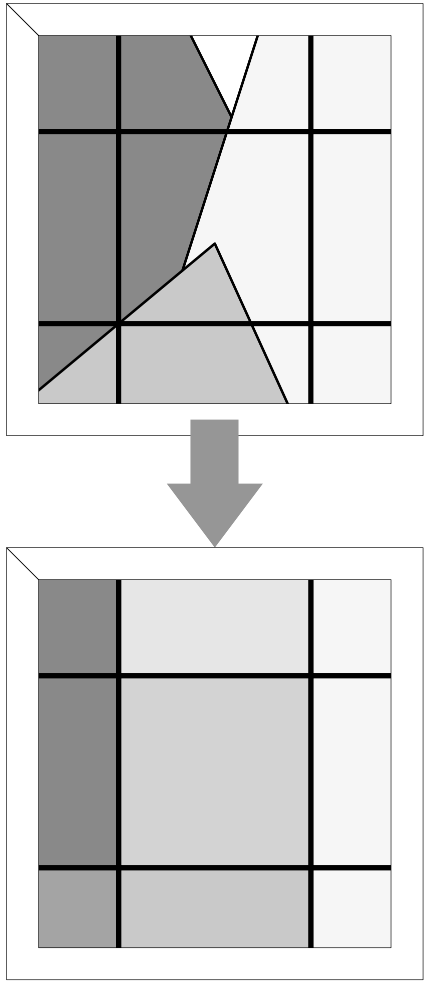
- all of the methods so far take a single sample for each pixel at the precise centre of the pixel
  - i.e. the value for each pixel is the colour of the polygon which happens to lie exactly under the centre of the pixel
- this leads to:
  - stair step (jagged) edges to polygons
  - small polygons being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



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

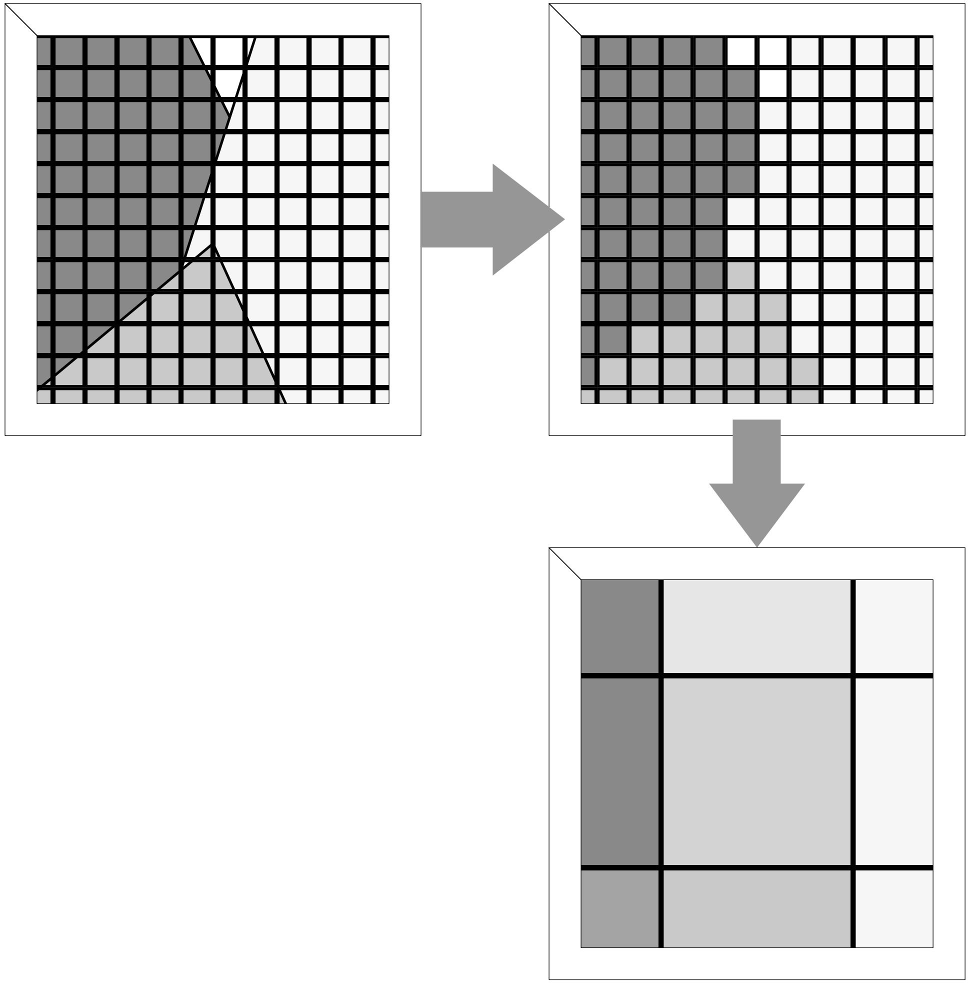
Anti-aliasing method 1: area averaging



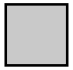
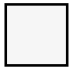
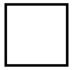
- **average the contributions of all polygons to each pixel**
  - **e.g. assume pixels are square and we just want the average colour in the square**
  - **Ed Catmull developed an algorithm which does this:**
    - works a scan-line at a time
    - clips all polygons to the scan-line
    - determines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

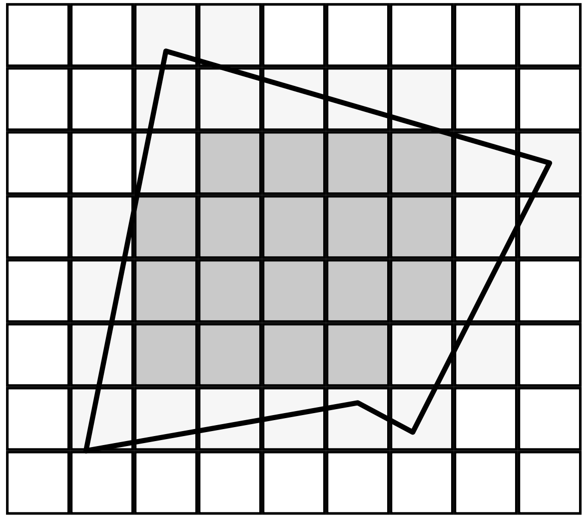
Anti-aliasing method 2: super-sampling

- 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



The A-buffer

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



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

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

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

- a mask is a 4×8 array of bits:

1	1	1	1	1	1	1	1
0	0	0	1	1	1	1	1
0	0	0	0	0	0	1	1
0	0	0	0	0	0	0	0

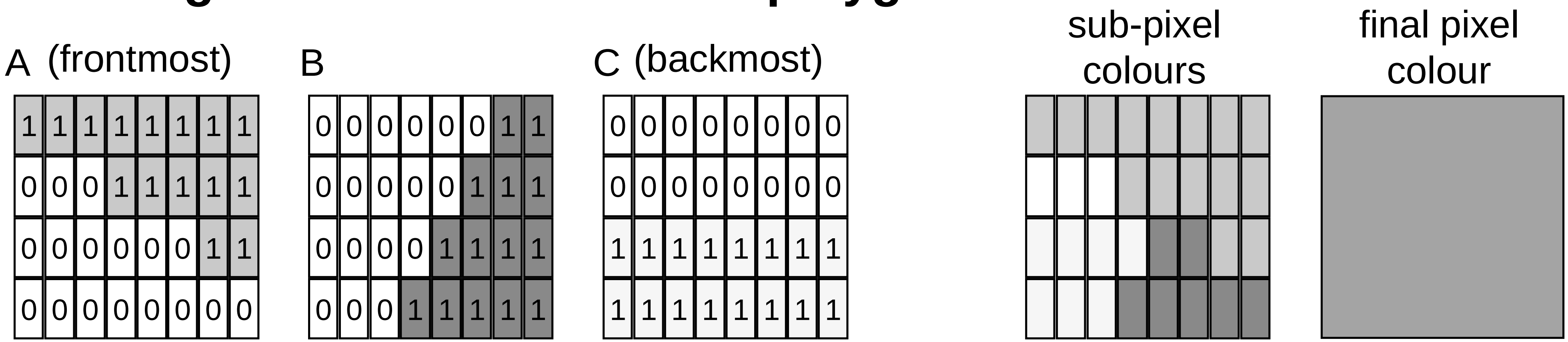
1 = polygon covers this sub-pixel

0 = polygon doesn't cover this sub-pixel

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

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

$\neg A \wedge B$ =00000000 00000000 00001100 00011111
$\neg A \wedge \neg B \wedge C$ =00000000 00000000 11110000 11100000

A covers 15/32 of the pixel
$\neg A \wedge B$ covers 7/32 of the pixel
$\neg A \wedge \neg B \wedge C$ covers 7/32 of the pixel

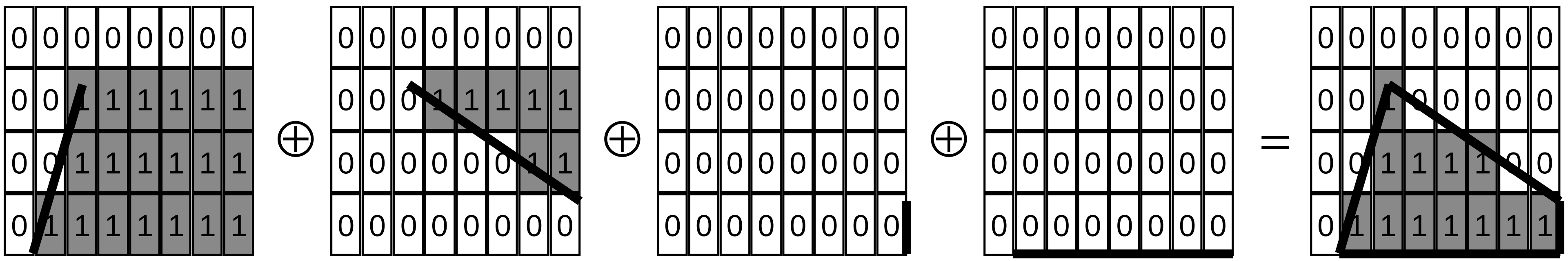
Making the A-buffer more efficient

- 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

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



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 seems more sensible
- what does the A stand for in A-buffer?
  - anti-aliased, area averaged, accumulator

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

Illumination & shading

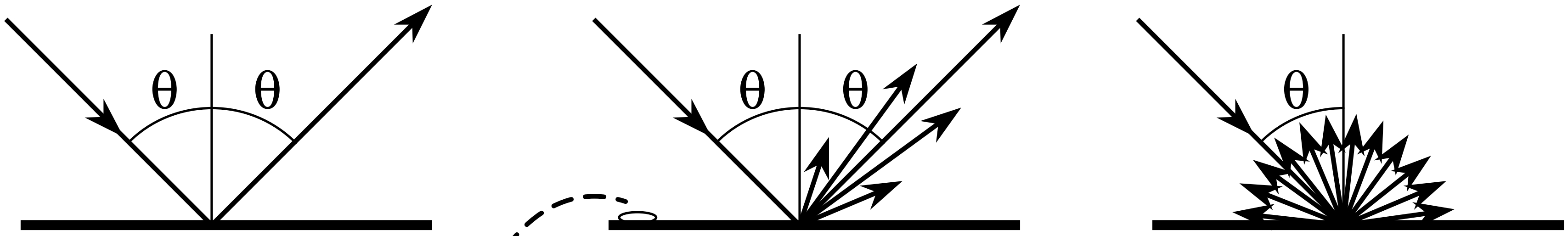
- until now we have assumed that each polygon is a uniform colour and have not thought about how that colour is determined
- things look more realistic if there is some sort of illumination in the scene
- we therefore need a mechanism of determining the colour of a polygon based on its surface properties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

Illumination & shading (continued)

- in the real world every light source emits millions of photons every second
- these photons bounce off objects, pass through objects, and are absorbed by objects
- a tiny proportion of these photons enter your eyes allowing you to see the objects

- tracing the paths of all these photons is not an efficient way of calculating the shading on the polygons in your scene

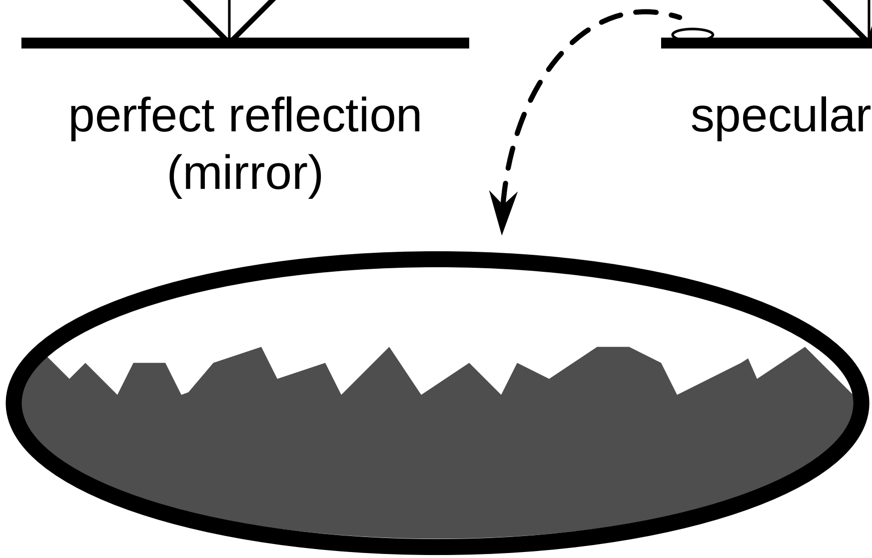
How do surfaces reflect light?



perfect reflection
(mirror)

specular reflection

diffuse reflection
(Lambertian reflection)

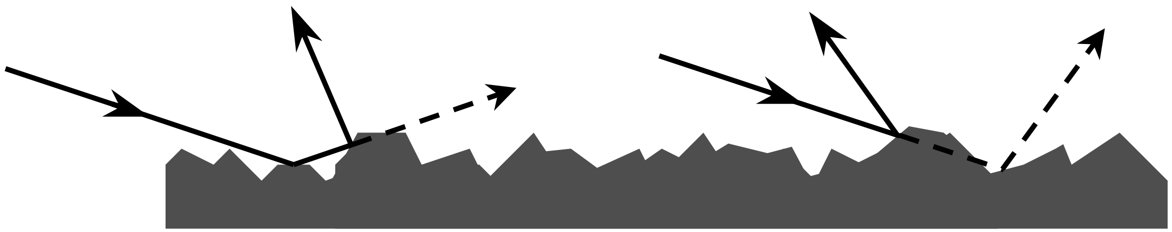


the surface of a specular reflector is facetted, each facet reflects perfectly but in a slightly different direction to the other facets

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

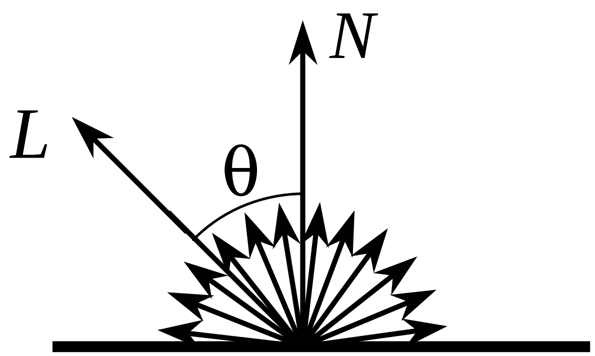


- plastics are good examples of surfaces with:
  - specular reflection in the light’s colour
  - diffuse reflection in the plastic’s colour

Calculating the shading of a polygon

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

Diffuse shading calculation



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

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

N is the normal to the polygon

I_l is the intensity of the light source

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

I is the intensity of the light reflected by the surface

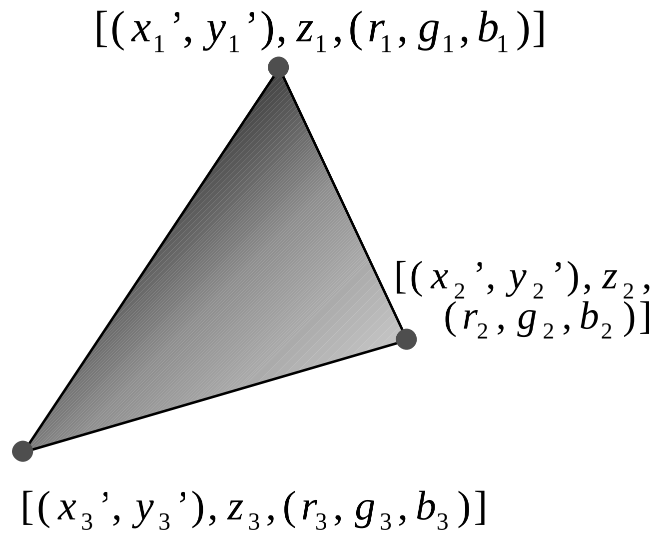
use this equation to set the colour of the whole polygon and draw the polygon using a standard polygon scan-conversion routine

Diffuse shading: comments

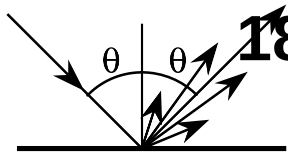
- **can have different I_l and different k_d for different wavelengths (colours)**
- **watch out for $\cos\theta < 0$**
  - **implies that the light is behind the polygon and so it cannot illuminate this side of the polygon**
- **do you use one-sided or two-sided polygons?**
  - **one sided: only the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

Gouraud shading

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

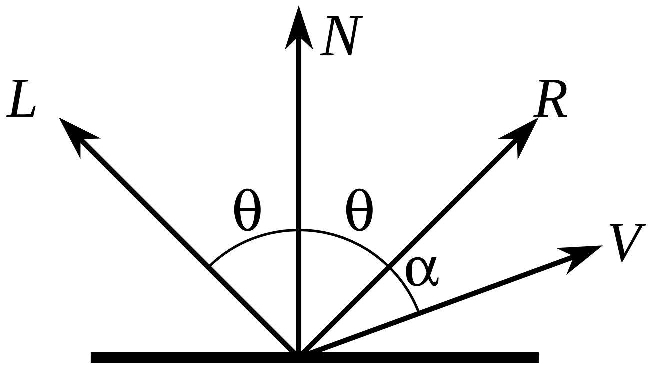


Henri Gouraud, “Continuous Shading of Curved Surfaces”, *IEEE Trans Computers*, **20**(6), 1971



Specular reflection

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



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

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

R is the vector of perfect reflection

N is the normal to the polygon

V is a normalised vector pointing at the viewer

I_l is the intensity of the light source

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

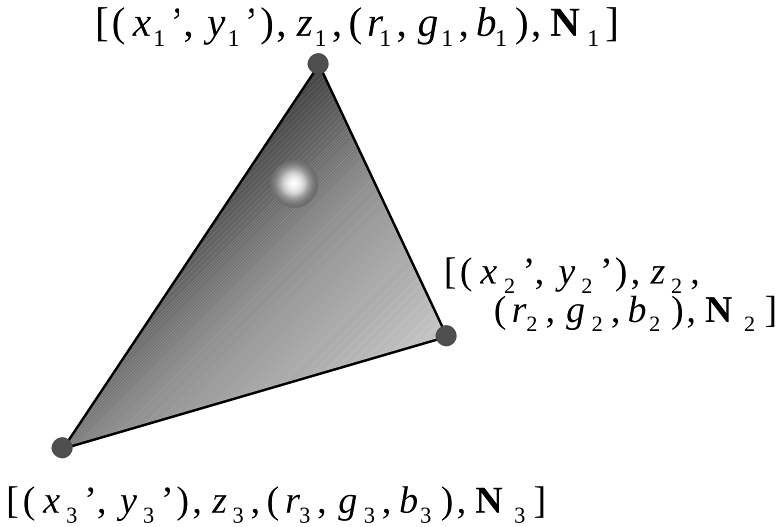
n is Phong's *ad hoc* "roughness" coefficient

I is the intensity of the specularly reflected light

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

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

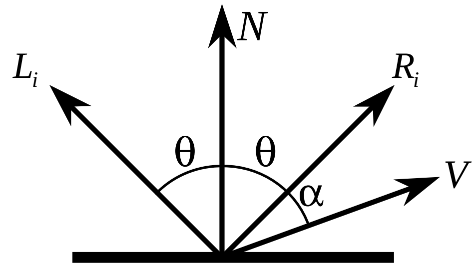
The gross assumptions revisited

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

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**

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



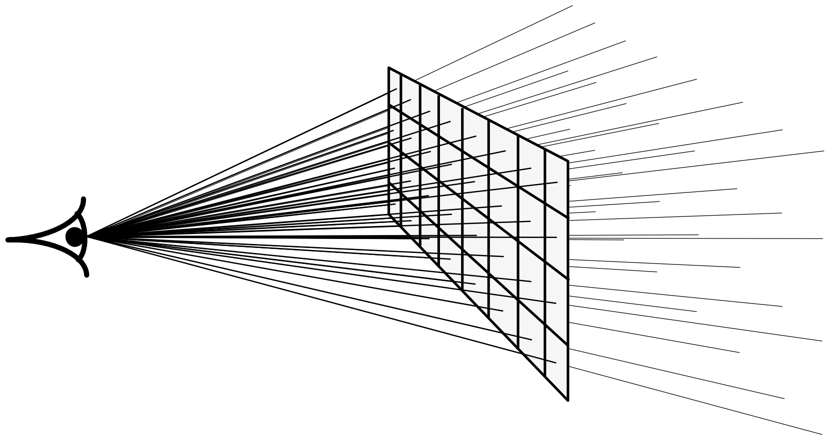
  - **the more lights there are in the scene, the longer this calculation will take**

Illumination & shading: comments

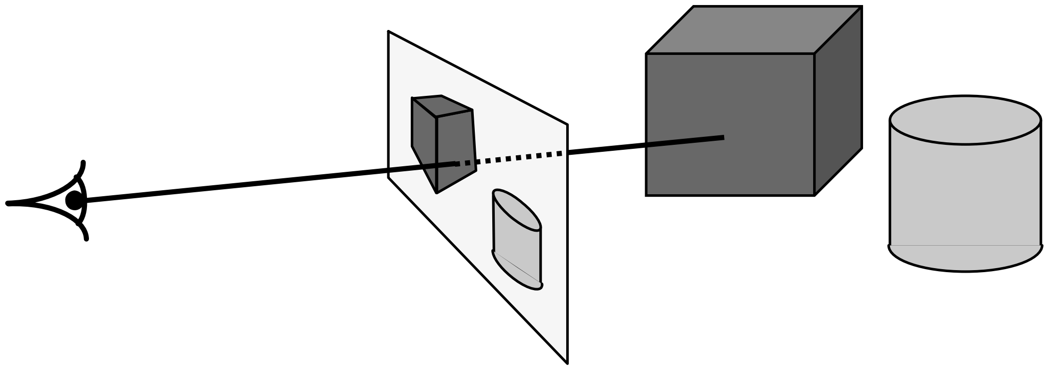
- **how good is this shading equation?**
  - **gives reasonable results but most objects tend to look as if they are made out of plastic**
  - **Cook & Torrance have developed a more realistic (and more expensive) shading model which takes into account:**
    - micro-facet geometry (which models, amongst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

Ray tracing

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



shoot a ray through each pixel



whatever the ray hits determines the colour of that pixel

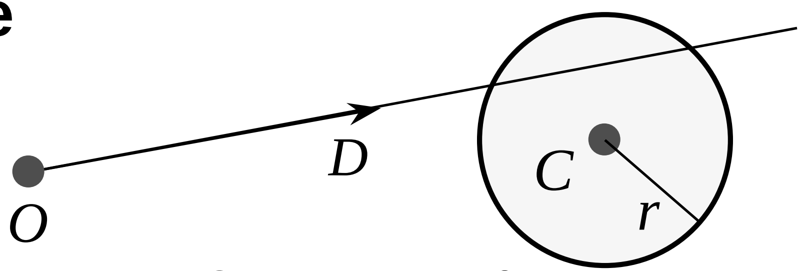
Ray tracing algorithm

```
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 ;
```

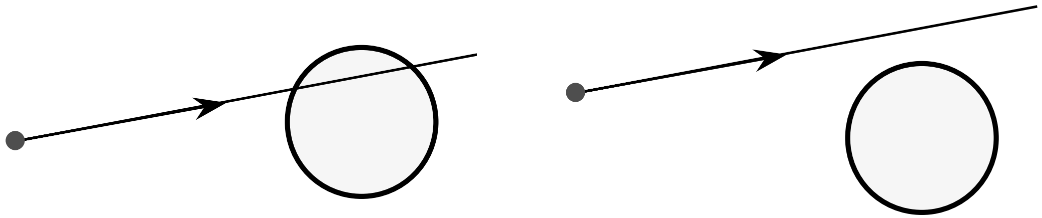
Intersection of a ray with an object 1

- **sphere**



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

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



d real $\qquad$ d imaginary

$$a = D \cdot D$$

$$b = 2D \cdot (O - C)$$

$$c = (O - C)\cdot(O - C) - r^2$$

$$d = \sqrt{b^2 - 4ac}$$

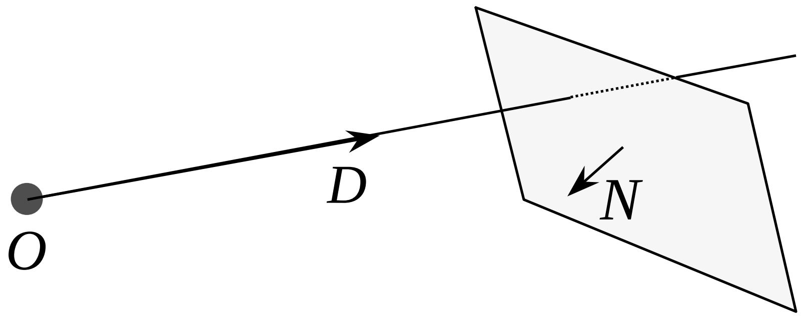
$$s_1 = \frac{-b + d}{2a}$$

$$s_2 = \frac{-b - d}{2a}$$

- **cylinder, cone, torus**
  - **all similar to sphere**

Intersection of a ray with an object 2

- **plane**



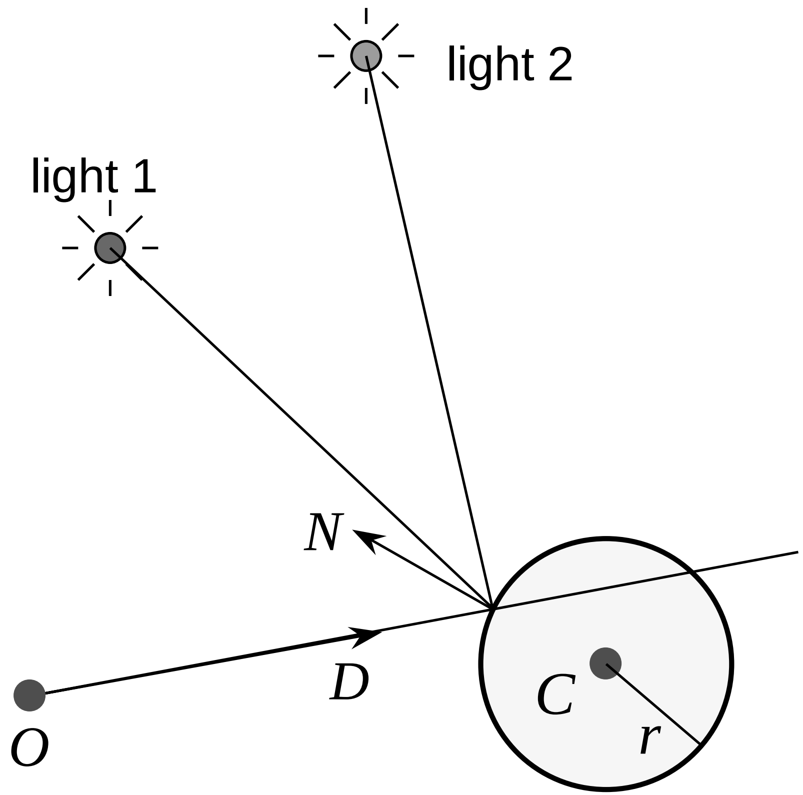
ray: $P = O + sD,\ s \geq 0$

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

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

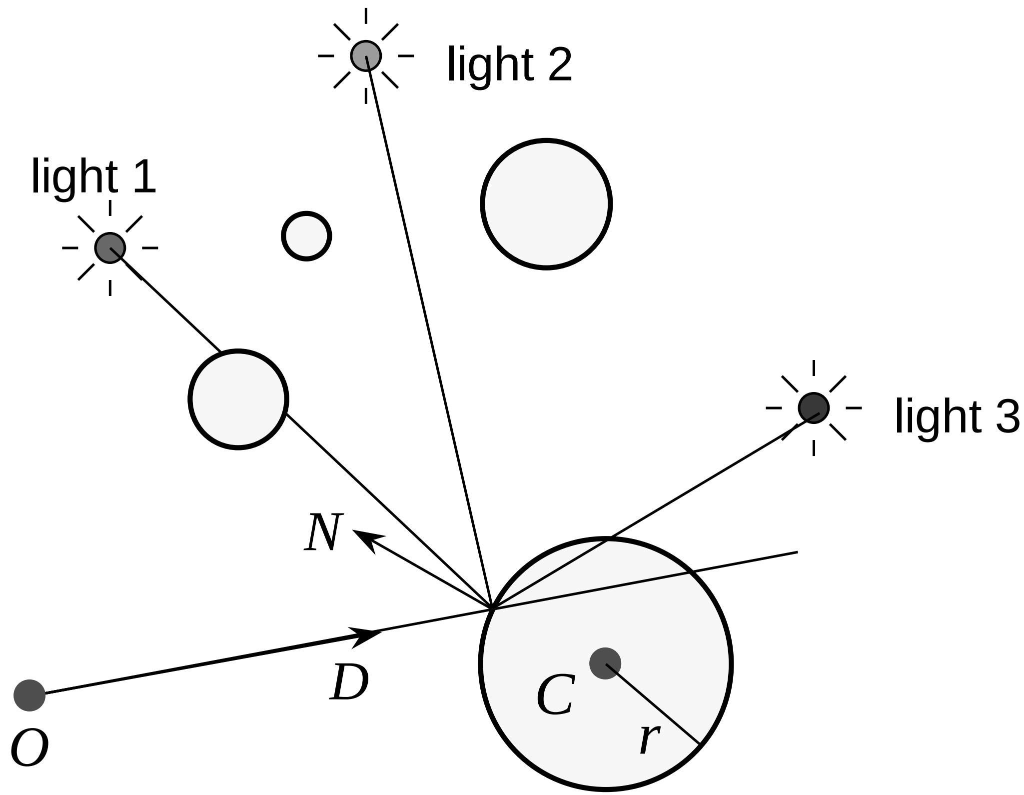
- **box, polygon, polyhedron**
  - **defined as a set of bounded planes**

Ray tracing: shading



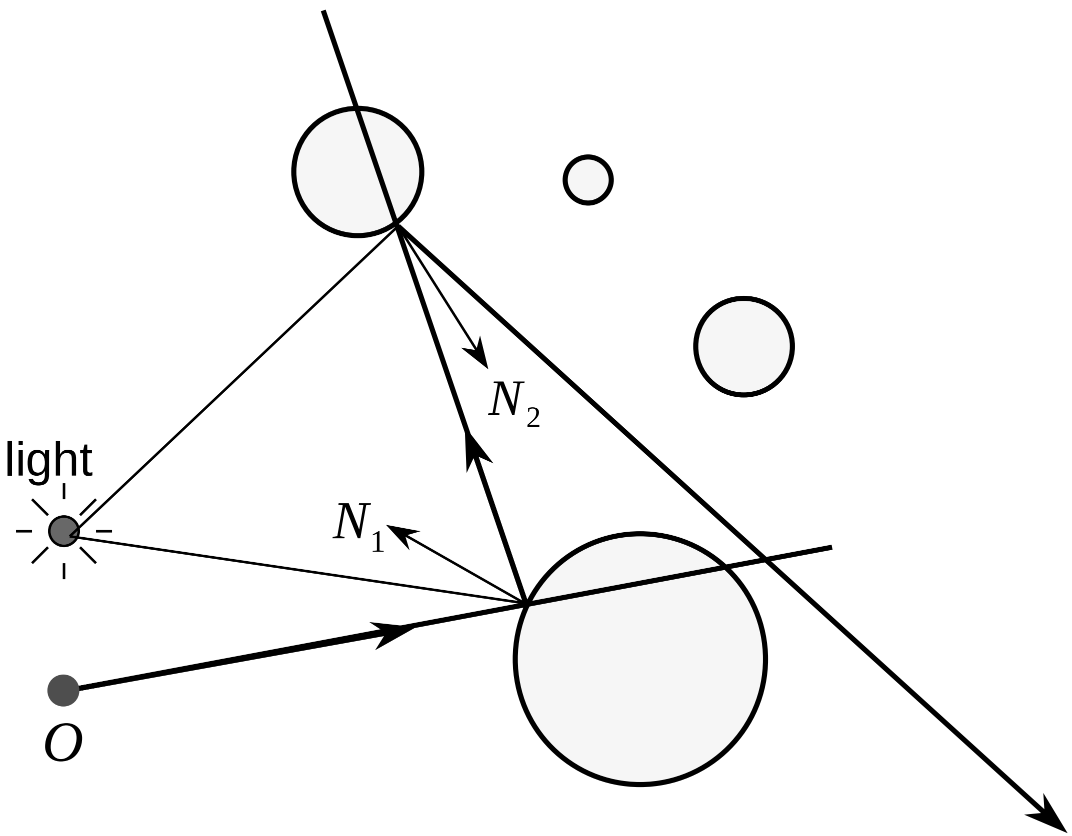
- **once you have the intersection of a ray with the nearest object you can also:**
  - **calculate the normal to the object at that intersection point**
  - **shoot rays from that point to all of the light sources, and calculate the diffuse and specular reflections off the object at that point**
  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

Ray tracing: shadows



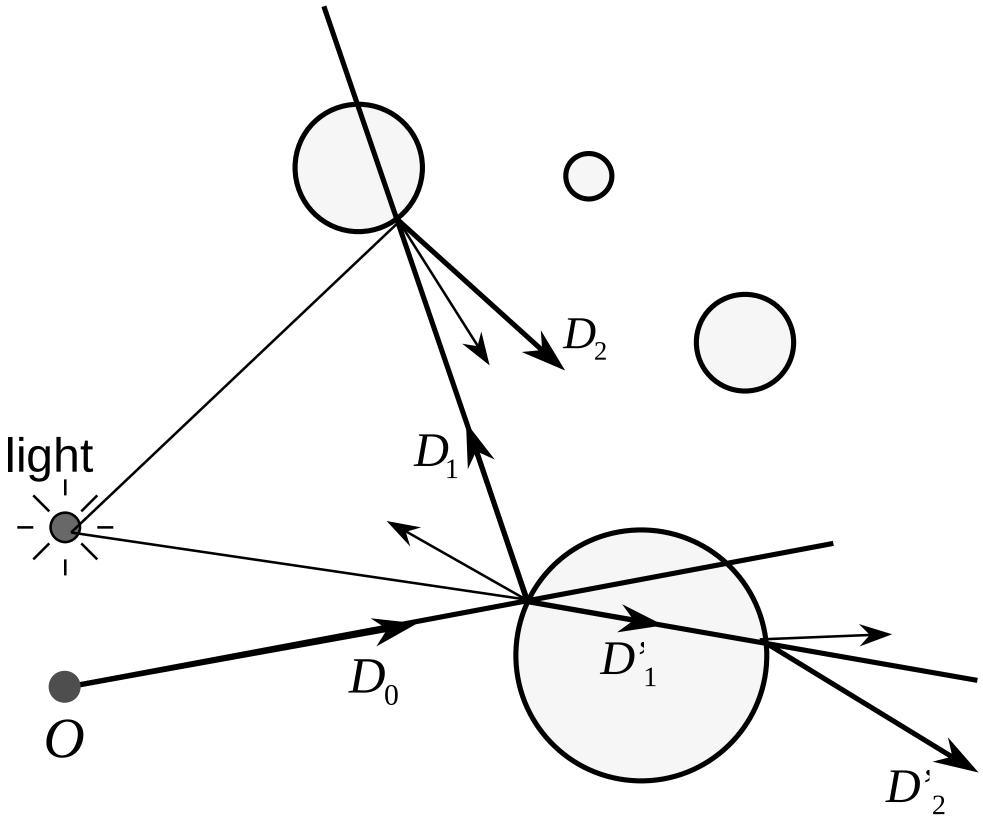
- because you are tracing rays from the intersection point to the light, you can check whether another object is between the intersection and the light and is hence casting a shadow
  - also need to watch for self-shadowing

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

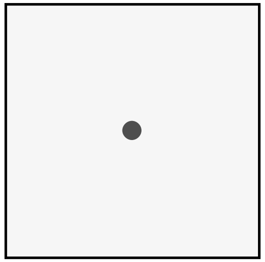
Ray tracing: transparency & refraction

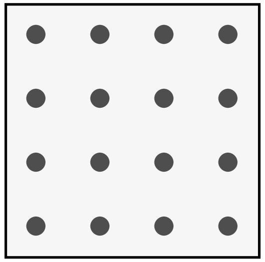


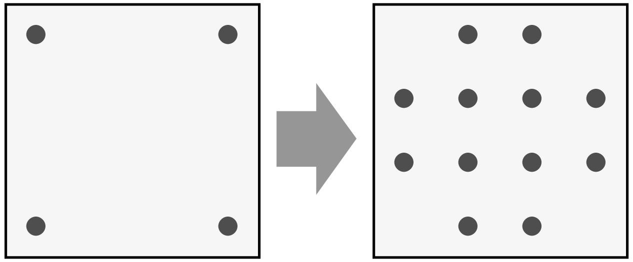
- objects can be totally or partially transparent
  - this allows objects behind the current one to be seen through it
- transparent objects can have refractive indices
  - bending the rays as they pass through the objects
- transparency + reflection means that a ray can split into two parts

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

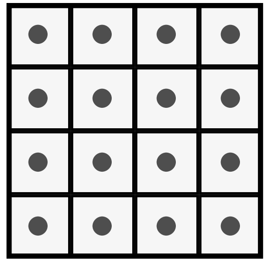


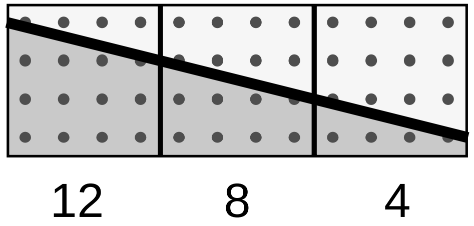




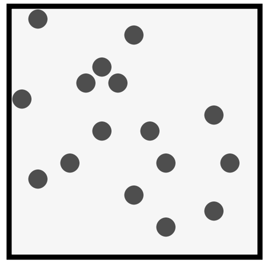
Types of super-sampling 1

- **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**



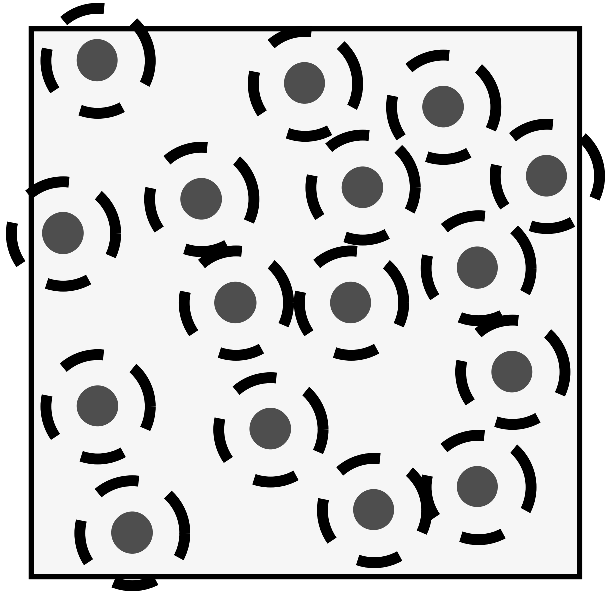


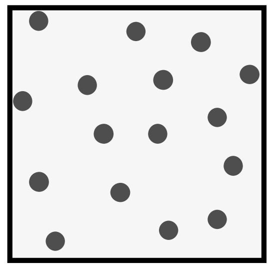
- **random**
  - **shoot N rays at random points in the pixel**
  - **replaces aliasing artefacts with noise artefacts**
    - the eye is far less sensitive to noise than to aliasing



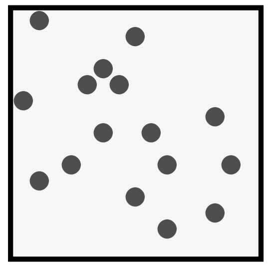
Types of super-sampling 2

- **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**





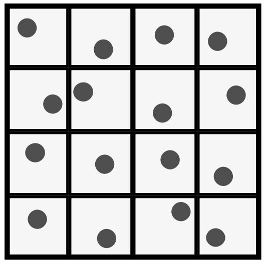
Poisson disc

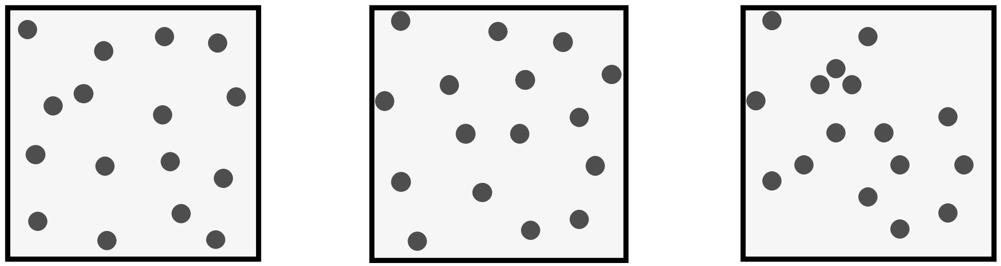


pure random

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**





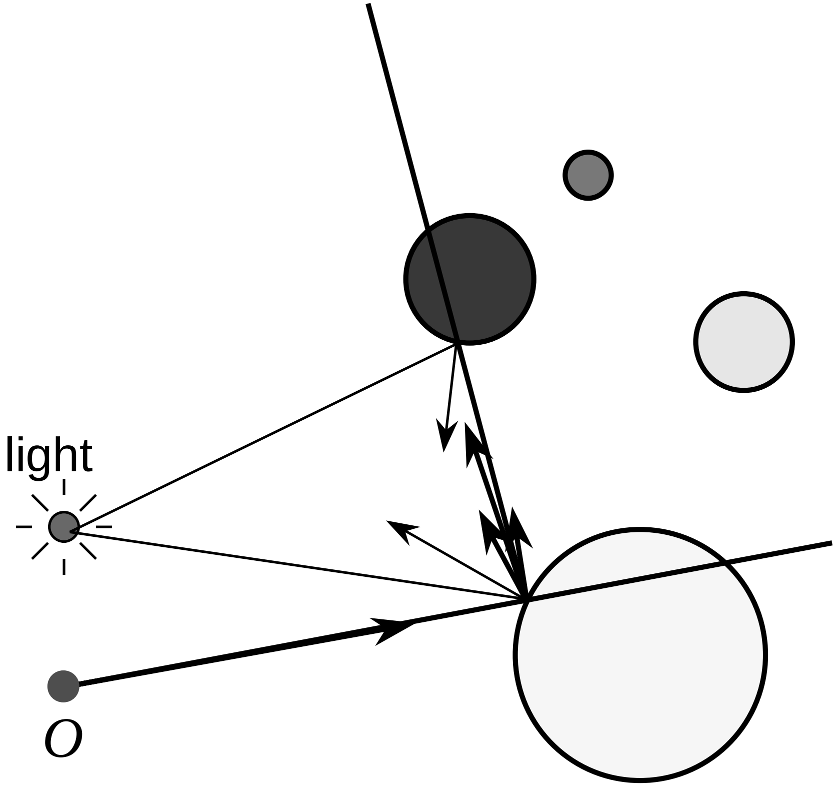
More reasons for wanting to take multiple samples per pixel

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

Examples of distributed ray tracing

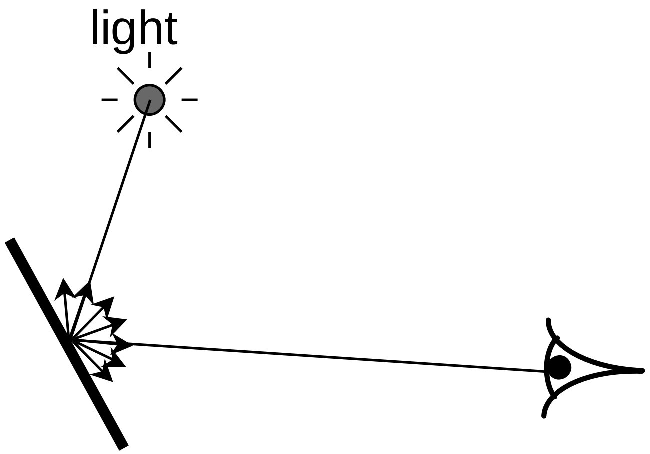
- **distribute the samples for a pixel over the pixel area**
  - get random (or jittered) super-sampling
  - used for anti-aliasing
- **distribute the rays going to a light source over some area**
  - allows area light sources in addition to point and directional light sources
  - produces soft shadows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

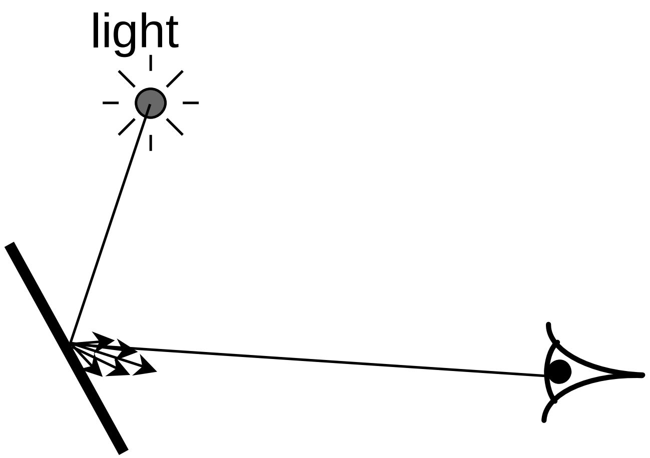
Distributed ray tracing for specular reflection



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

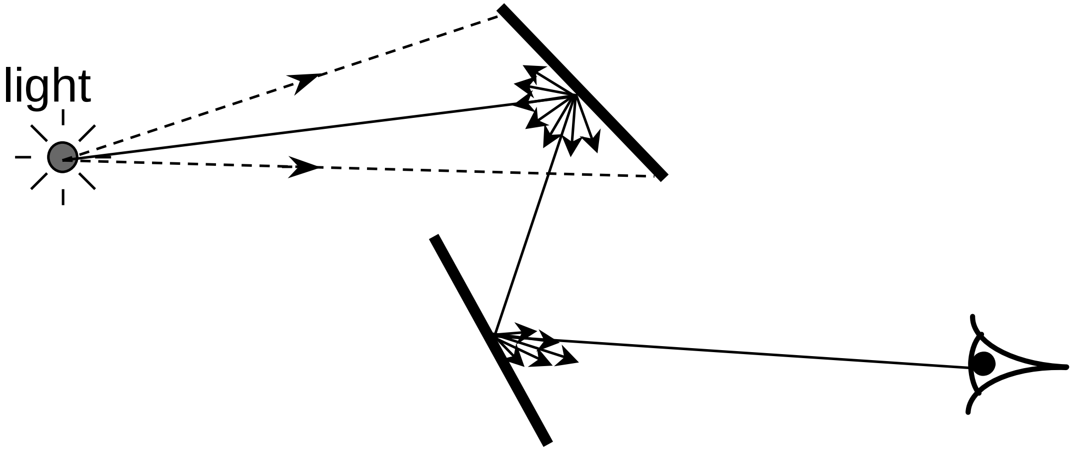
Handling direct illumination



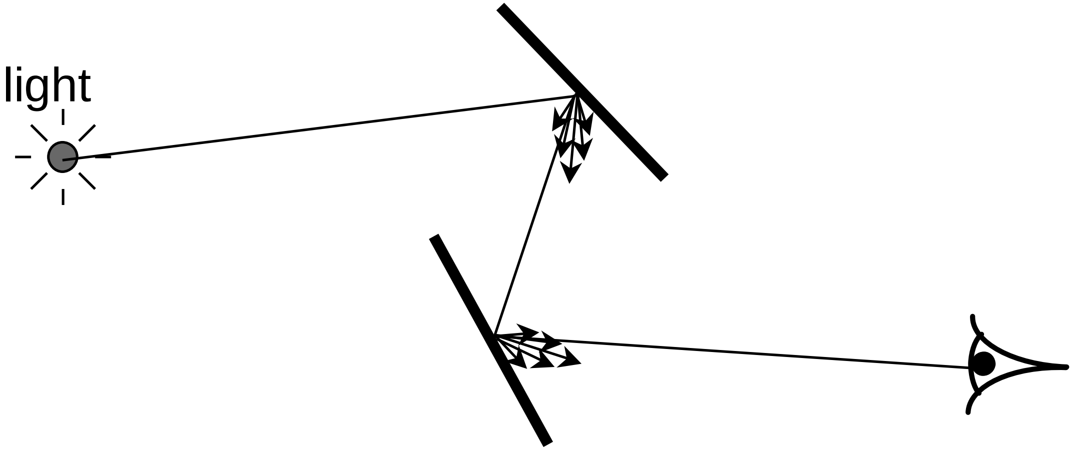


- **diffuse reflection**
  - **handled by ray tracing and polygon scan conversion**
  - **assumes that the object is a perfect Lambertian reflector**
- **specular reflection**
  - **also handled by ray tracing and polygon scan conversion**
  - **use Phong's approximation to true specular reflection**

Handing indirect illumination: 1

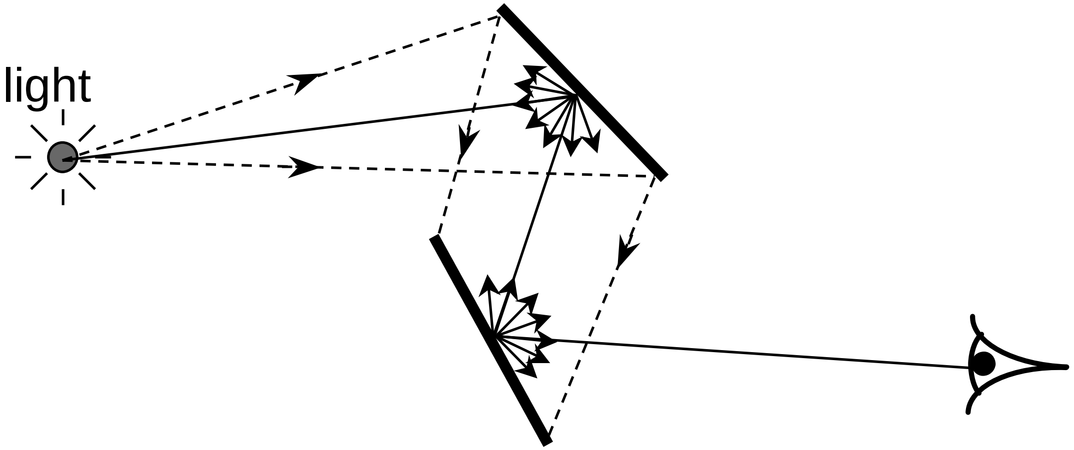


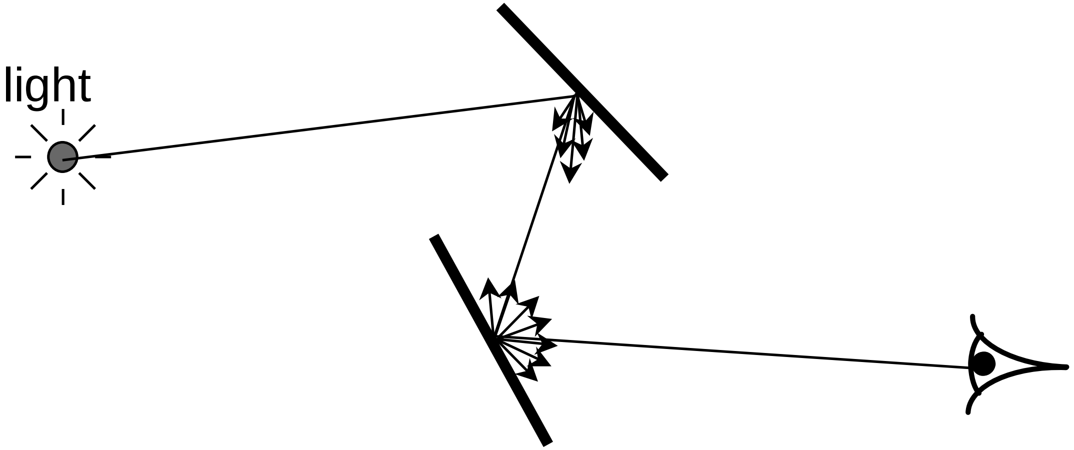
- diffuse to specular
  - handled by distributed ray tracing



- specular to specular
  - also handled by distributed ray tracing

Handing indirect illumination: 2



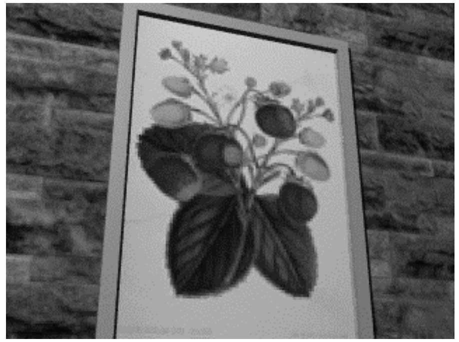


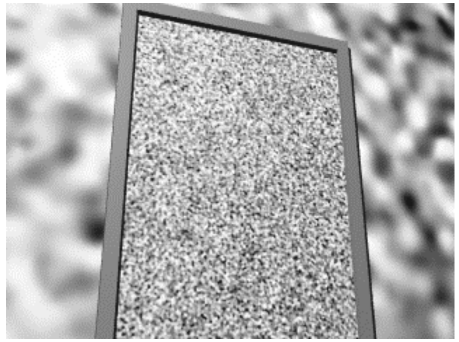
- diffuse to diffuse
  - handled by radiosity
    - covered in the Part II Advanced Graphics course
- specular to diffuse
  - handled by no usable algorithm
  - some research work has been done on this but uses enormous amounts of CPU time

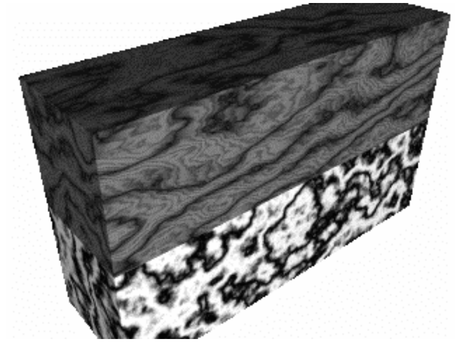
Hybrid algorithms

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

Surface detail

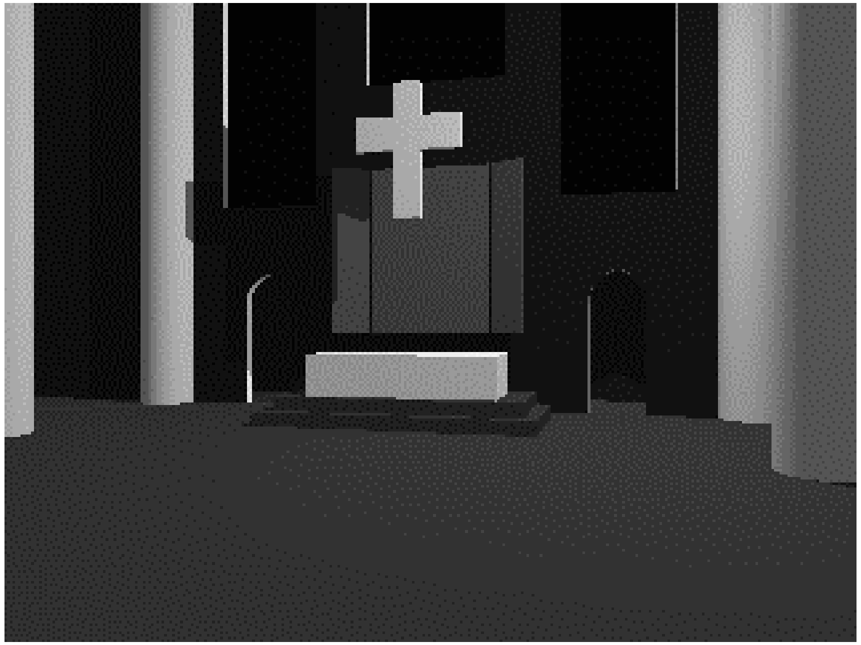




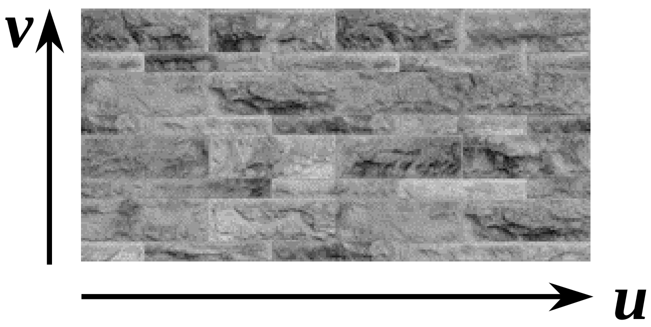


- so far we have assumed perfectly smooth, uniformly coloured surfaces
- real life isn't like that:
  - multicoloured surfaces
    - e.g. a painting, a food can, a page in a book
  - bumpy surfaces
    - e.g. almost any surface! (very few things are perfectly smooth)
  - textured surfaces
    - e.g.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

Texture mapping

without	with
	
all surfaces are smooth and of uniform colour	most surfaces are textured with 2D texture maps the pillars are textured with a solid texture

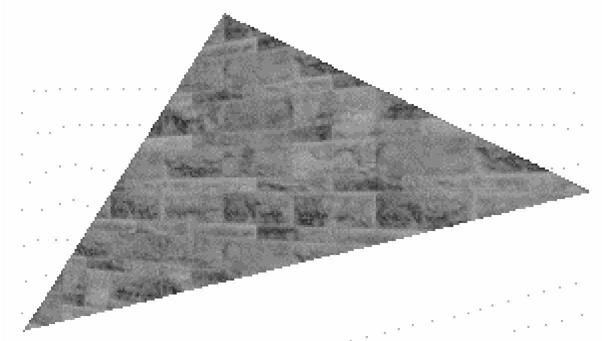
Basic texture mapping

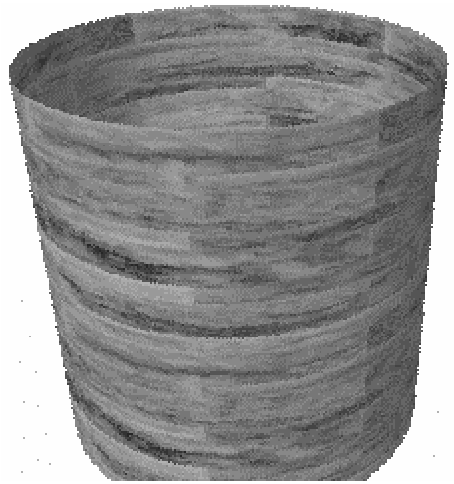


- a texture is simply an image, with a 2D coordinate system (u,v)

- 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

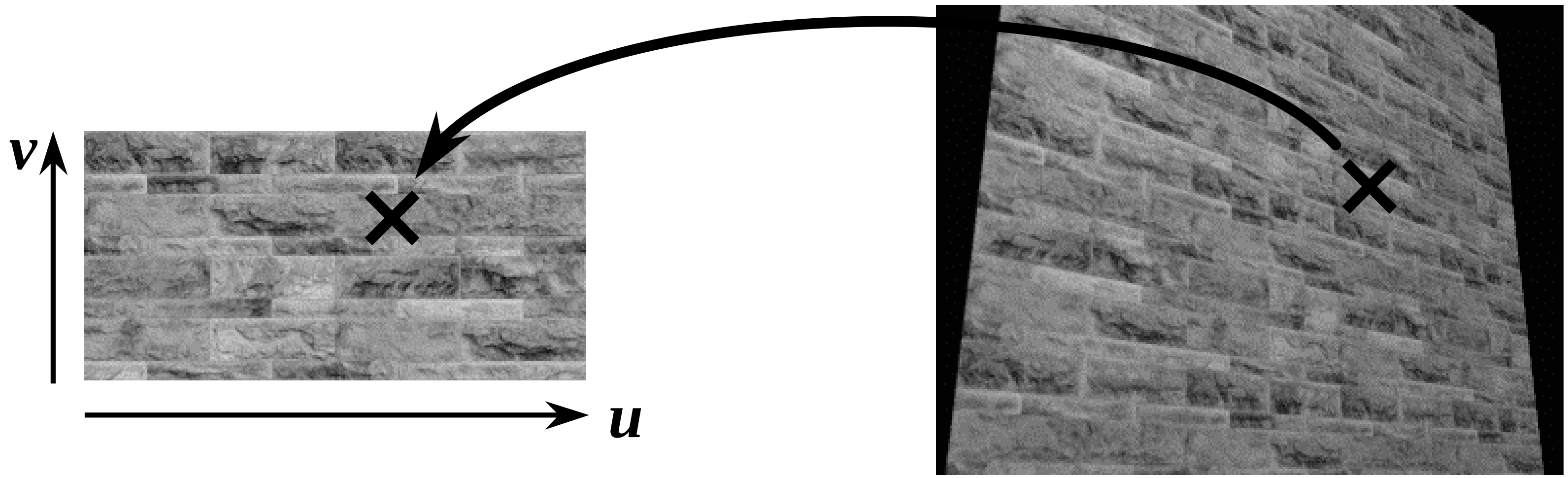
Paramaterising a primitive





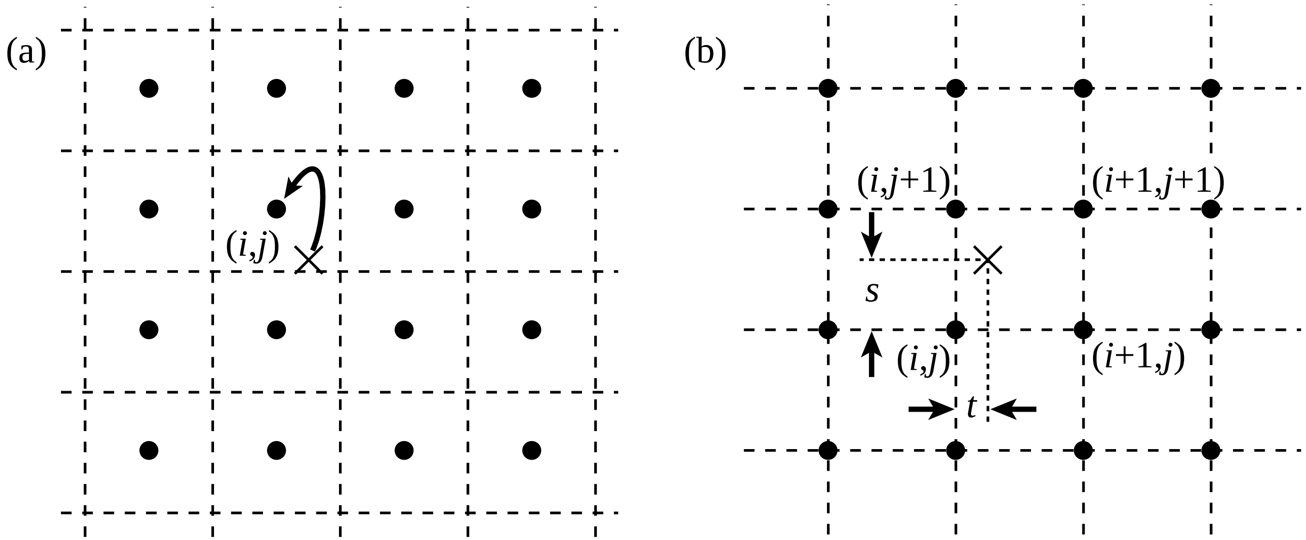
- polygon: give (u,v) coordinates for three vertices, or treat as part of a plane
- plane: give u-axis and v-axis directions in the plane
- cylinder: one axis goes up the cylinder, the other around the cylinder

Sampling texture space



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

Sampling texture space: finding the value

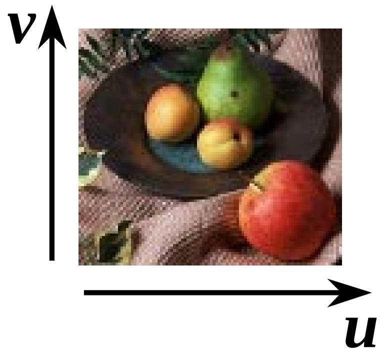


- nearest neighbour: the sample value is the nearest pixel value to the sample point
- bilinear reconstruction: the sample value is the weighted mean of pixels around the sample point

Sampling texture space: interpolation methods

- nearest neighbour
  - fast with many artefacts
- bilinear
  - reasonably fast, blurry
- biquadratic
  - slower, good results
- bicubic
  - even slower, slightly better results
- others
  - Gaussian, sinc function, global polynomials
  - not really necessary for this sort of work

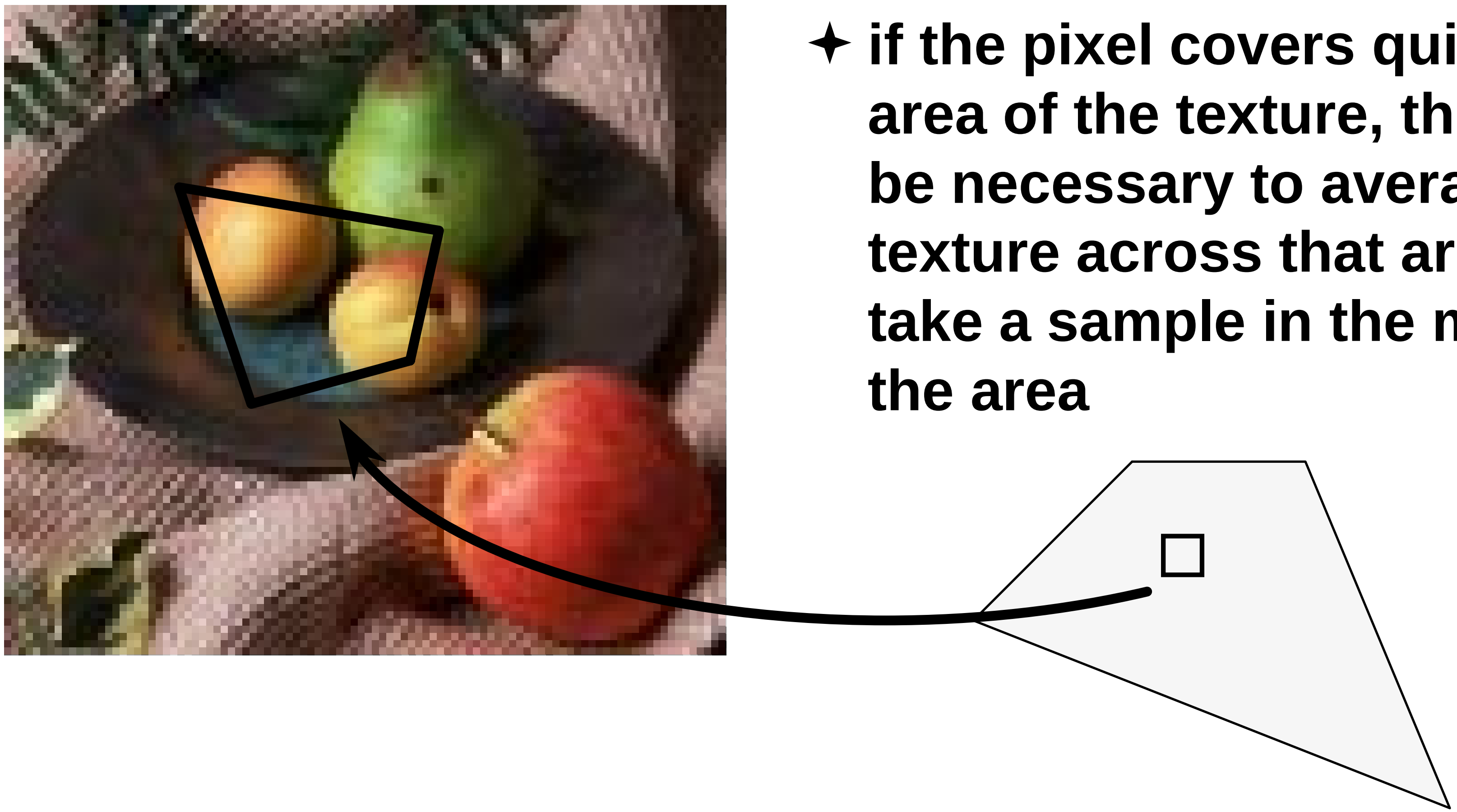
Texture mapping examples



nearest-
neighbour

bilinear

Down-sampling

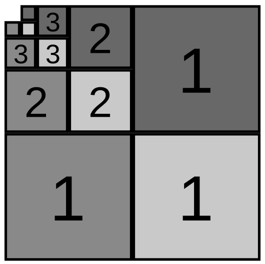


- if the pixel covers quite a large area of the texture, then it will be necessary to average the texture across that area, not just take a sample in the middle of the area

Multi-resolution texture

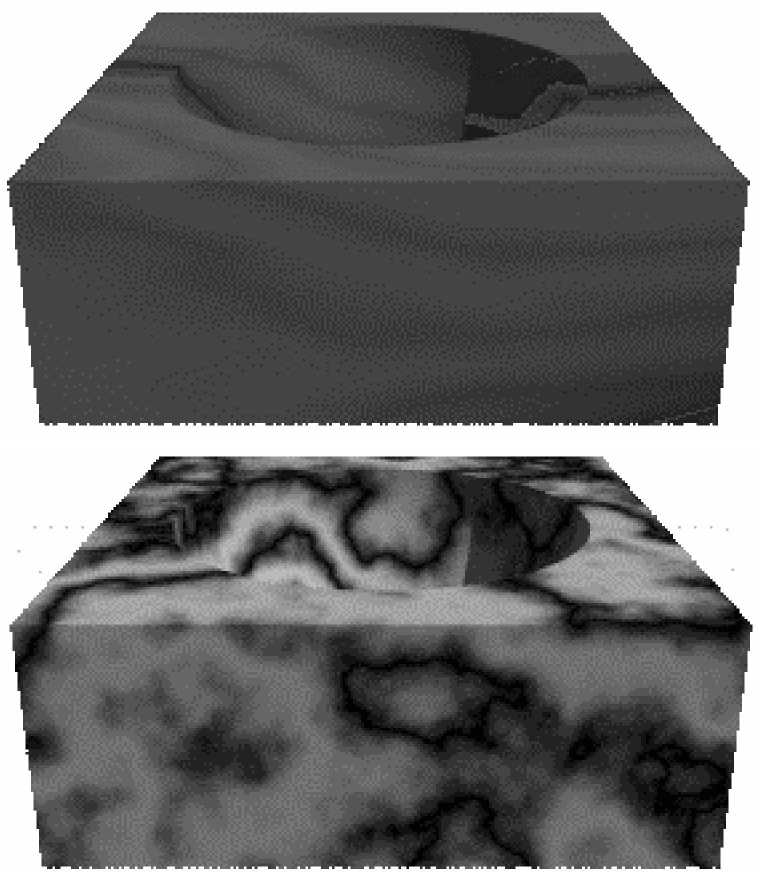
can have multiple versions of the texture at different resolutions

use tri-linear interpolation to get sample values at the appropriate resolution



MIP map: an efficient way to store multi-resoltion colour textures

Solid textures



- texture mapping applies a 2D texture to a surface

  colour = ***f***(***u***,***v***)

- solid textures have colour defined for every point in space

  colour = ***f***(***x***,***y***,***z***)

- permits the modelling of objects which appear to be carved out of a material

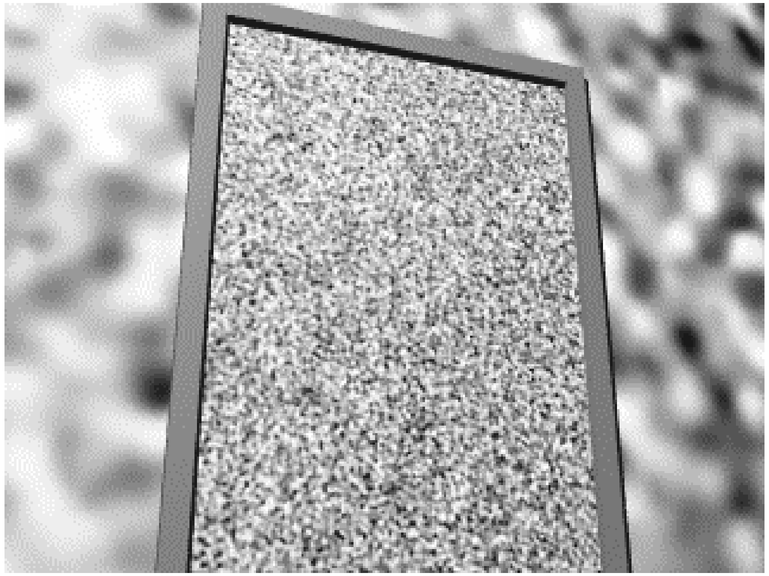
What can a texture map modify?

- 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

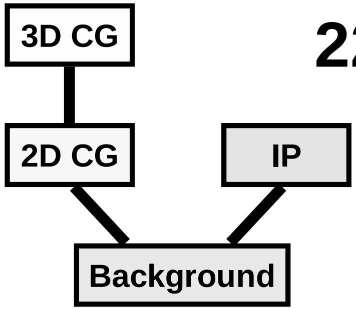
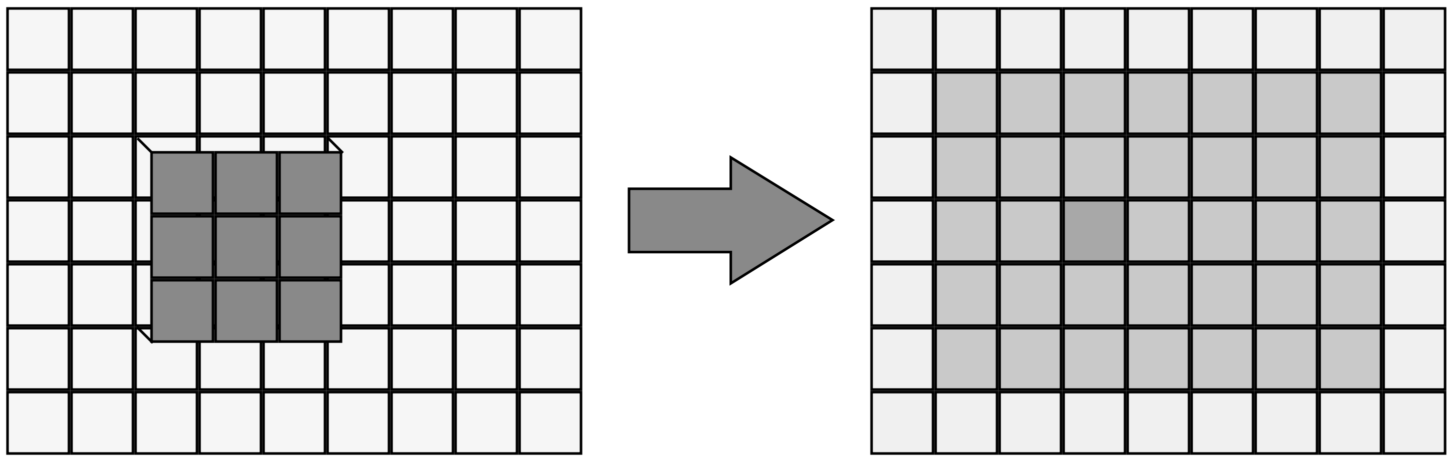


Image Processing

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

Filtering

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



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:

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

Example filters - averaging/blurring

Basic 3x3 blurring filter

$$\begin{array}{|c|c|c|}\hline 1/9 & 1/9 & 1/9 \\ \hline 1/9 & 1/9 & 1/9 \\ \hline 1/9 & 1/9 & 1/9 \\ \hline \end{array} = 1/9 \times \begin{array}{|c|c|c|}\hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline \end{array}$$

Gaussian 5x5 blurring filter

$$1/112 \times \begin{array}{|c|c|c|c|c|}\hline 1 & 2 & 4 & 2 & 1 \\ \hline 2 & 6 & 9 & 6 & 2 \\ \hline 4 & 9 & 16 & 9 & 4 \\ \hline 2 & 6 & 9 & 6 & 2 \\ \hline 1 & 2 & 4 & 2 & 1 \\ \hline \end{array}$$

Gaussian 3x3 blurring filter

$$1/16 \times \begin{array}{|c|c|c|}\hline 1 & 2 & 1 \\ \hline 2 & 4 & 2 \\ \hline 1 & 2 & 1 \\ \hline \end{array}$$

Example filters - edge detection

Horizontal **Vertical** **Diagonal**

Horizontal:

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

Vertical:

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

Diagonal:

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

Prewitt filters

1	0
0	-1

0	1
-1	0

Roberts filters

Horizontal:

1	2	1
0	0	0
-1	-2	-1

Vertical:

1	0	-1
2	0	-2
1	0	-1

Diagonal:

2	1	0
1	0	-1
0	-1	-2

Sobel filters

Example filter - horizontal edge detection

Horizontal edge detection filter

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

$*$

Image

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

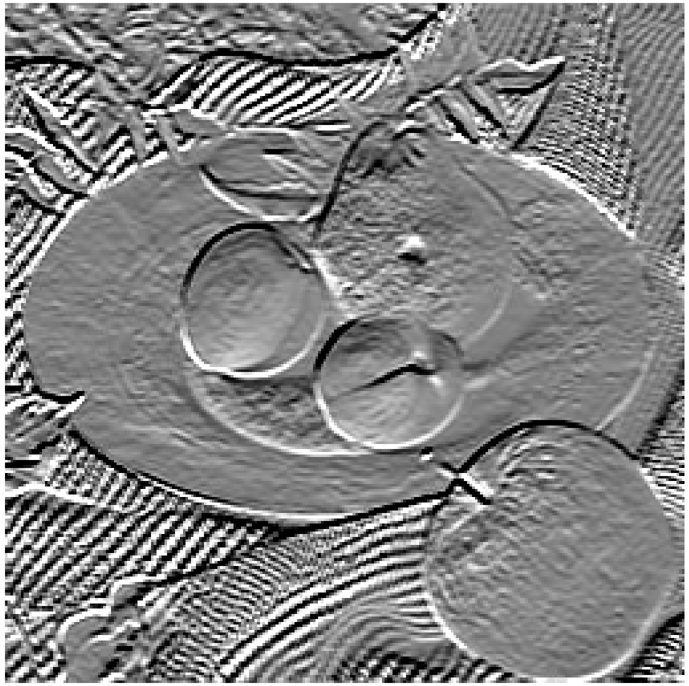
$=$

Result

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

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

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

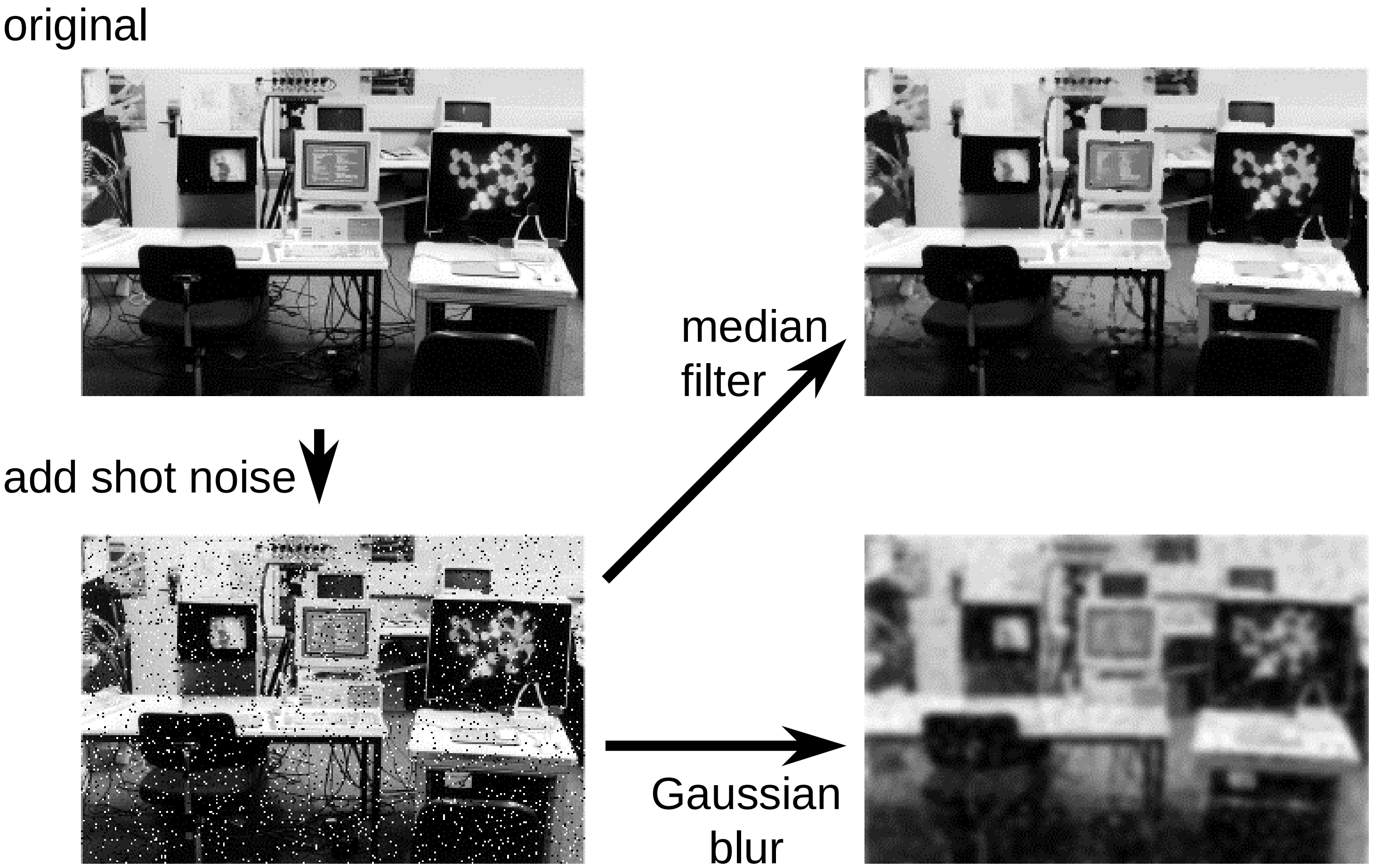
10	15	17	21	24	27
12	16	20	25	99	37
15	22	23	25	38	42
18	37	36	39	40	44
34	2	40	41	43	47

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

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

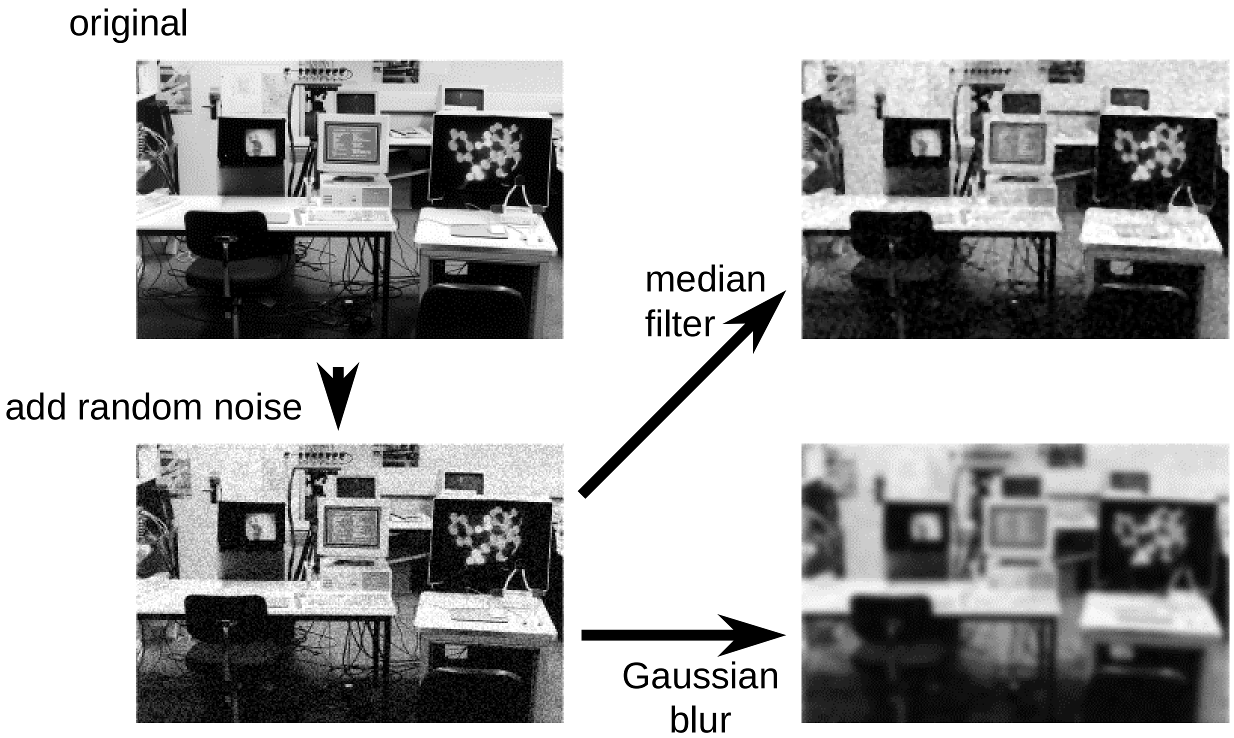
sort into order and take median

Median filter - example



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

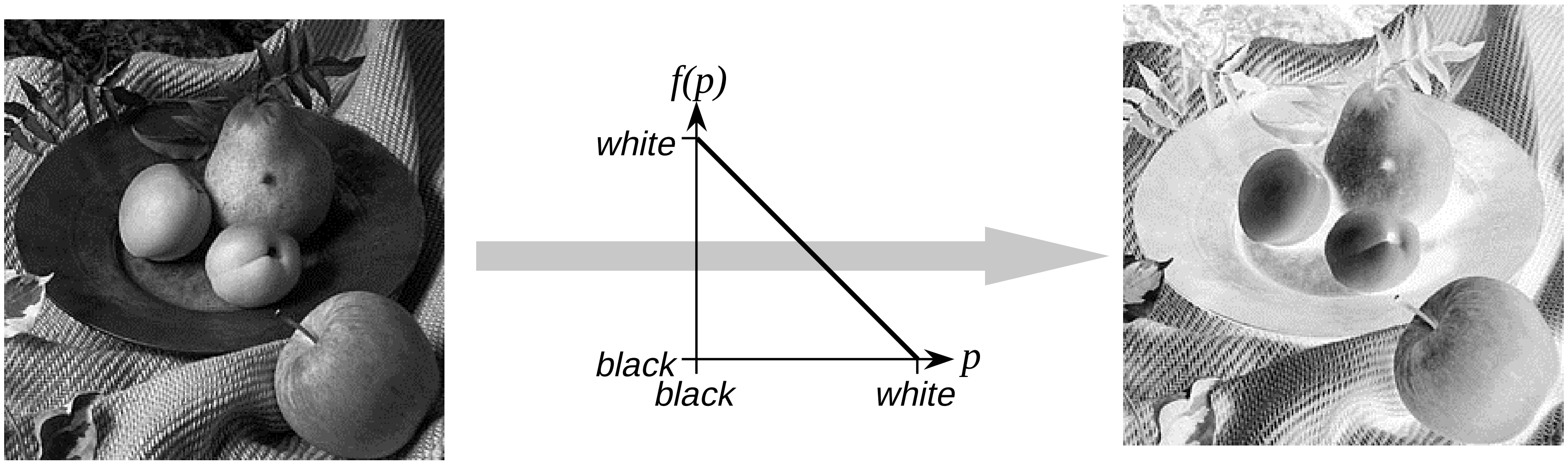
Point processing

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

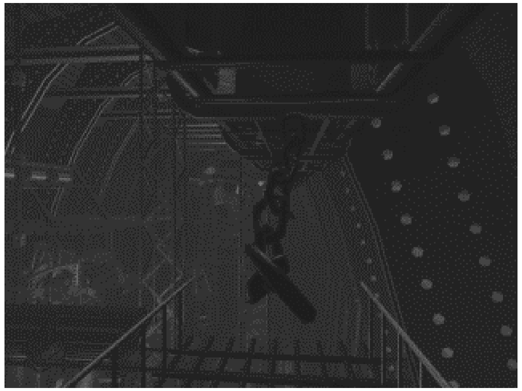
$$p'(i,j) = f\{p(i,j)\}$$

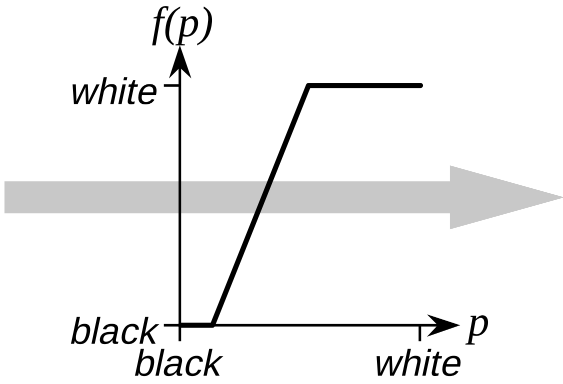
  - where $p(i,j)$ is the value of the pixel and $p'(i,j)$ is the modified value
  - the modification function, $f(p)$, can perform any operation that maps one intensity value to another

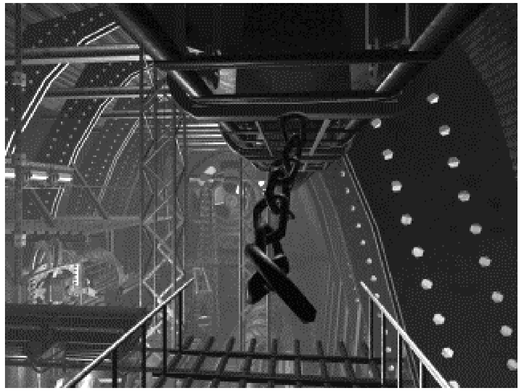
Point processing
inverting an image



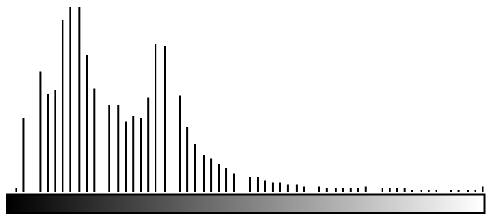
Point processing
improving an image’s contrast







dark histogram



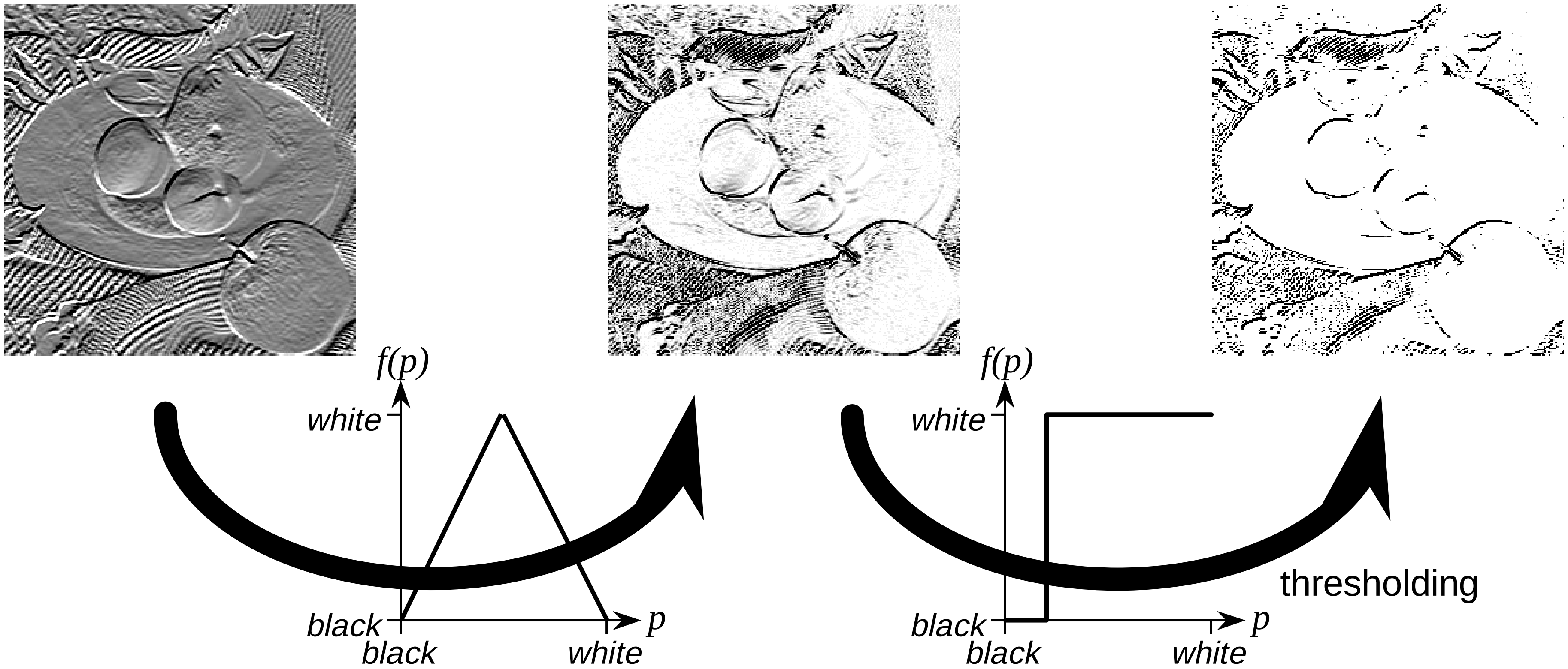
improved histogram

Point processing
modifying the output of a filter

black or white = edge
mid-grey = no edge

black = edge
white = no edge
grey = indeterminate

black = edge
white = no edge



Point processing: gamma correction

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

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

- the voltage is directly related to the pixel value:

$$V \propto p$$

- gamma correction modifies pixel values in the inverse manner:

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

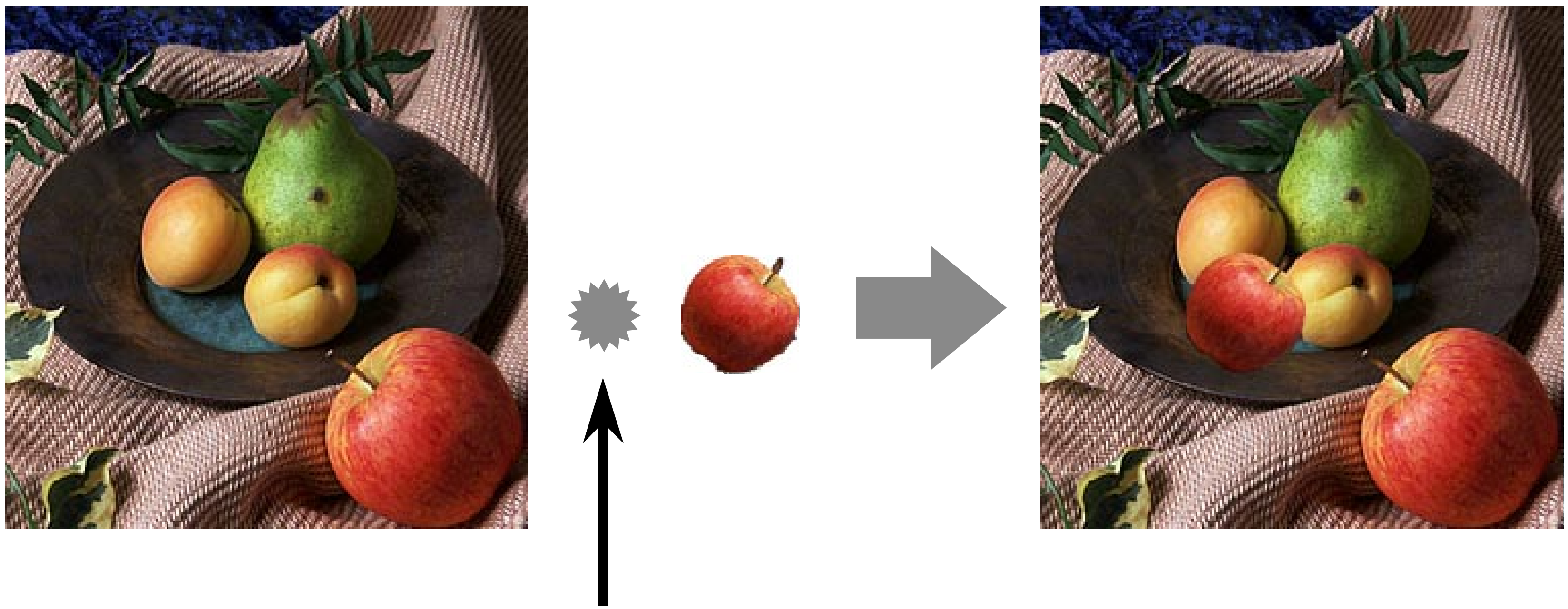
- thus generating the appropriate intensity on the CRT:

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

- CRTs generally have gamma values around 2.0

Image compositing

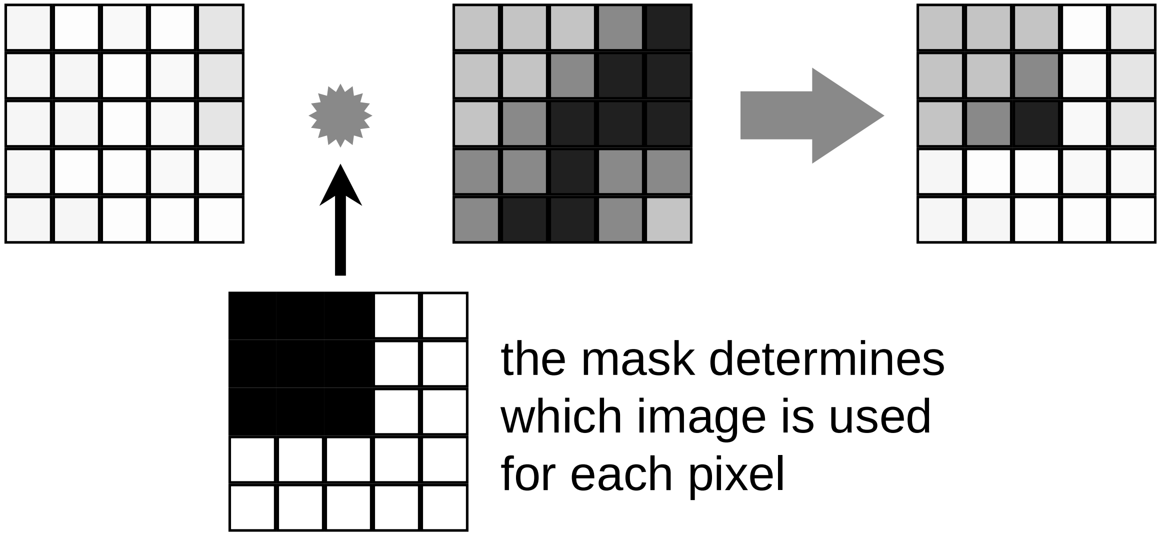
- merging two or more images together



what does this operator do?

Simple compositing

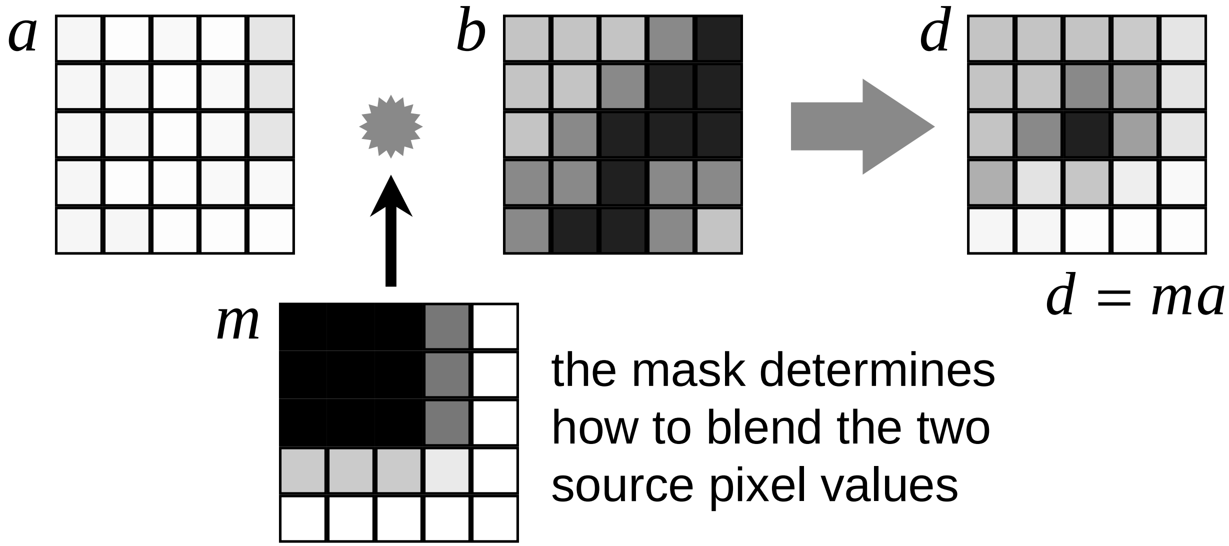
- copy pixels from one image to another
  - only copying the pixels you want
  - use a mask to specify the desired pixels



the mask determines
which image is used
for each pixel

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



$$d = ma + (1 - m)b$$

the mask determines how to blend the two source pixel values

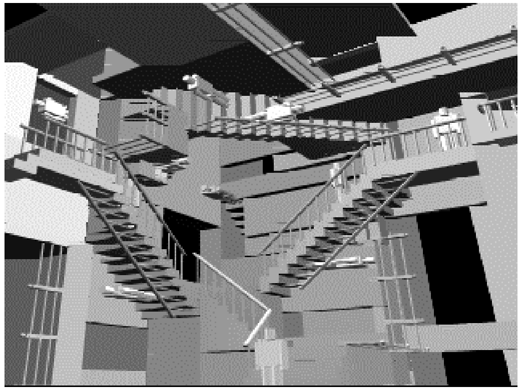
Arithmetic operations

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

Difference example

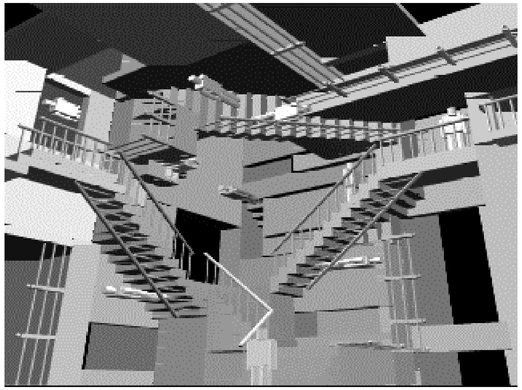
the two images are taken from slightly different viewpoints

a



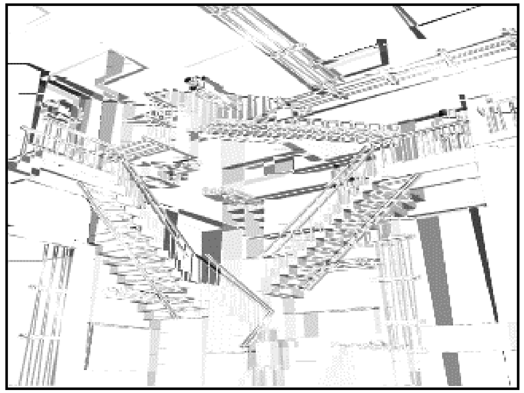
$\ominus$

b



$=$

d



take the difference between the two images

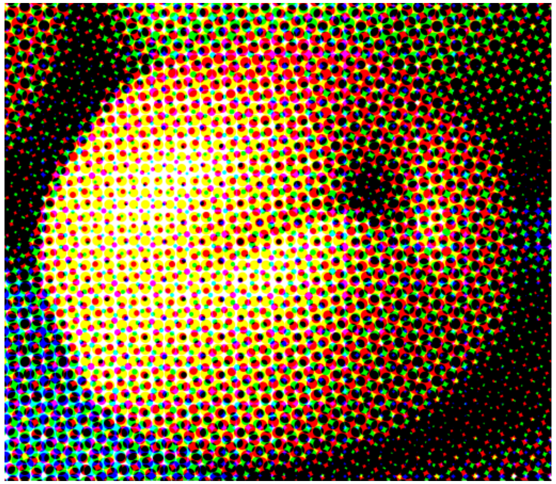
$$d = 1 - |a - b|$$

where 1 = white and 0 = black

black = large difference
white = no difference

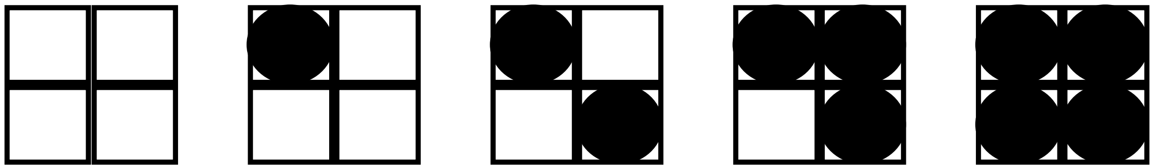
Halftoning & dithering

- 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



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

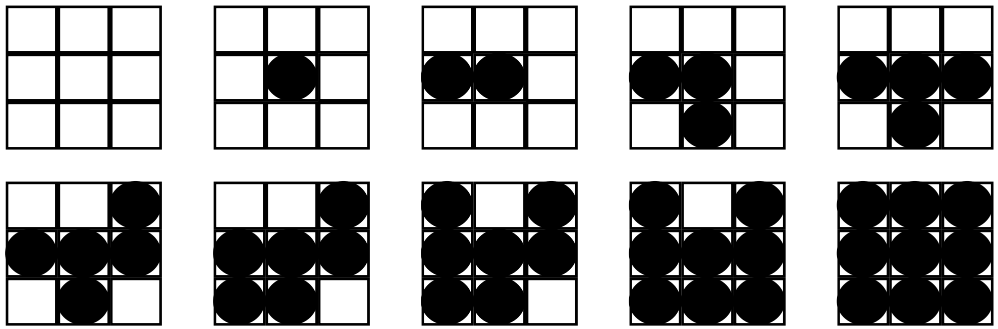


0-51	52-102	103-153	154-204	205-255

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

Halftoning dither matrix

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

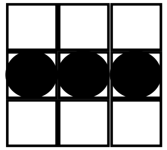


- ✦ these patterns can be represented by the *dither matrix*:

7	9	5
2	1	4
6	3	8

  - 1-to-9 pixel mapping

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)

Ordered dither

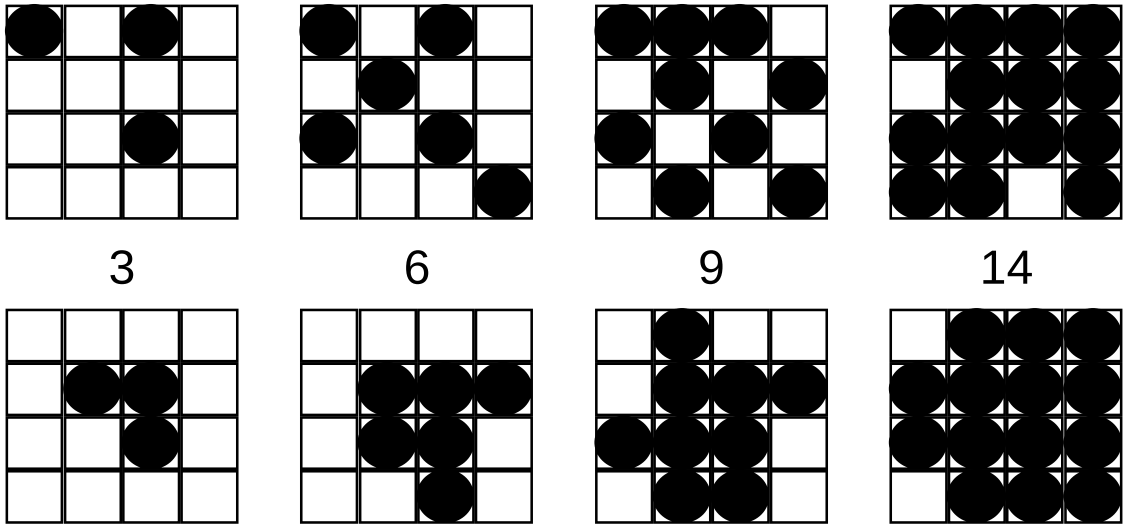
- **halftone prints and photocopies well, at the expense of large dots**
- **an ordered dither matrix produces a nicer visual result than a halftone dither matrix**

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



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

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

$$d_{m,n} \qquad \begin{array}{cc|c|c|c|c|} & & \multicolumn{4}{c}{m} \\ & & 0 & 1 & 2 & 3 \\ \hline & 0 & 1 & 9 & 3 & 11 \\ \hline & 1 & 15 & 5 & 13 & 7 \\ \hline n & 2 & 4 & 12 & 2 & 10 \\ \hline & 3 & 14 & 8 & 16 & 6 \\ \hline \end{array}$$

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

Error diffusion - example (1)

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

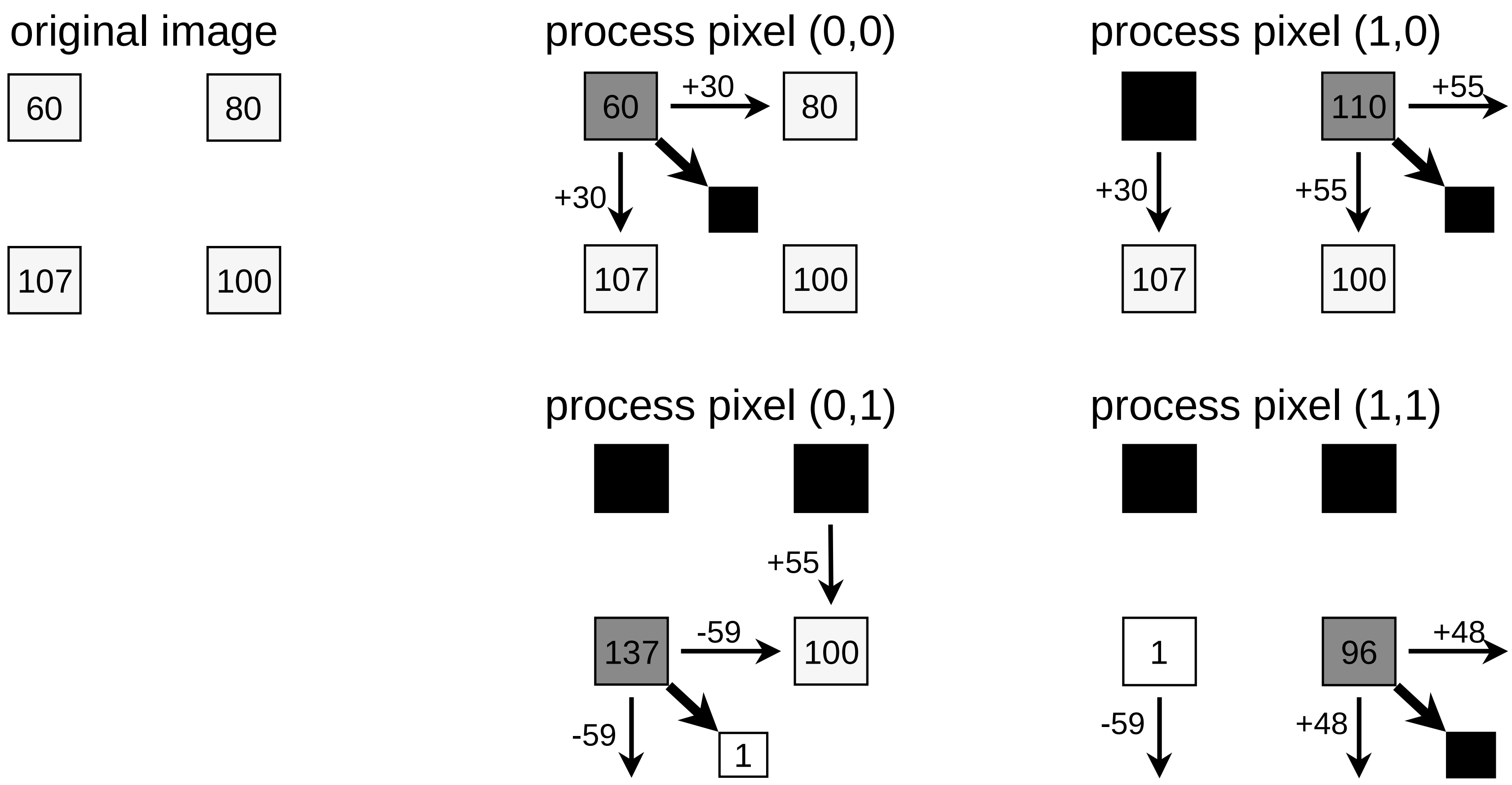
8-bit value $f_{i,j}$	1-bit value $b_{i,j}$	error $e_{i,j}$
0-127	0	$f_{i,j}$
128-255	1	$f_{i,j} - 255$

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

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

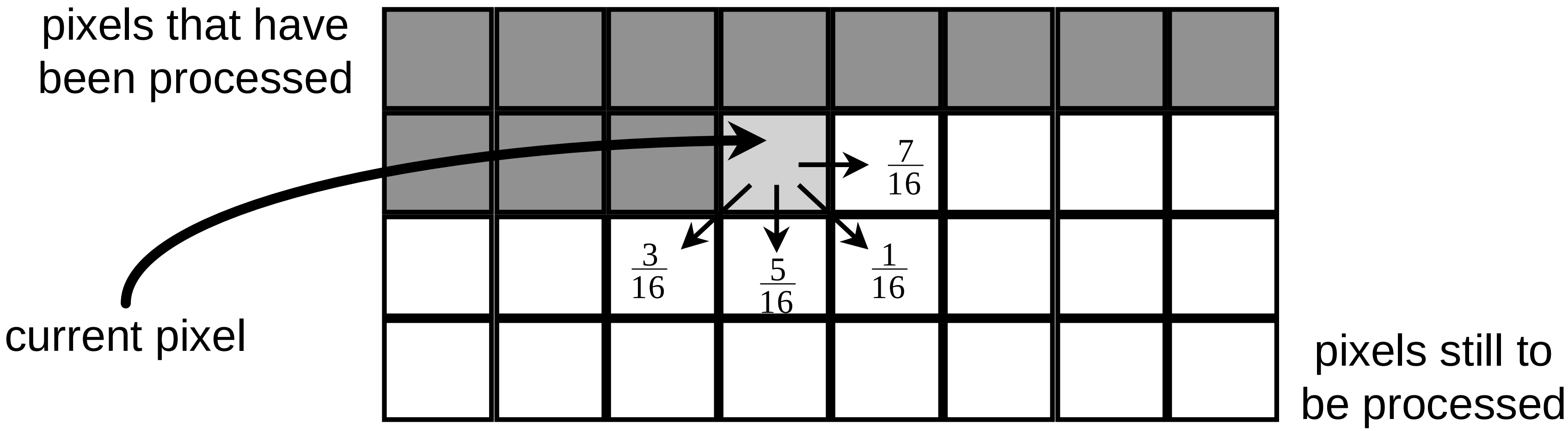
in this example the errors from the pixels to the left and above are taken into account

Error diffusion - example (2)



Error diffusion

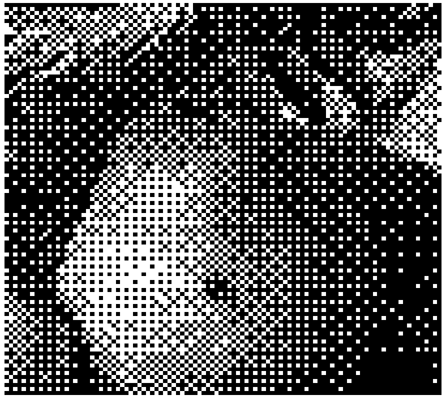
- Floyd & Steinberg developed the error diffusion method in 1975
  - often called the "Floyd-Steinberg algorithm"
- their original method diffused the errors in the following proportions:



Halftoning & dithering — examples

original

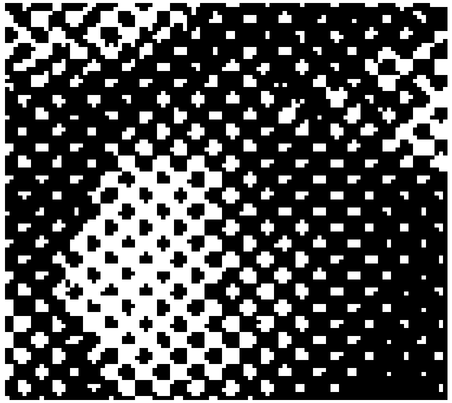
ordered dither



error diffused

thresholding

halftoning
(4×4 cells)



halftoning
(5×5 cells)

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

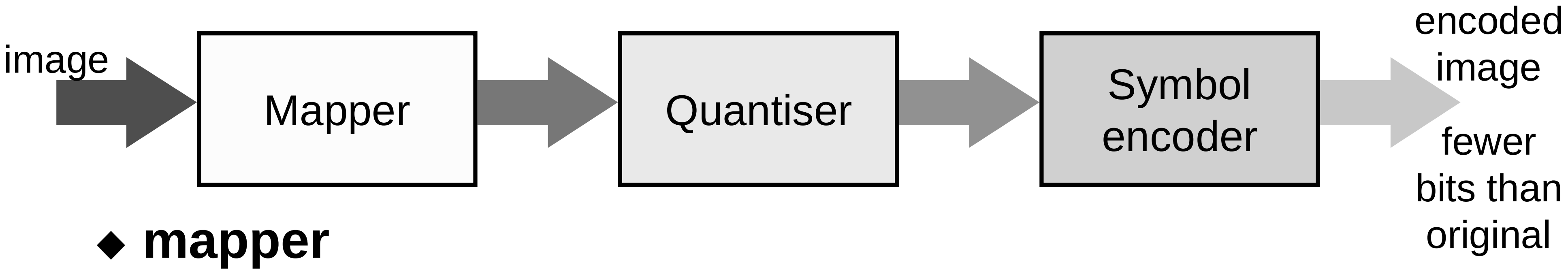
Encoding & compression

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

What you should note about image data

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

Encoding - overview



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

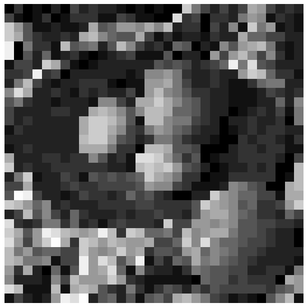
all three operations are optional

Lossless *vs* lossy compression

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

Raw image data

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



32✕32 pixels

```
5 54 5 18 5 30 16 69 43 58 40 33 18 13 16 3 16 9 7 189 119 69 44 60 42 68 161 149 70 37 48 35 57
56 12 15 64 41 21 14 4 3 218 57 64 6 54 57 46 118 149 140 32 45 39 24 199 156 81 16 12 29 12 15
130 168 124 174 38 59 50 9 65 29 128 22 192 125 147 29 38 22 198 170 78 42 41 43 43 46 163 188
27 57 24 40 24 21 43 37 44 163 110 100 74 51 39 31 232 20 121 50 55 10 186 77 111 112 40 86 186
81 7 32 18 136 78 151 159 187 114 35 18 29 233 3 86 35 87 26 42 52 14 13 13 31 50 73 20 18 22 81
152 186 137 80 131 47 19 47 24 66 72 29 194 161 63 17 9 8 29 33 33 38 31 27 81 74 74 66 38 48 65
66 42 26 36 51 55 77 229 61 65 11 28 32 41 35 36 28 24 34 138 130 150 109 56 37 30 45 38 41 157
44 110 176 71 36 30 25 41 44 47 60 20 11 19 16 155 156 165 125 69 39 38 48 38 22 18 49 107 119
43 32 44 30 26 45 44 39 33 37 63 22 148 178 141 121 76 55 44 42 25 13 17 21 39 70 47 25 57 93 121
39 11 128 137 61 41 168 170 195 168 135 102 83 48 39 33 19 16 23 33 42 95 43 121 71 34 39 40 38
168 137 78 143 182 189 160 109 104 87 57 36 35 6 16 34 41 36 63 26 118 75 37 41 34 33 31 39 33
95 21 181 197 134 125 109 66 46 31 3 33 38 42 33 38 46 12 109 25 41 36 34 36 34 34 37 174 202
148 132 101 79 58 41 32 0 11 26 53 46 45 48 38 42 42 38 32 37 36 37 40 30 183 201 201 152 92 67
41 24 15 4 7 43 43 41 50 45 10 44 17 37 41 37 33 31 33 33 172 180 168 112 54 55 11 182 179 159
48 39 48 46 12 25 162 39 37 28 44 49 43 41 58 130 85 40 49 14 212 218 202 162 98 60 75 8 11 27
195 40 45 34 41 48 61 48 42 61 53 35 30 35 178 212 182 206 155 80 70 30 6 14 39 36 53 43 45 8 6
35 59 49 31 79 73 78 62 81 108 195 175 156 112 60 53 6 11 22 42 49 51 48 49 3 16 184 77 83 156
63 80 65 73 84 157 142 126 77 51 9 12 27 32 142 109 89 56 8 6 169 178 80 240 231 71 36 30 28 35
90 55 42 2 3 37 37 192 155 129 101 106 72 65 19 157 168 195 192 157 110 132 39 40 38 35 38 42 51
48 41 89 197 174 144 138 98 92 56 45 69 161 199 46 65 187 79 131 64 41 96 46 38 37 42 47 44 56
165 173 142 103 81 59 58 41 96 78 204 54 42 52 125 118 45 102 39 55 17 57 62 45 60 46 39 188 69
135 81 84 72 60 43 47 40 209 158 83 154 232 211 186 162 156 167 223 190 58 201 175 101 104 124
162 118 89 81 63 48 39 33 12 209 162 71 152 210 250 176 58 201 191 147 188 160 147 147 166 79
137 110 101 83 70 70 48 34 37 2 182 121 157 83 101 104 76 65 194 155 136 156 202 162 173 64 84
130 123 106 77 63 49 37 39 36 26 189 165 119 123 131 24 70 85 229 154 215 176 92 141 223 20 73
99 83 71 49 35 36 30 30 23 151 58 169 33 12 99 22 76 234 156 180 219 108 30 128 59 26 27 26 47
45 38 52 55 11 112 128 40 35 40 21 126 65 179 162 156 158 201 145 44 35 18 27 14 21 23 0 101 78
162 155 220 174 27 17 20 173 29 160 187 172 93 59 46 121 57 14 50 76 69 31 78 56 82 76 64 66 66
69 26 20 33 160 235 224 253 29 84 102 25 78 22 81 103 78 158 192 148 125 68 53 30 29 23 18 82
```

1024 bytes

Symbol encoding on raw data

(an example of symbol encoding)

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

e.g.

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

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

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

Quantisation as a compression method

(an example of quantisation)

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

Difference mapping

(an example of mapping)

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

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

Difference mapping - example (1)

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

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

Difference mapping - example (2)

(an example of mapping and symbol encoding combined)

- **this is a very simple variable length code**

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

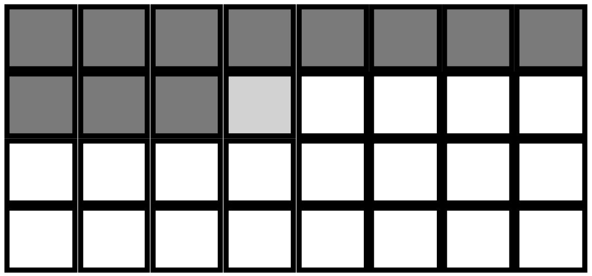
7.29 bits/pixel

91% of the space of the original image

Predictive mapping

(an example of mapping)

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



e.g. prediction

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

difference - this is what we transmit

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

Run-length encoding

(an example of symbol encoding)

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

original pixels

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

run-length encoding

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

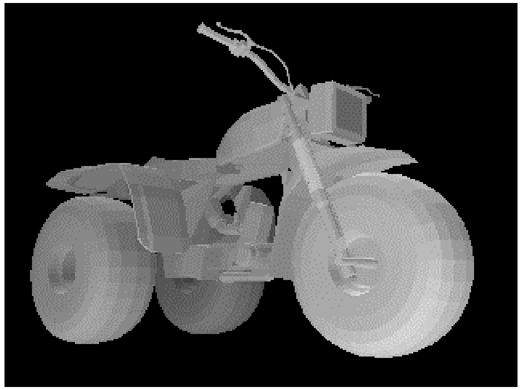
Run-length encoding - example (1)

- **run length is encoded as an 8-bit value:**
  - **first bit determines type of run**
    - 0 = identical pixels, 1 = non-identical pixels
  - **other seven bits code length of run**
    - binary value of *run length* - 1 (*run length* $\in \{1,\ldots,128\}$)
- **pixels are encoded as 8-bit values**

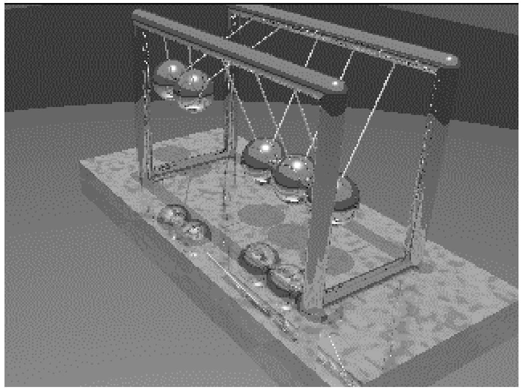
- **best case: all runs of 128 identical pixels**
  - **compression of 2/128 = 1.56%**
- **worst case: no runs of identical pixels**
  - **compression of 129/128=100.78% (expansion!)**

Run-length encoding - example (2)

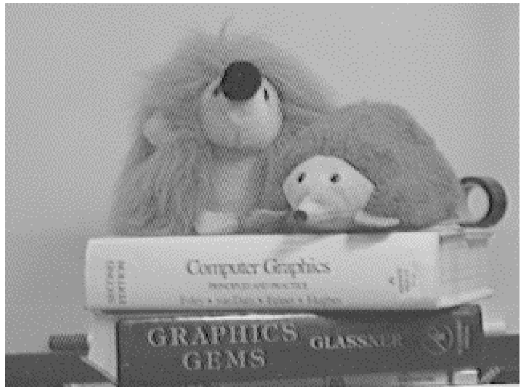
- works well for computer generated imagery
- not so good for real-life imagery
- especially bad for noisy images



19.37%



44.06%



99.76%

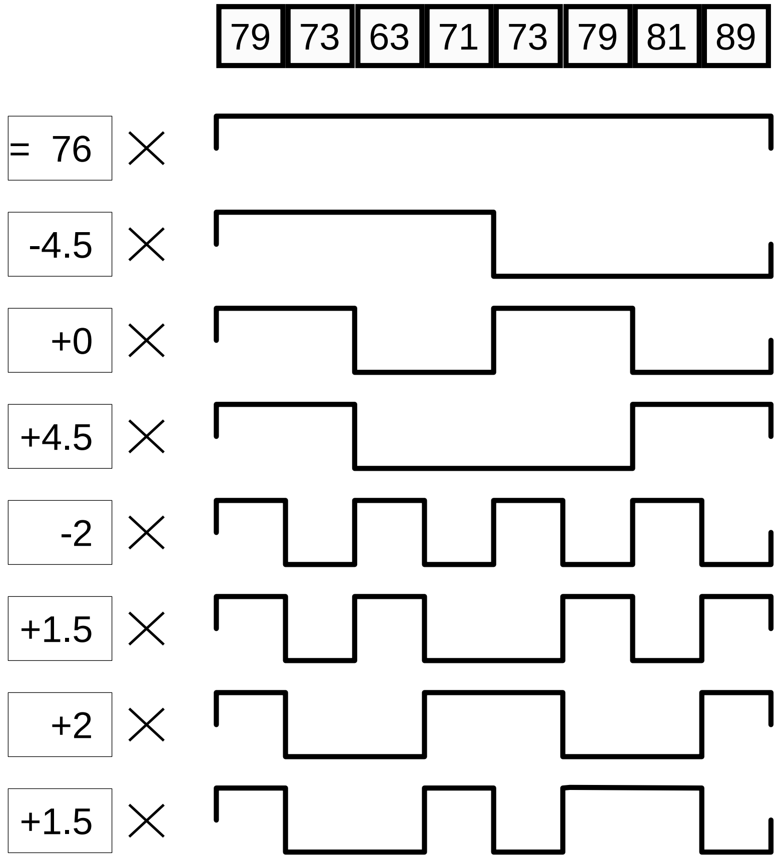
compression ratios

CCITT fax encoding

- fax images are binary
- 1D CCITT group 3
  - binary image is stored as a series of run lengths
  - don’t need to store pixel values!
- 2D CCITT group 3 & 4
  - predict this line’s runs based on previous line’s runs
  - encode differences

Transform coding

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



Mathematical foundations

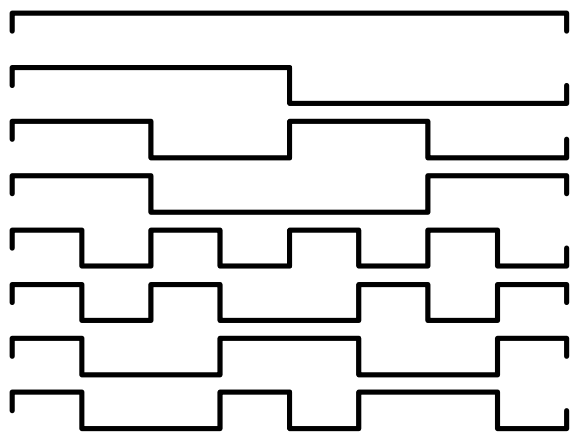
- each of the N pixels, $f(x)$, is represented as a weighted sum of coefficients, $F(u)$

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

$H(u,x)$ is the array of weights

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

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



Calculating the coefficients

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

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

forward transform

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

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

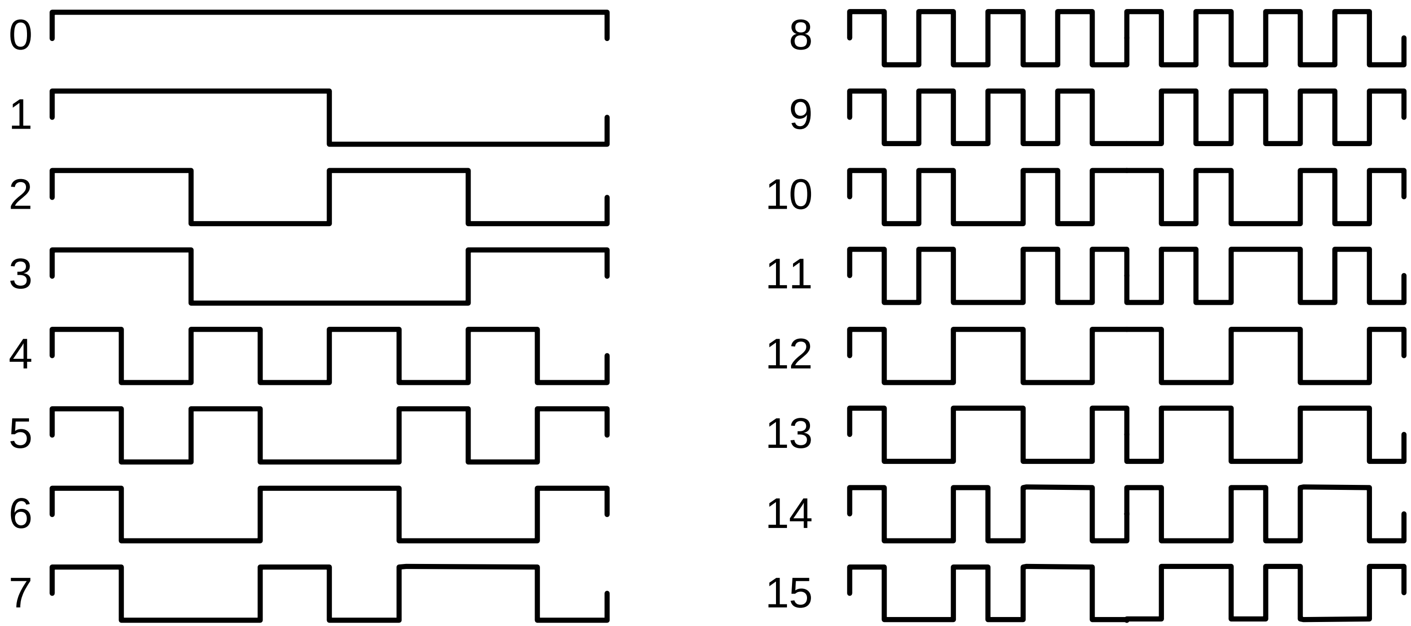
inverse transform

Walsh-Hadamard transform

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

the first sixteen Walsh basis functions

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



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

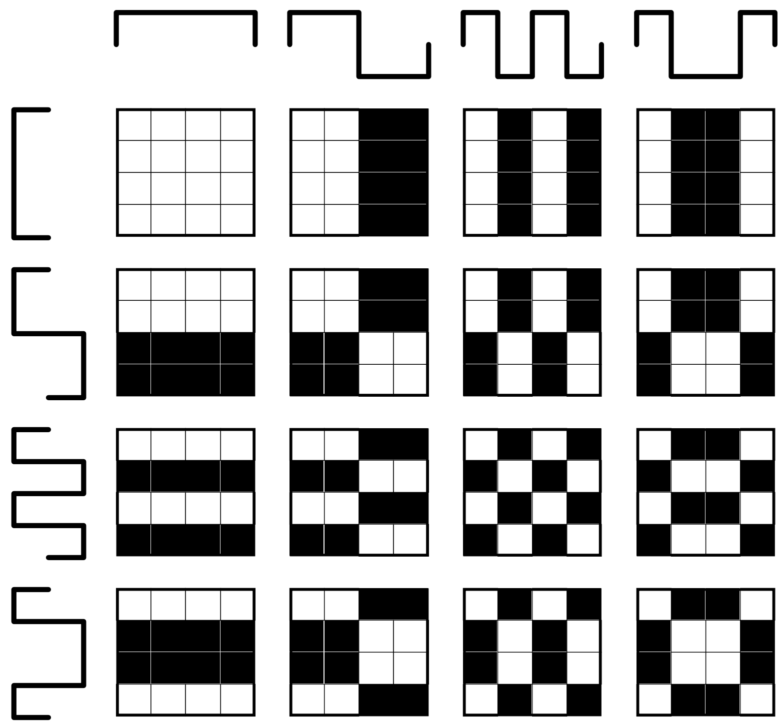
2D transforms

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

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

2D Walsh basis functions

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



Discrete Fourier transform (DFT)

- **forward transform:**

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

- **inverse transform:**

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

  - **thus:**

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

$$H(u,x) = e^{i2\pi xu/N}$$

DFT - alternative interpretation

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

$$f(x) = \sum_{u=0}^{\frac{N}{2}-1} A(u)\cos(2\pi ux + \theta(u))$$

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

Discrete cosine transform (DCT)

- **forward transform:**

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

- **inverse transform:**

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

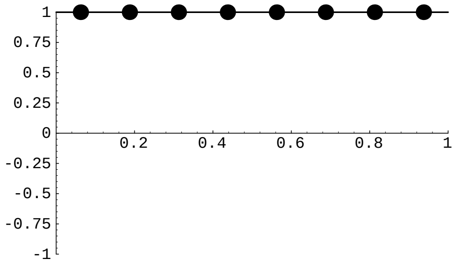
where:

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

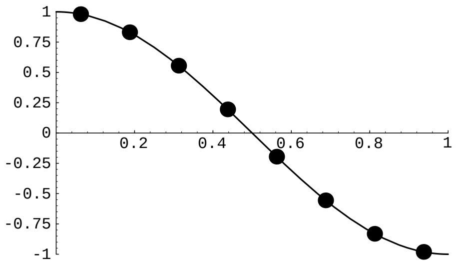
DCT basis functions

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

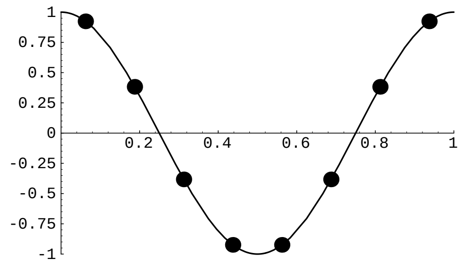
0



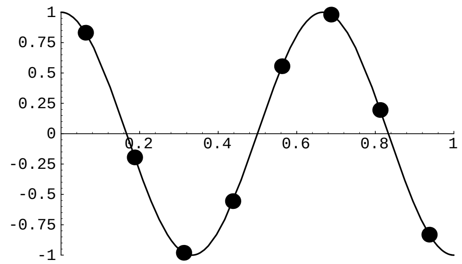
1



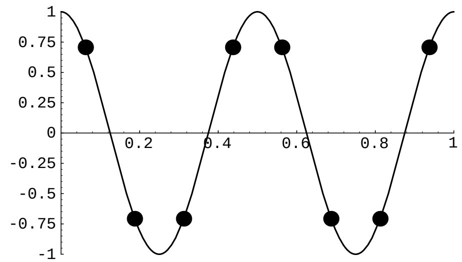
2



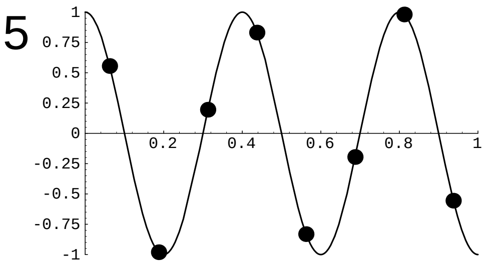
3



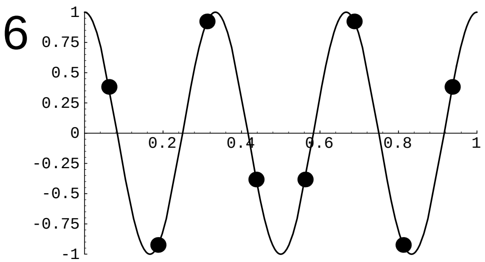
4



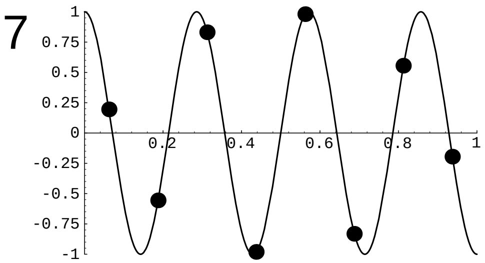
5



6



7

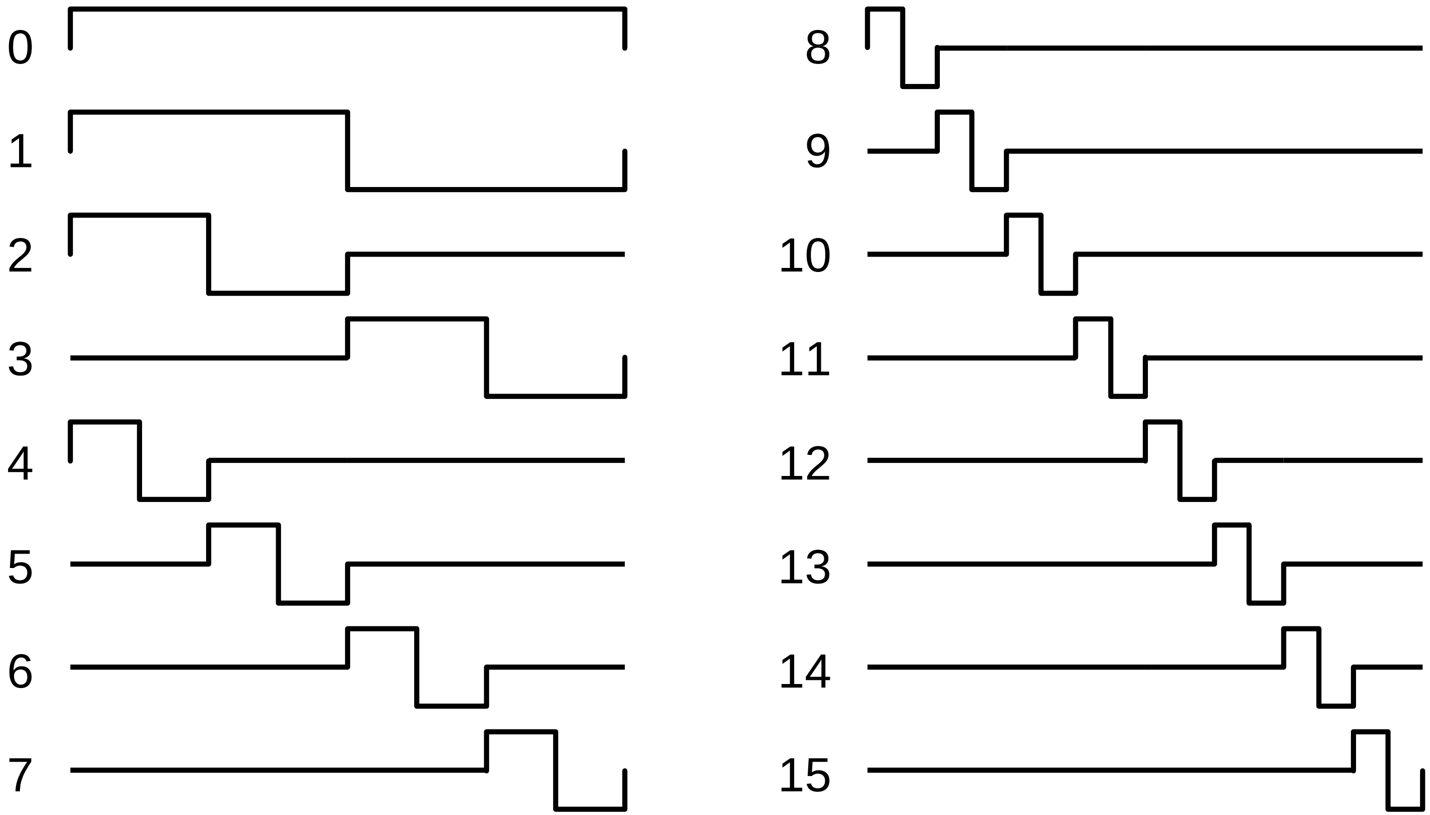


Haar transform: wavelets

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

Haar basis functions

the first sixteen Haar basis functions



Karhunen-Loève transform (KLT)

“eigenvector”, “principal component”, “Hotelling” transform

- ✦ **based on statistical properties of the image source**
- ✦ **theoretically best transform encoding method**
- ✦ **but different basis functions for every different image source**

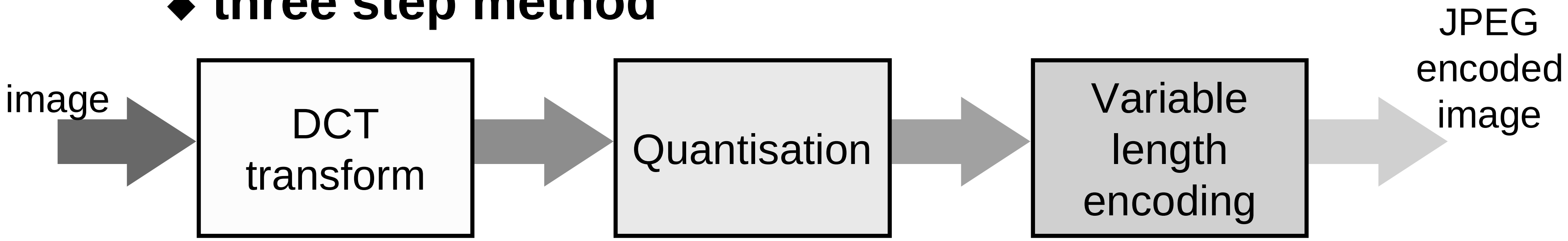
first derived by Hotelling (1933) for discrete data; by Karhunen (1947) and Loève (1948) for continuous data

JPEG: a practical example

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

JPEG sequential baseline scheme

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

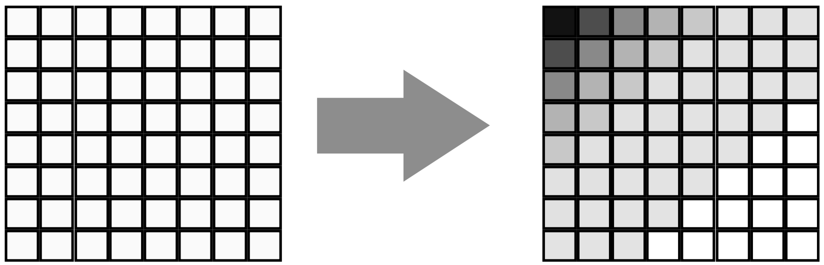


the following slides describe the steps involved in the JPEG compression of an 8 bit/pixel image

JPEG example: DCT transform

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

image



2D DCT

the most important coefficients are in the top left hand corner

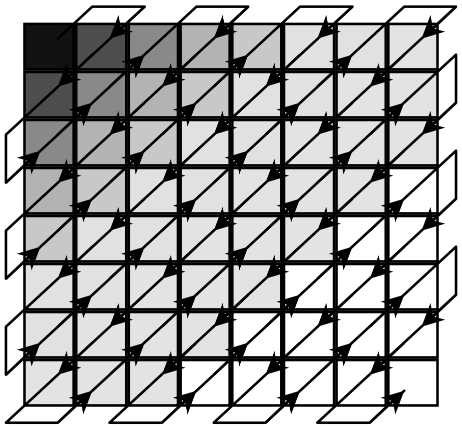
JPEG example: quantisation

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

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

$Z(u,v)$

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



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

JPEG example: symbol encoding

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