University of Cambridge
Computer Laboratory

INTRODUCTION TO COMPUTER GRAPHICS

COMPUTER SCIENCE TRIPOS PART IA

PETER ROBINSON & RAFAŁ MANTIUK

MICHAELMAS TERM 2016

William Gates Building
15 JJ Thomson Avenue
Cambridge CB3 0FD

www.cl.cam.ac.uk

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

Introduction to Computer Graphics

Peter Robinson & Rafał Mantiuk

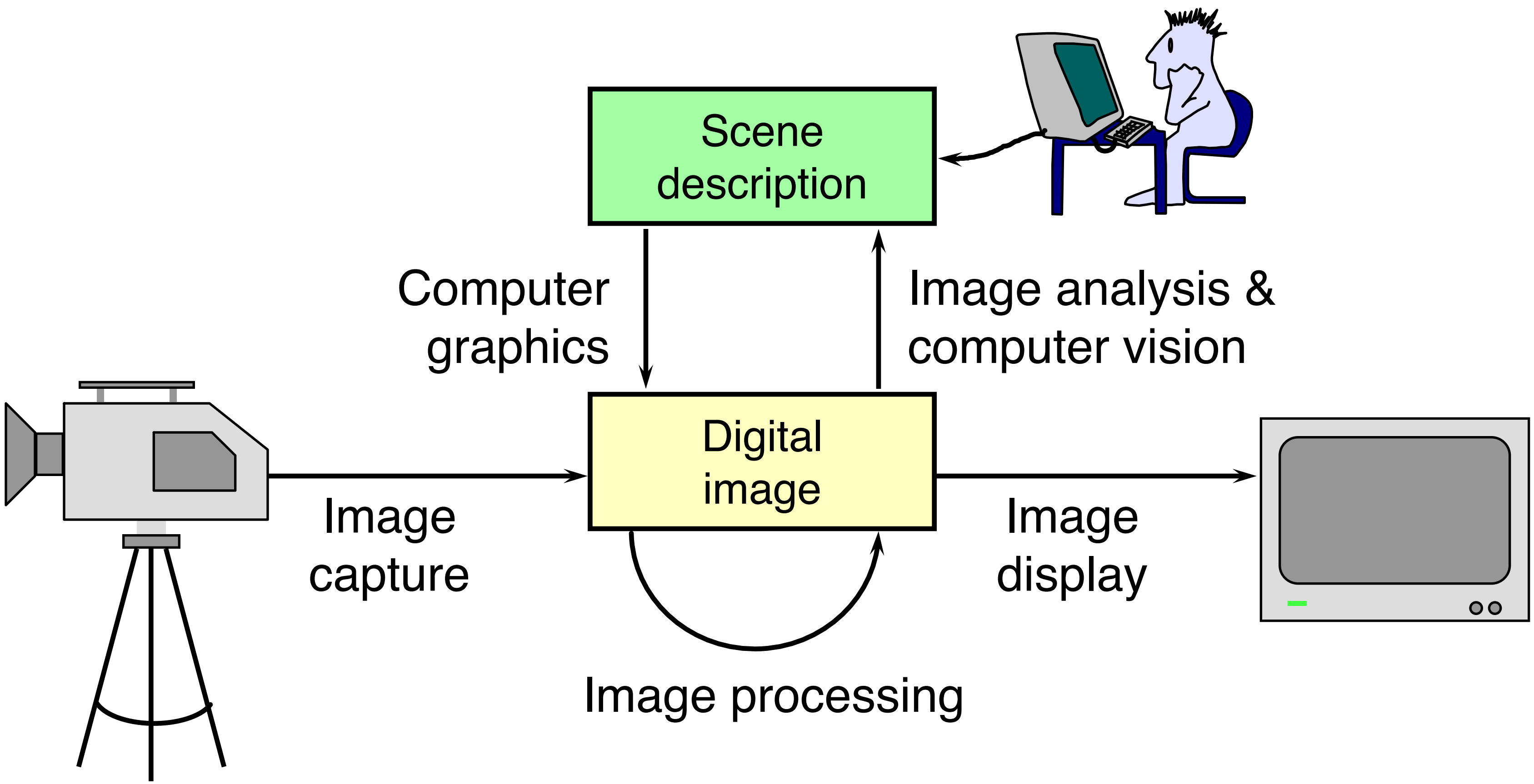
www.cl.cam.ac.uk/~pr & ~rkm38

Eight lectures & three practical classes for Part IA CST

Two supervisions suggested

Two exam questions on Paper 3

What are Computer Graphics & Image Processing?



Why bother with CG & IP?

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

What are CG & IP used for?

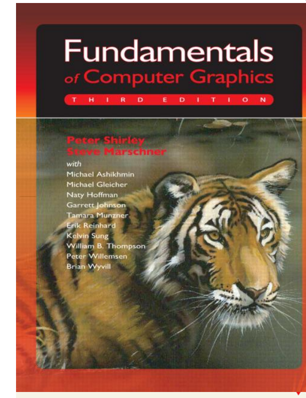
- 2D computer graphics
  - graphical user interfaces: Mac, Windows, X…
  - graphic design: posters, cereal packets…
  - typesetting: book publishing, report writing…
- Image processing
  - photograph retouching: publishing, posters…
  - photocollaging: satellite imagery…
  - art: new forms of artwork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

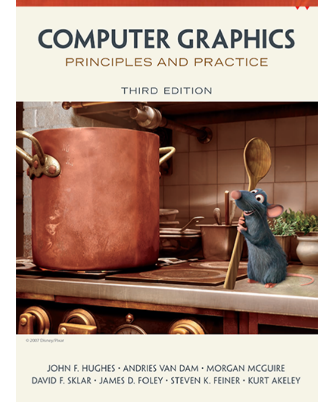
Course Structure

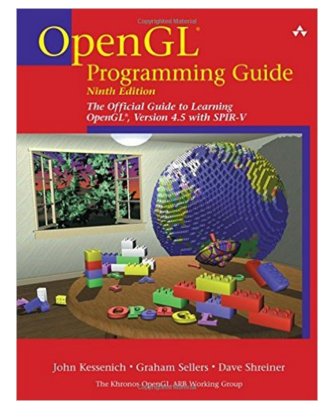
- **Background**
  - What is an image? Human vision. Resolution and quantisation. Storage of images in memory. [1 lecture]
- **Rendering**
  - Perspective. Reflection of light from surfaces and shading. Geometric models. Ray tracing. [3 lectures]
- **Graphics pipeline**
  - Polygonal mesh models. Transformations using matrices in 2D and 3D. Homogeneous coordinates. Projection: orthographic and perspective. [1 lecture]
- **Graphics hardware and modern OpenGL**
  - Vertex processing. Rasterisation. Fragment processing. Working with meshes and textures. [2 lectures]
- **Technology**
  - Colour spaces. Output devices: brief overview of display and printer technologies. [1 lecture]

Course books

- *Fundamentals of Computer Graphics*
  - Shirley & Marschner
    CRC Press 2015 (4th edition)
- *Computer Graphics: Principles & Practice*
  - Hughes, van Dam, McGuire, Sklar et al.
    Addison-Wesley 2013 (3rd edition)
- *OpenGL Programming Guide:*
  The Official Guide to Learning OpenGL Version 4.5 with SPIR-V
  - Kessenich, Sellers & Shreiner
    Addison Wesley 2016 (7th edition and later)







Introduction to Computer Graphics

- **Background**
  - What is an image?
  - Human vision
  - Resolution and quantisation
  - Storage of images in memory
- **Rendering**
- **Graphics pipeline**
- **Graphics hardware and modern OpenGL**
- **Technology**

What is an image?

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

What is a *digital* image?

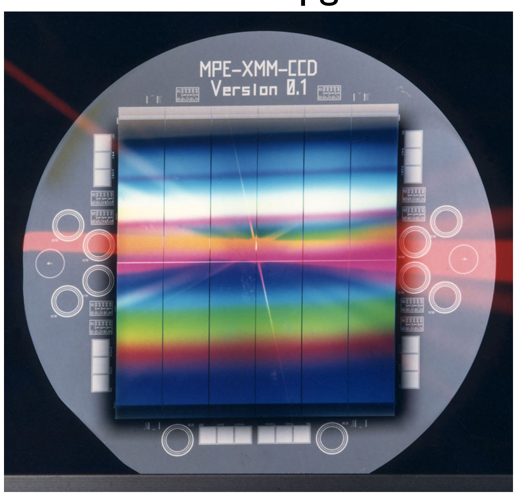
- a contradiction in terms
  - if you can see it, it's not digital
  - if it's digital, it's just a collection of numbers
- a sampled and quantised version of a real image
- a rectangular array of intensity or colour values

Image capture

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

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

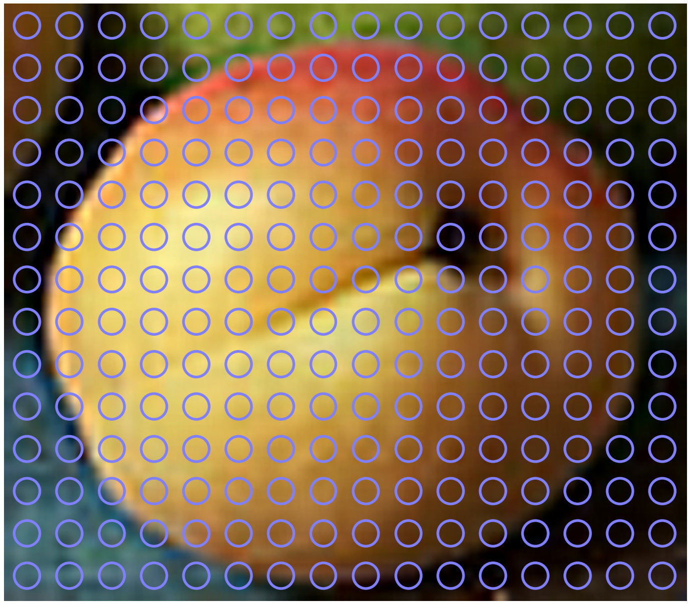
area CCD
www.hll.mpg.de



flatbed scanner
www.nuggetlab.com

Heidelberg
drum scanner

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

Sampling

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

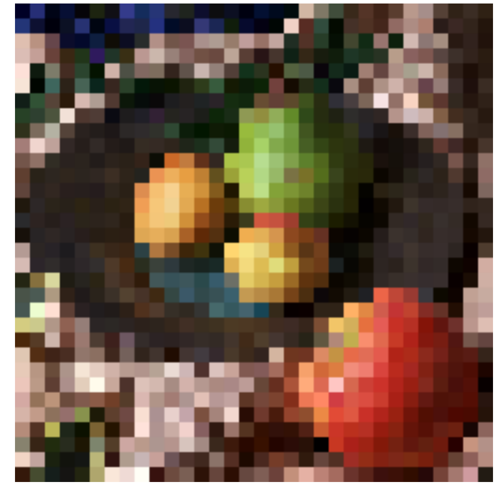
Sampling resolution

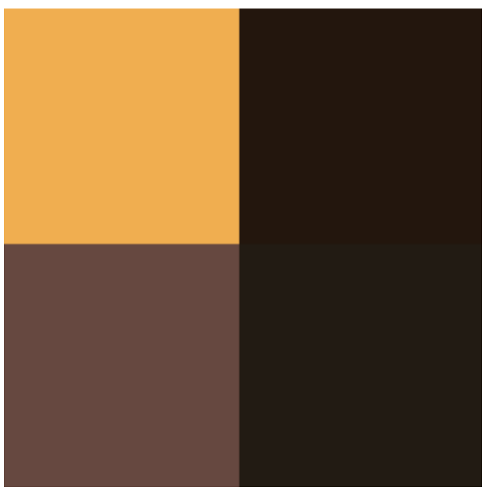
256×256

128×128

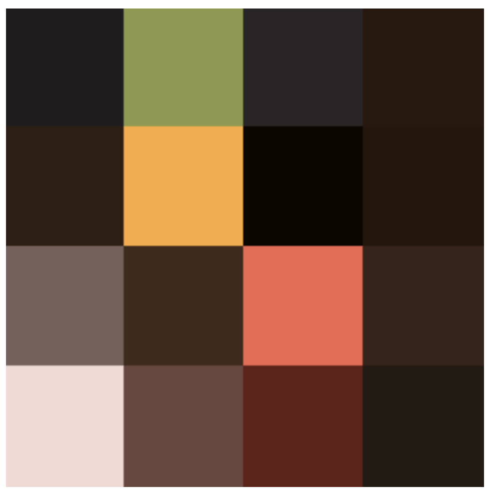
64×64

32×32



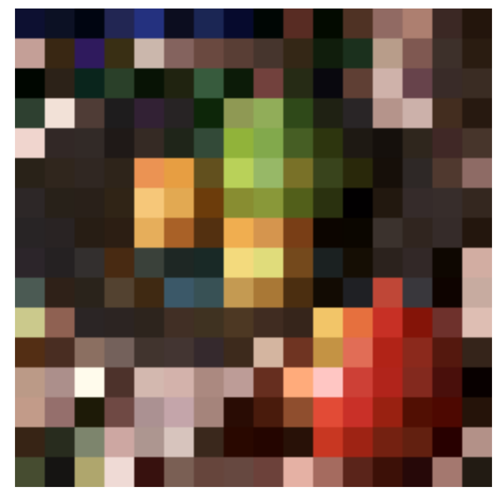


2×2



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 often sufficient
  - some applications use 10 or 12 or 16 bits
  - more detail later in the course
- colour is stored as a set of numbers
  - usually as 3 numbers of 5–16 bits each
  - more detail later in the course

Quantisation levels

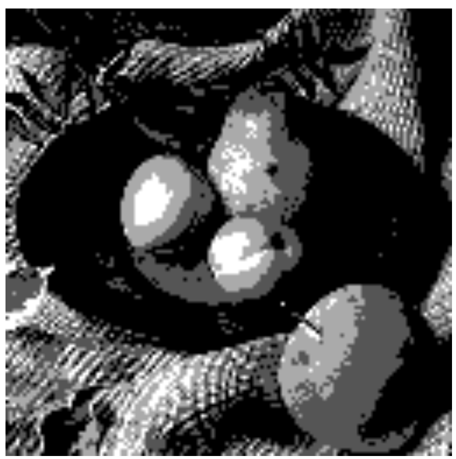
8 bits
(256 levels)

7 bits
(128 levels)

6 bits
(64 levels)

5 bits
(32 levels)

1 bit
(2 levels)



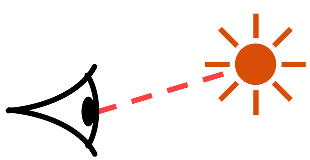
2 bits
(4 levels)

3 bits
(8 levels)

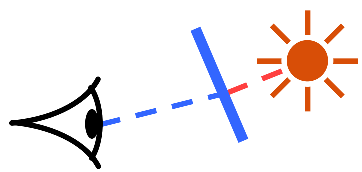
4 bits
(16 levels)

What is required for vision?

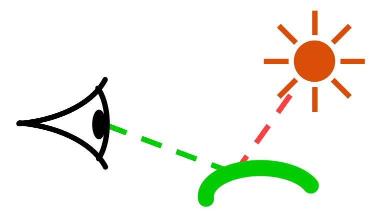
- illumination
  - some source of light
- objects
  - which reflect (or transmit) the light
- eyes
  - to capture the light as an image



direct viewing

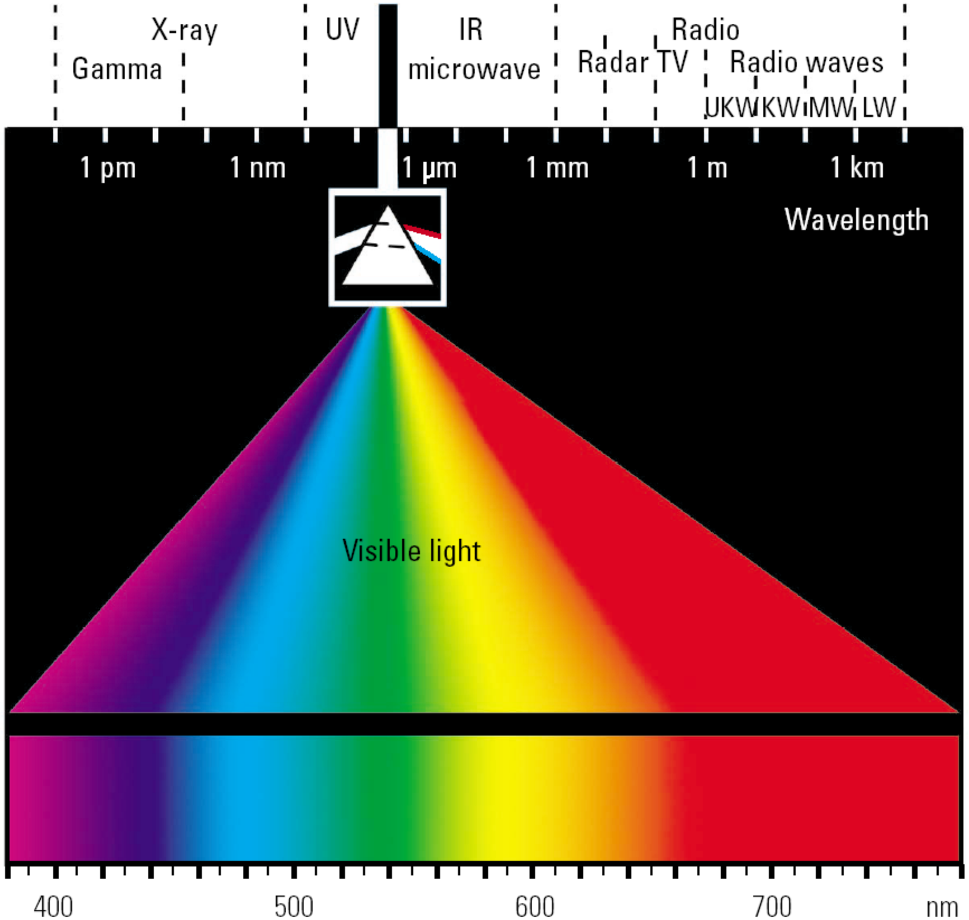


transmission



reflection

The spectrum



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

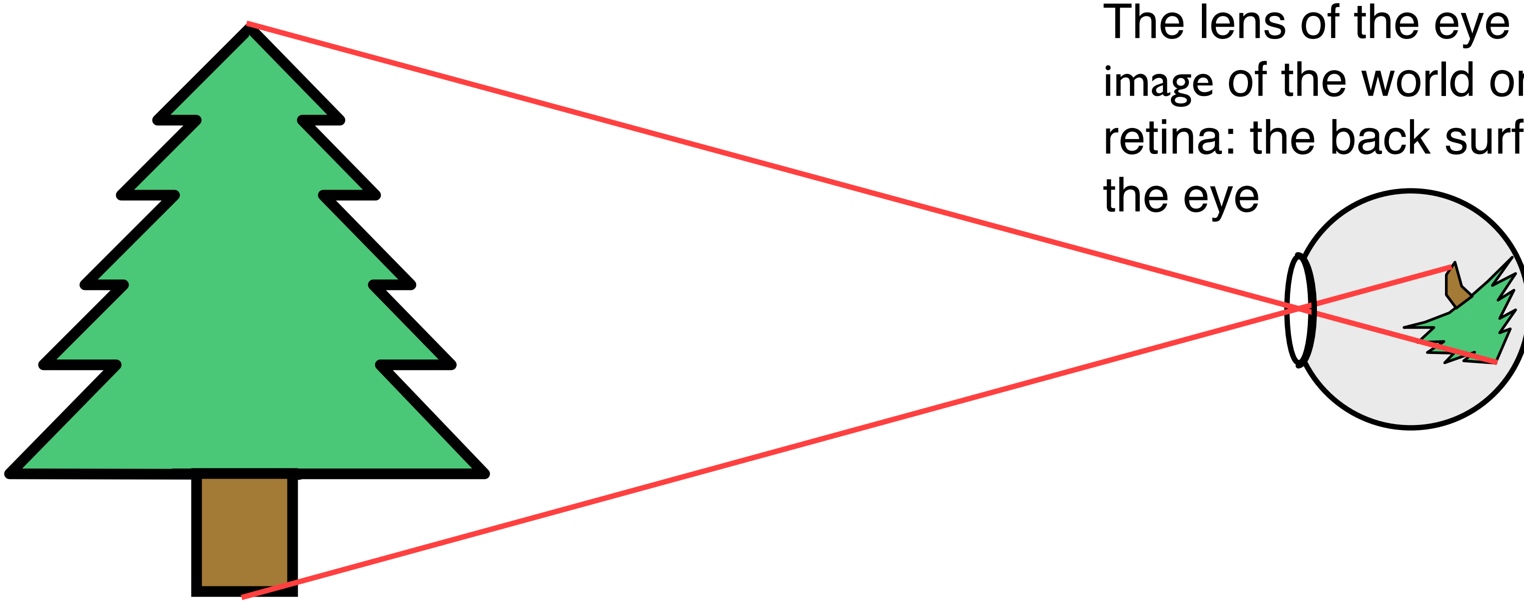
the short wavelength end of the spectrum is violet

the long wavelength end of the spectrum is red

violet blue green yellow red

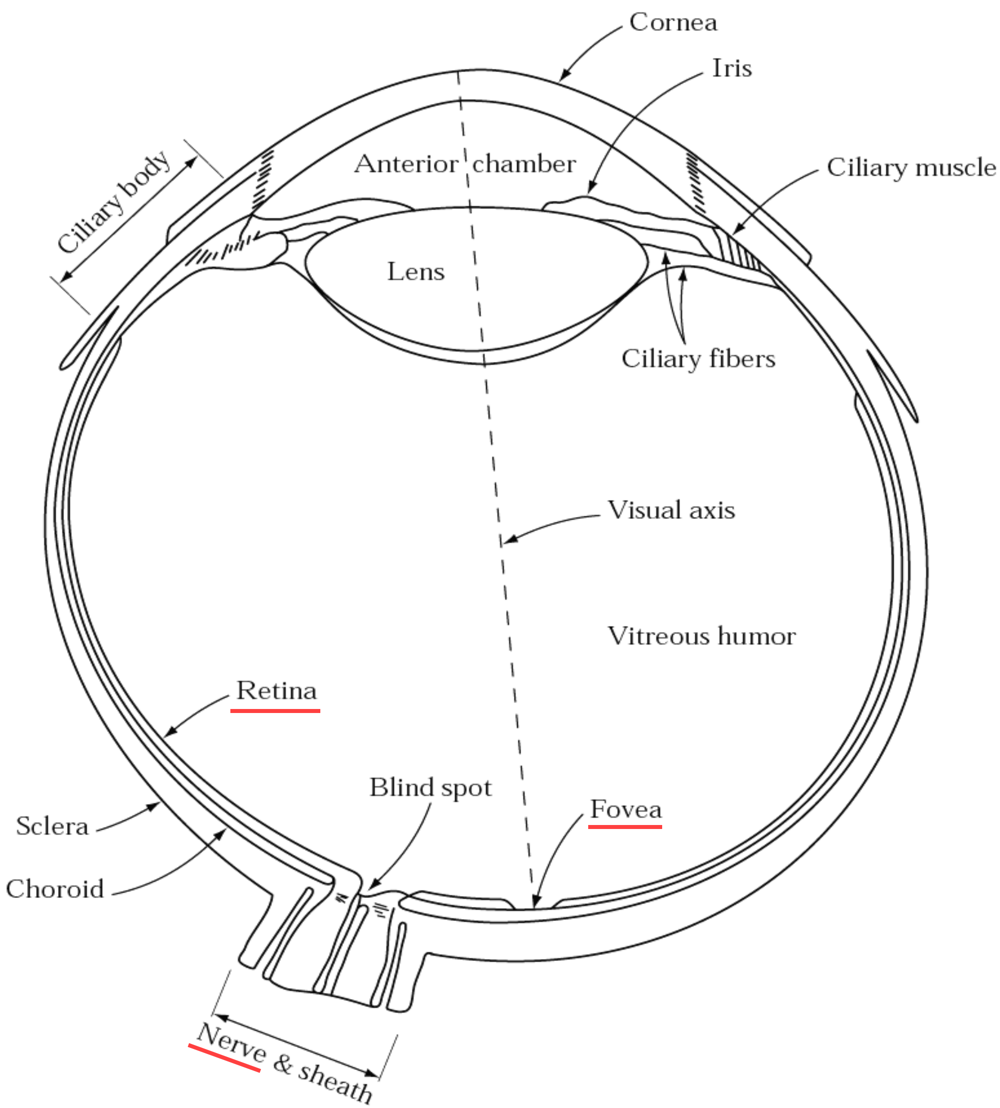
The workings of the human visual system

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



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

Structure of the human eye

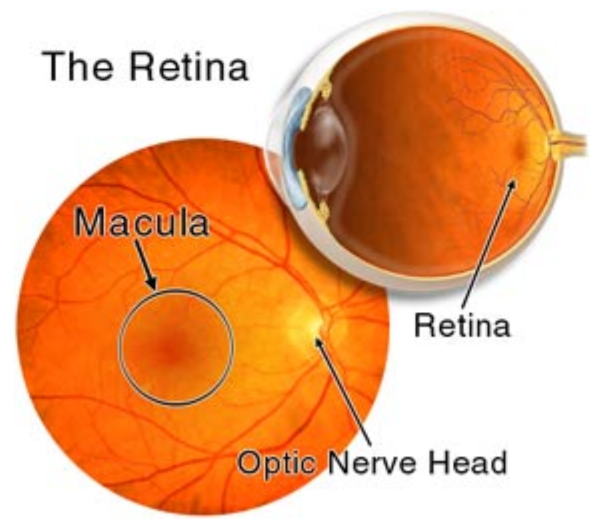


- the retina is an array of light detection cells
- the fovea is the high resolution area of the retina
- the optic nerve takes signals from the retina to the visual cortex in the brain

Fig. 2.1 from Gonzalez & Woods

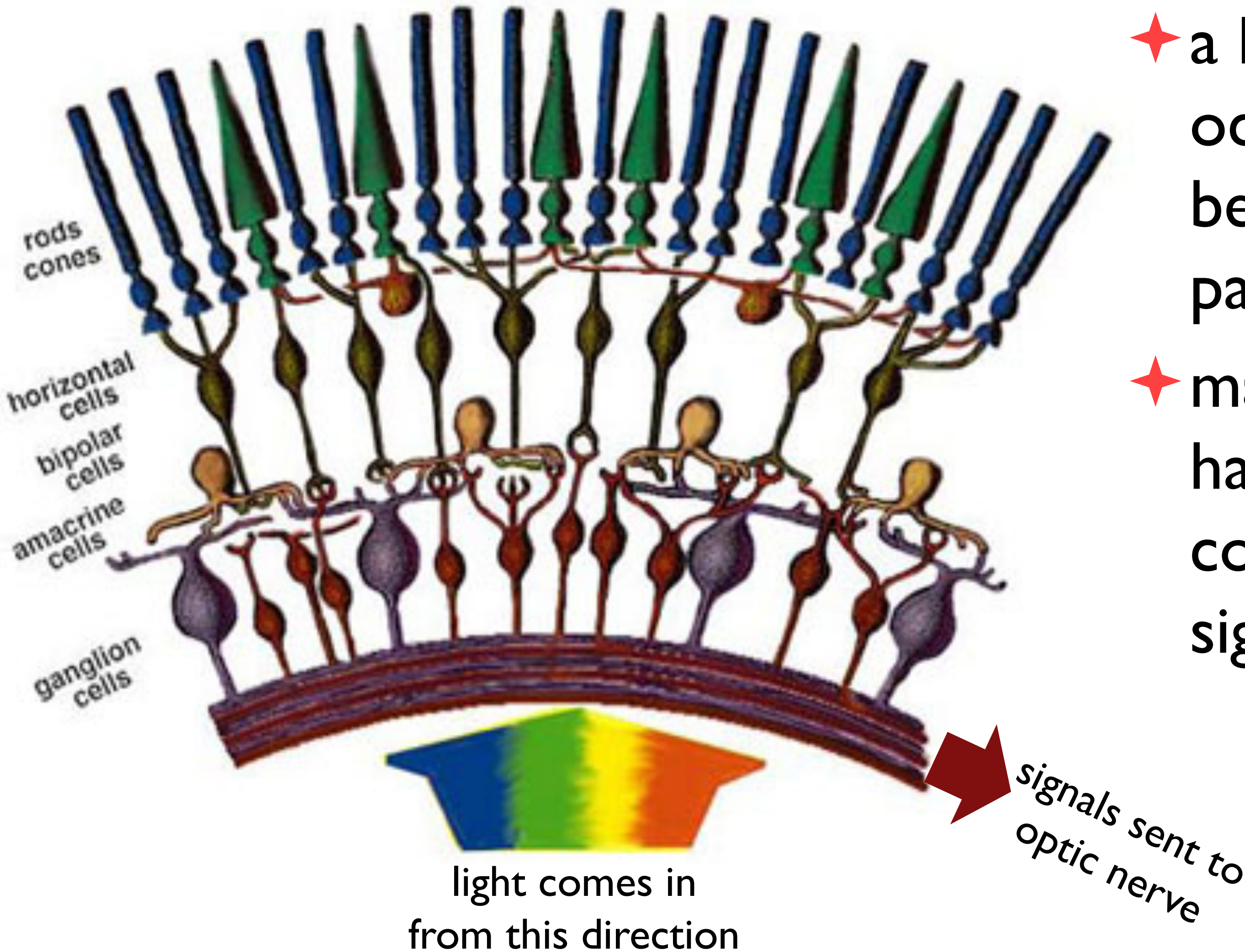
The retina

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



www.stlukeseye.com

Detailed structure of retinal processing



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

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

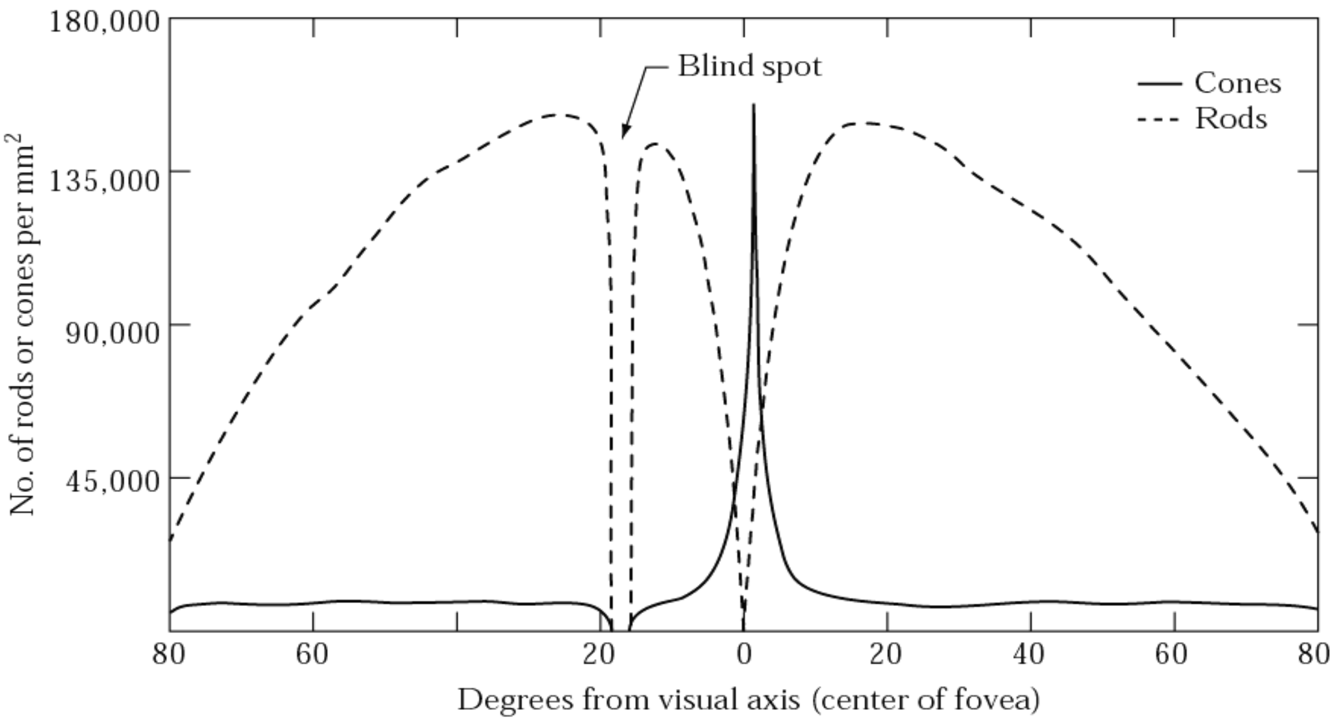
Light detectors in the retina

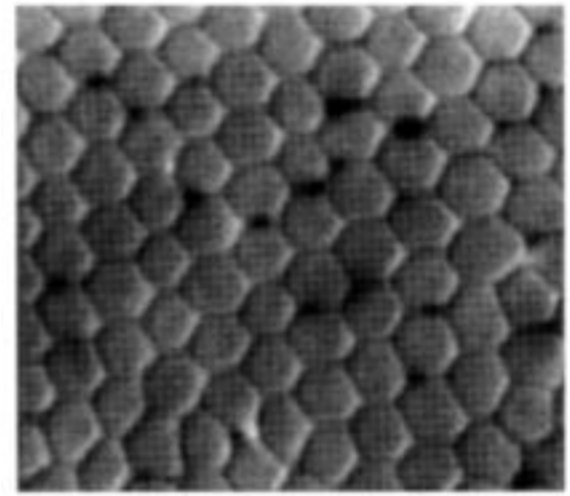
- two classes
  - rods
  - cones
- cones come in three types
  - sensitive to short, medium and long wavelengths
  - allow you to see in colour
- the cones are concentrated in the macula, at the centre of the retina
- the fovea is a densely packed region in the centre of the macula
  - contains the highest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

Foveal vision

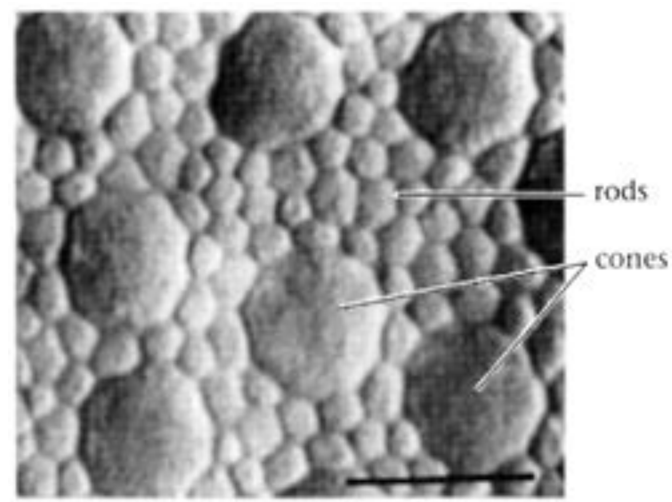
- 150,000 cones per square millimetre in the fovea
  - high resolution
  - colour
- outside fovea: mostly rods
  - lower resolution
    - many rods' inputs are combined to produce one signal to the visual cortex in the brain
  - principally monochromatic
    - there are very few cones, so little input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

Distribution of rods & cones





cones in the fovea



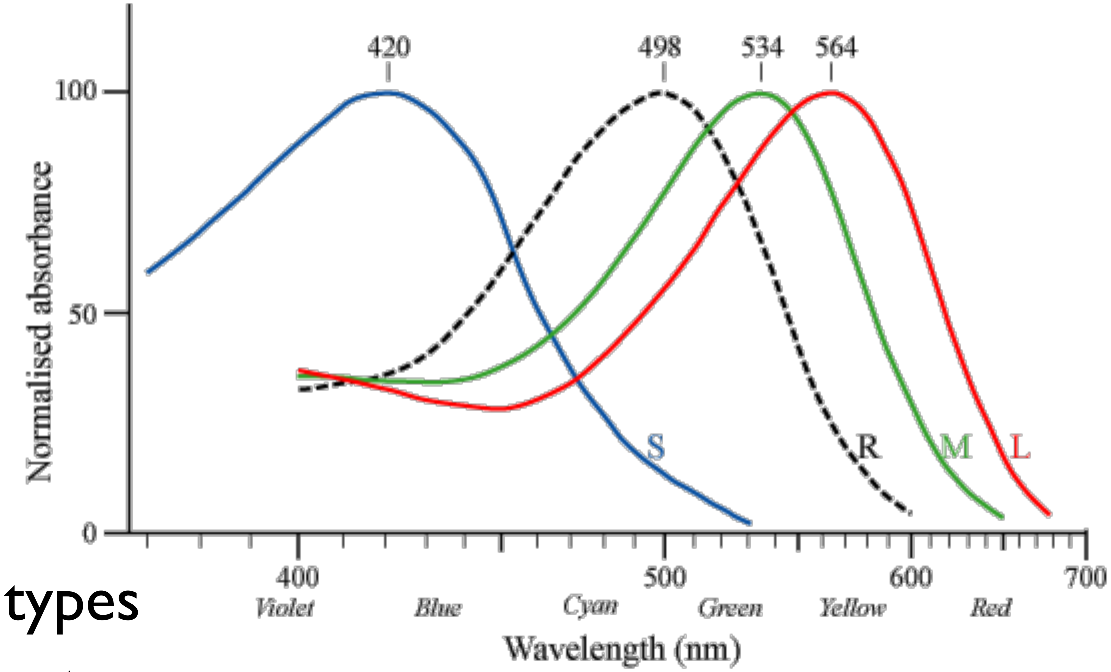
rods & cones outside the fovea

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

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

Colour vision

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

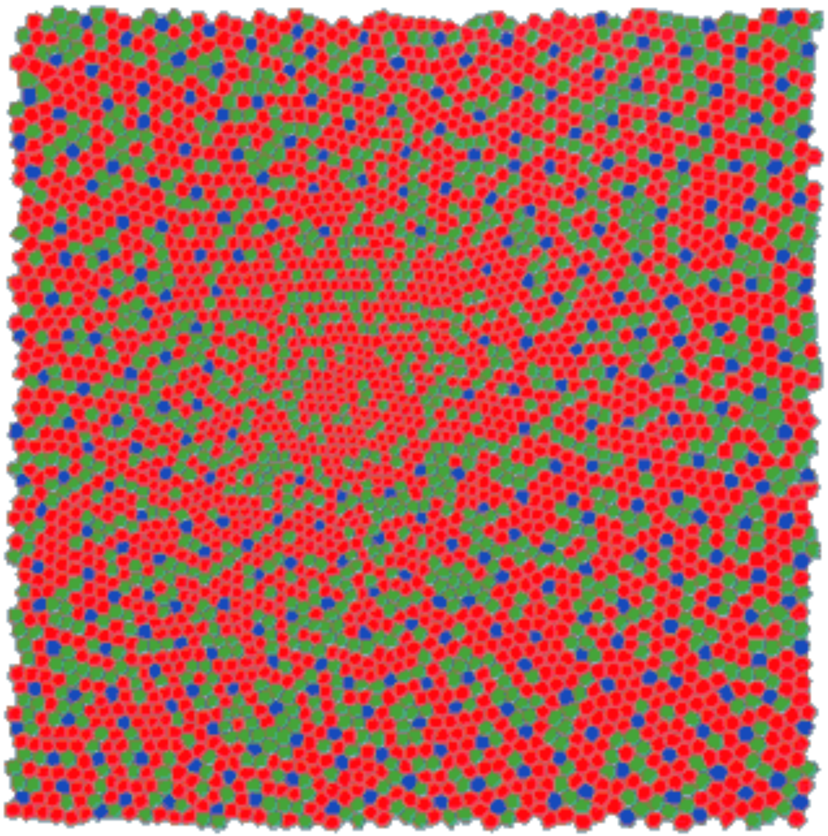


- different numbers of the different types
  - far fewer of the short wavelength receptors
  - so cannot see fine detail in blue
- overall intensity response of the cones can be calculated
  - $y(\lambda) = l(\lambda) + m(\lambda) + s(\lambda)$
  - $y = k \int P(\lambda)\, y(\lambda)\, d\lambda$ is the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

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

Distribution of different cone types

simulated cone distribution at the centre of the fovea



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

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

Colour signals sent to the brain

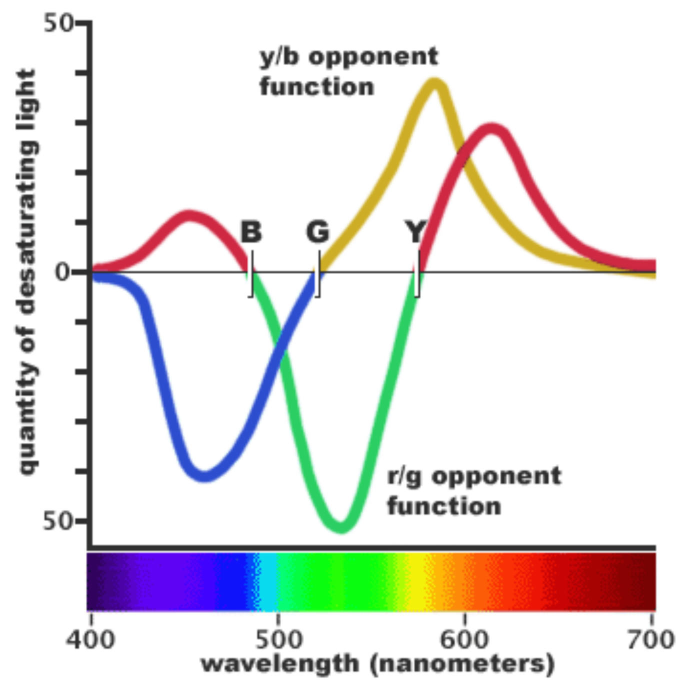
- the signal that is sent to the brain is pre-processed by the retina

long + medium + short = luminance

long − medium = red-green

long + medium − short = yellow-blue

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

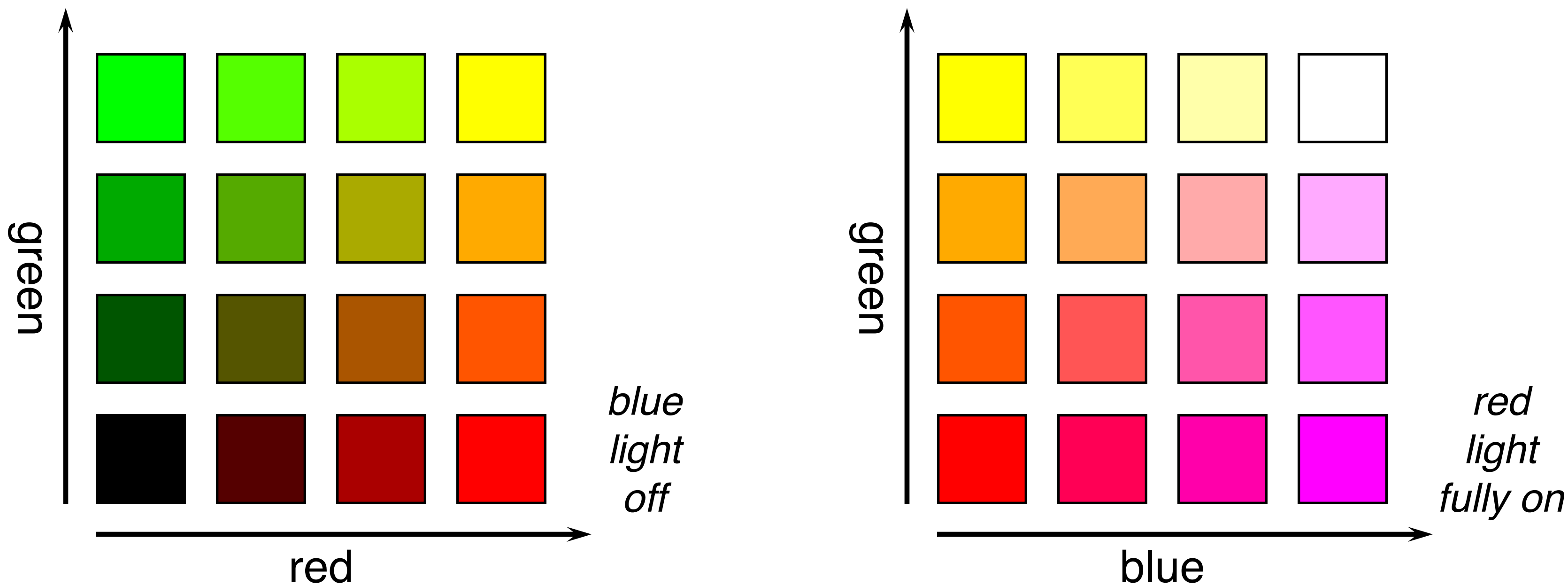


Chromatic metamerism

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

Mixing coloured lights

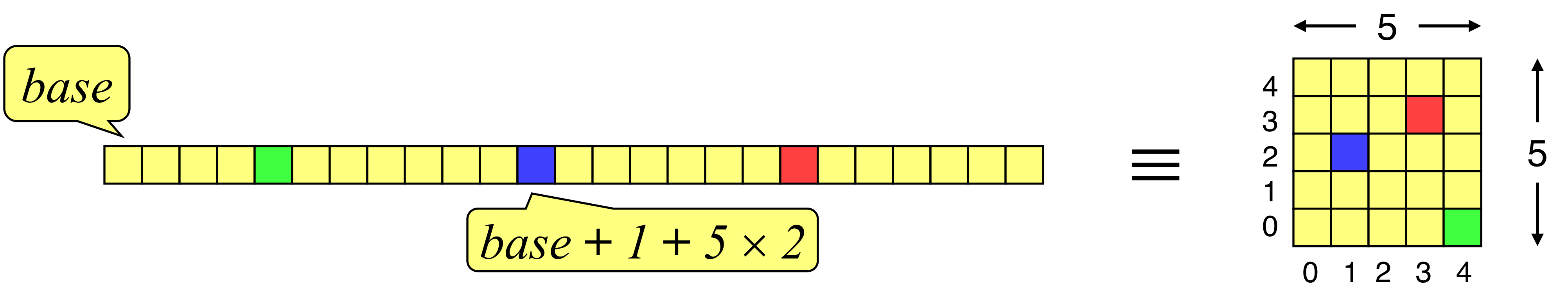
- by mixing different amounts of red, green, and blue lights we can generate a wide range of responses in the human eye



- not all colours can be created in this way

Storing images in memory

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

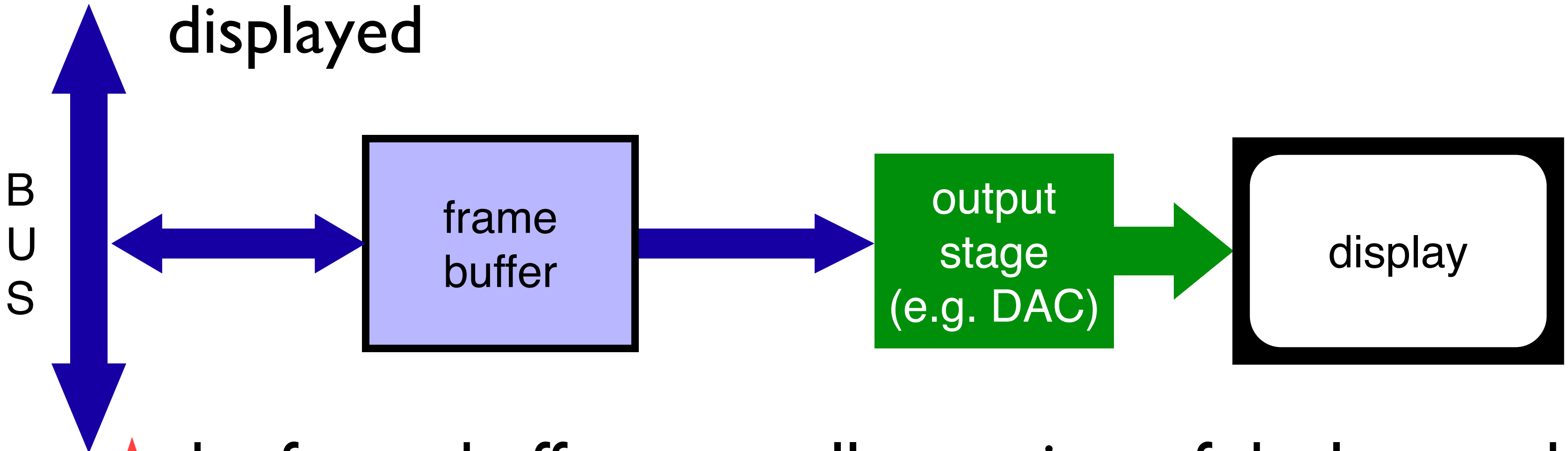


Colour images

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

The frame buffer

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

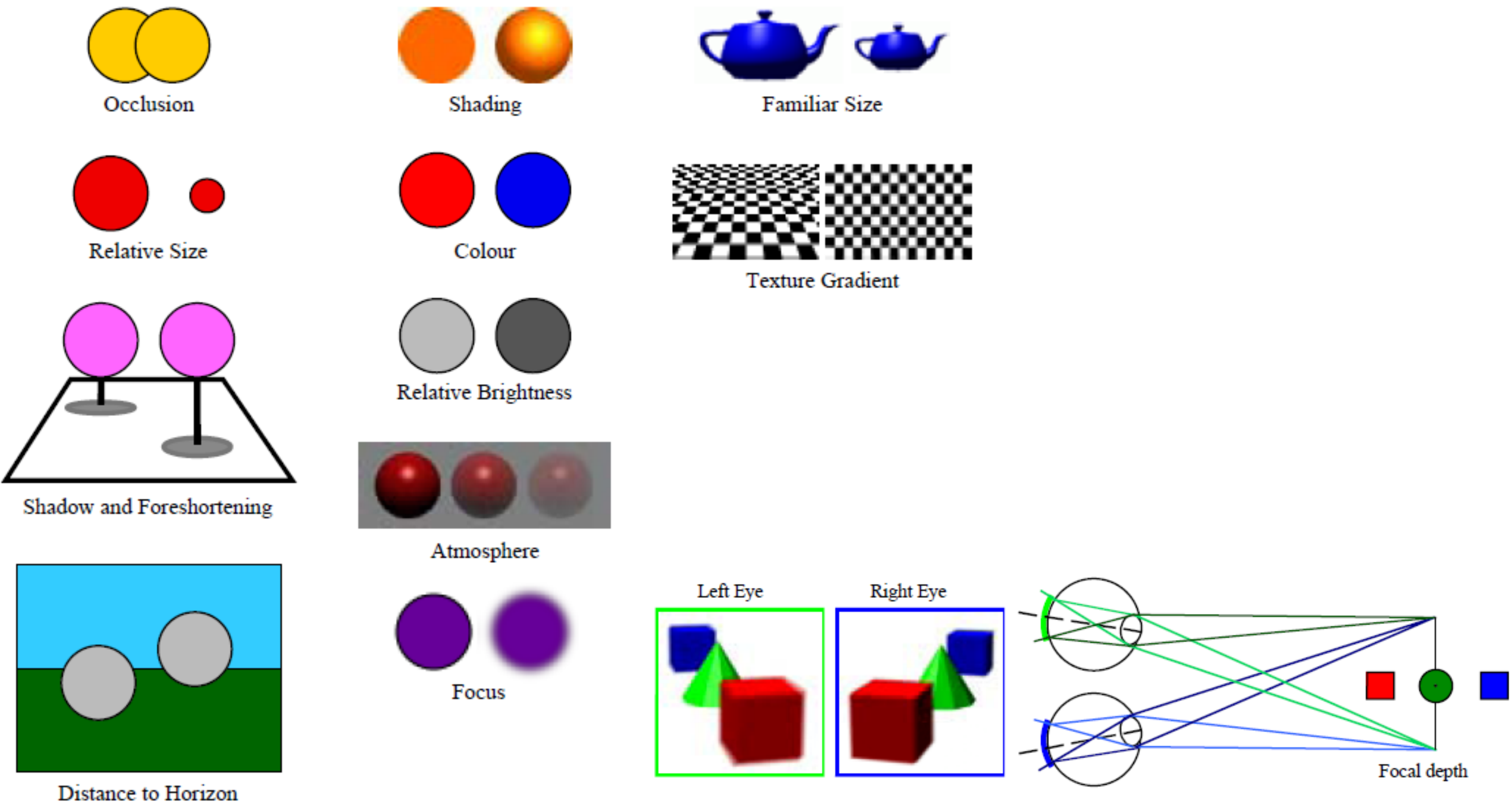


- the frame buffer normally consists of dual-ported Dynamic RAM (DRAM)
  - sometimes referred to as Video RAM (VRAM)

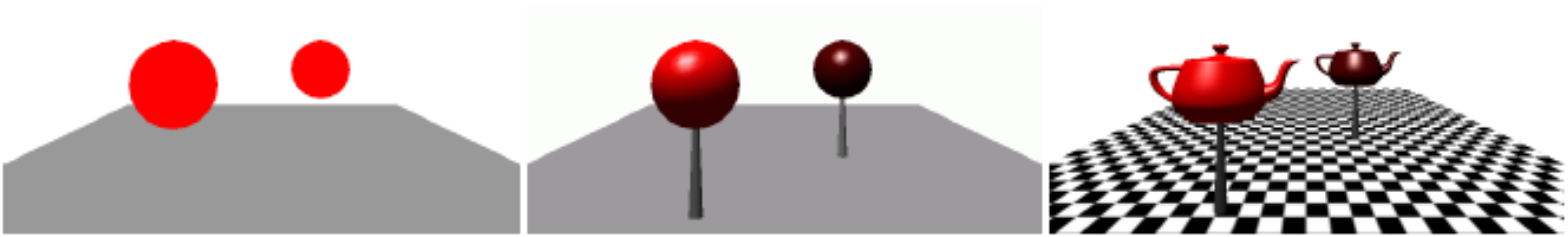
Introduction to Computer Graphics

- Background
- **Rendering**
  - Perspective
  - Reflection of light from surfaces and shading
  - Geometric models
  - Ray tracing
- Graphics pipeline
- Graphics hardware and modern OpenGL
- Technology

Depth cues


Occlusion
Shading
Familiar Size
Relative Size
Colour
Texture Gradient
Shadow and Foreshortening
Relative Brightness
Atmosphere
Distance to Horizon
Focus
Left Eye
Right Eye
Focal depth

Rendering depth



Perspective in photographs

Gates Building – the rounded version
(Stanford)

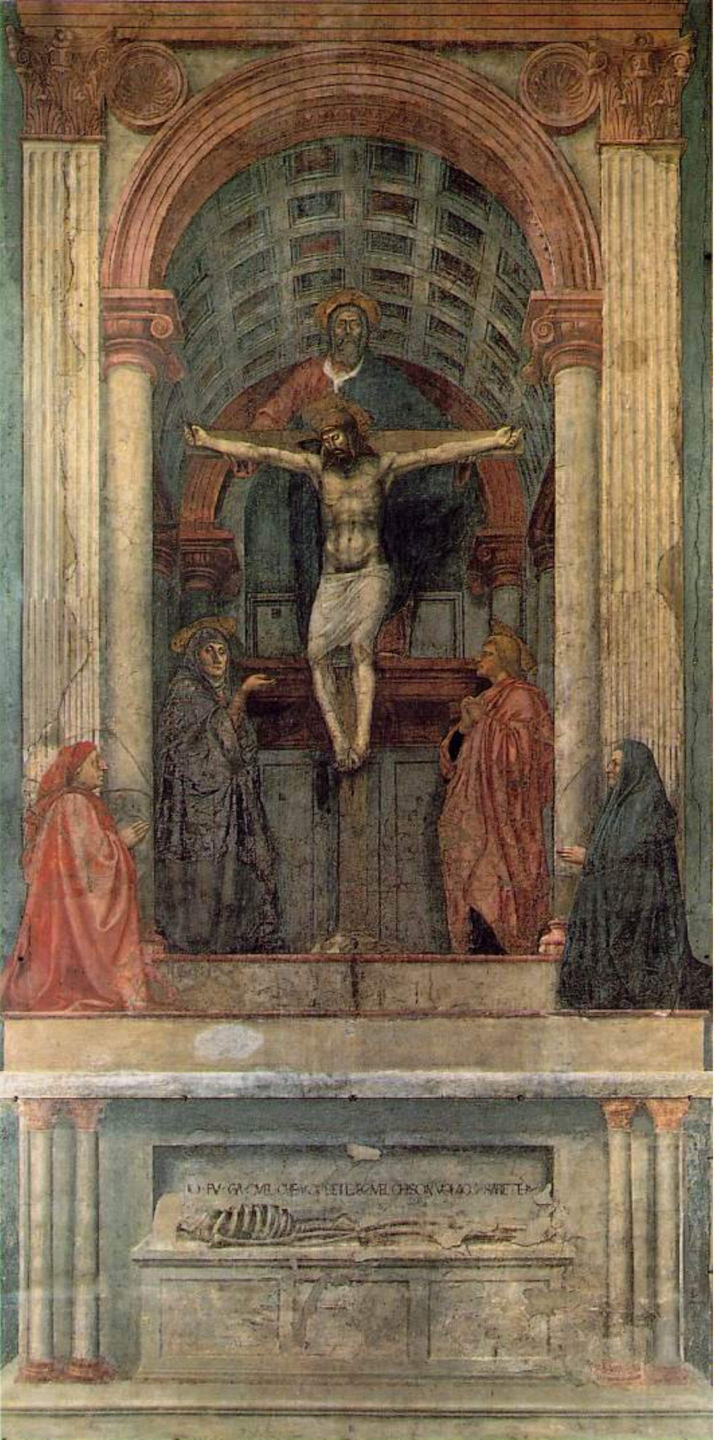
Gates Building – the rectilinear version
(Cambridge)

Early perspective

- Presentation at the Temple
- Ambrogio Lorenzetti 1342
- Uffizi Gallery Florence

Wrong perspective

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



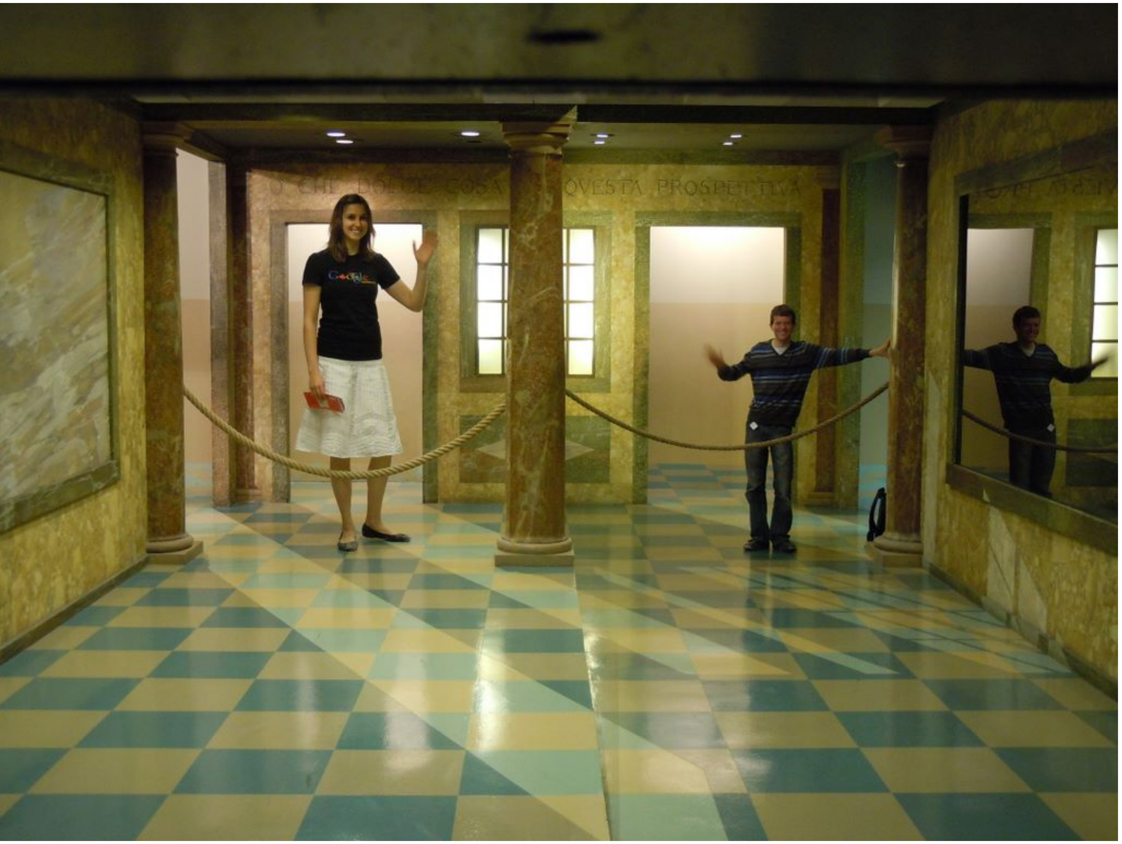
Renaissance perspective

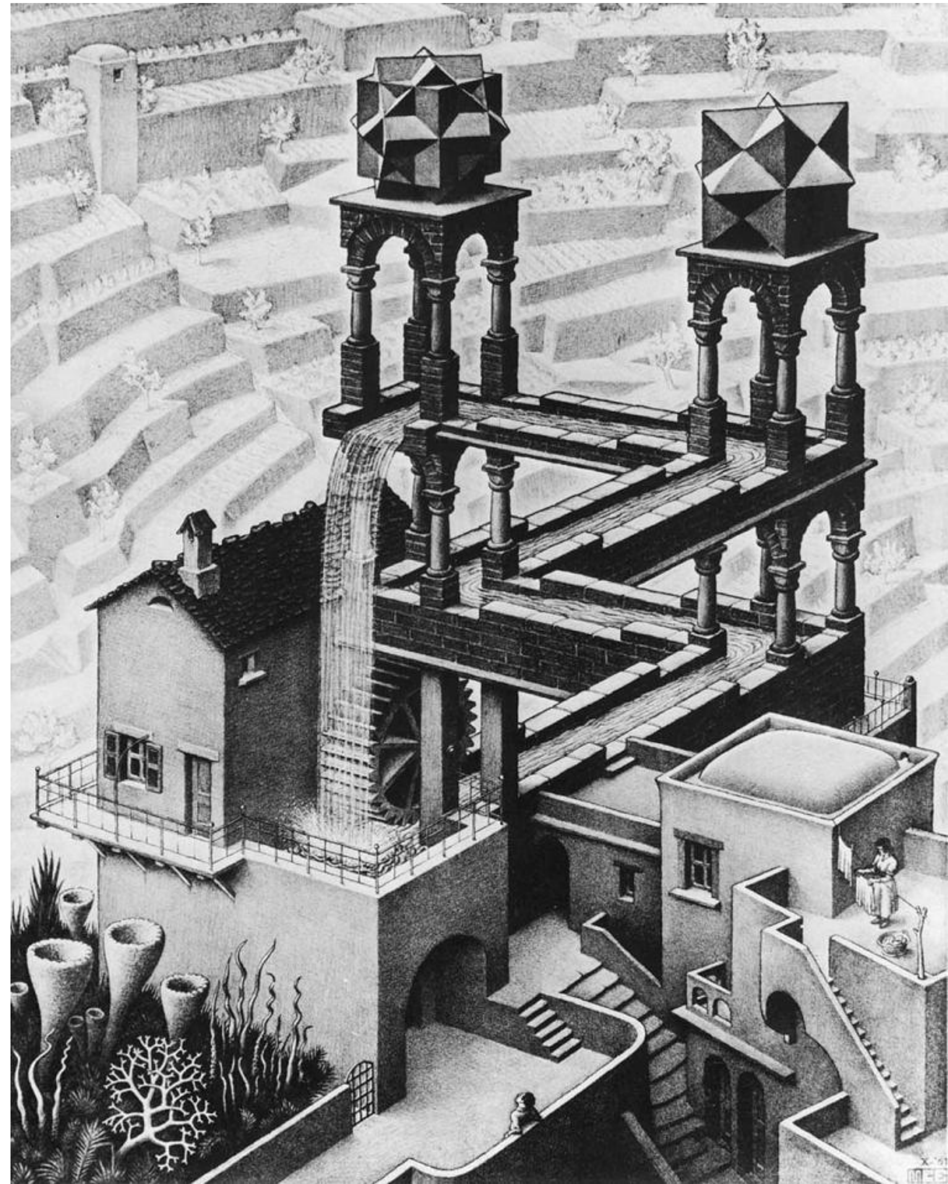
- Geometrical perspective
  Filippo Brunelleschi 1413
- Holy Trinity fresco
- Masaccio (Tommaso di Ser Giovanni di Simone) 1425
- Santa Maria Novella
  Florence
- *De pictura* (On painting)
  textbook by Leon Battista Alberti
  1435

More perspective

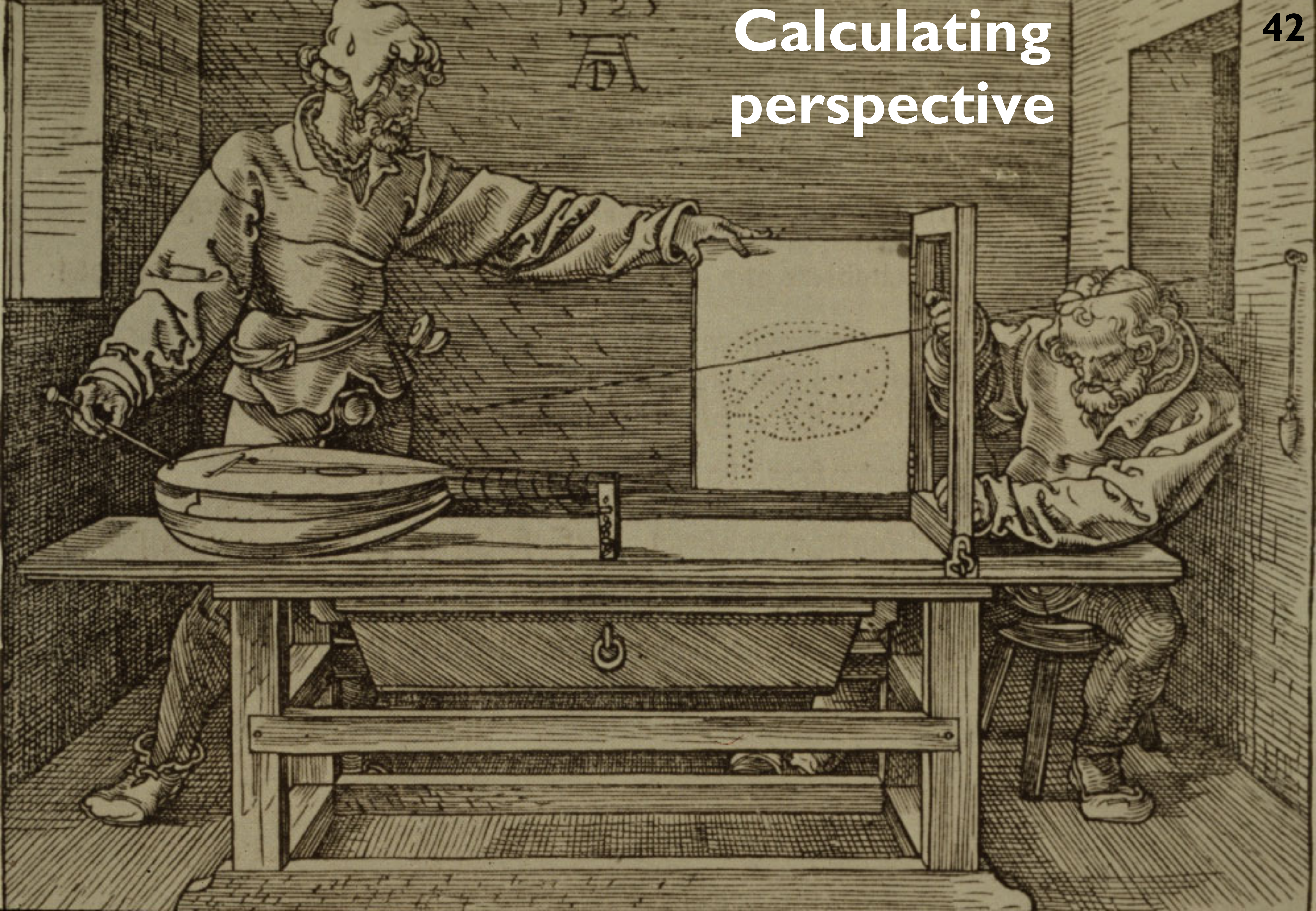
- The Annunciation with Saint Emidius
- Carlo Crivelli 1486
- National Gallery London

False perspective





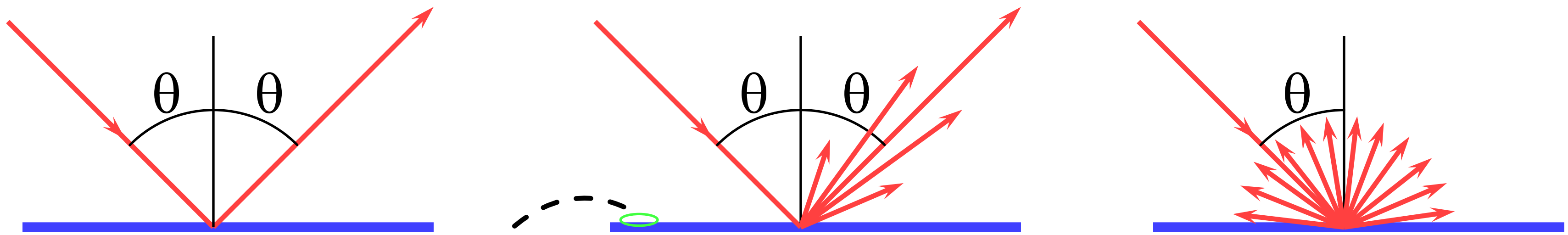
Calculating perspective



Illumination and shading

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

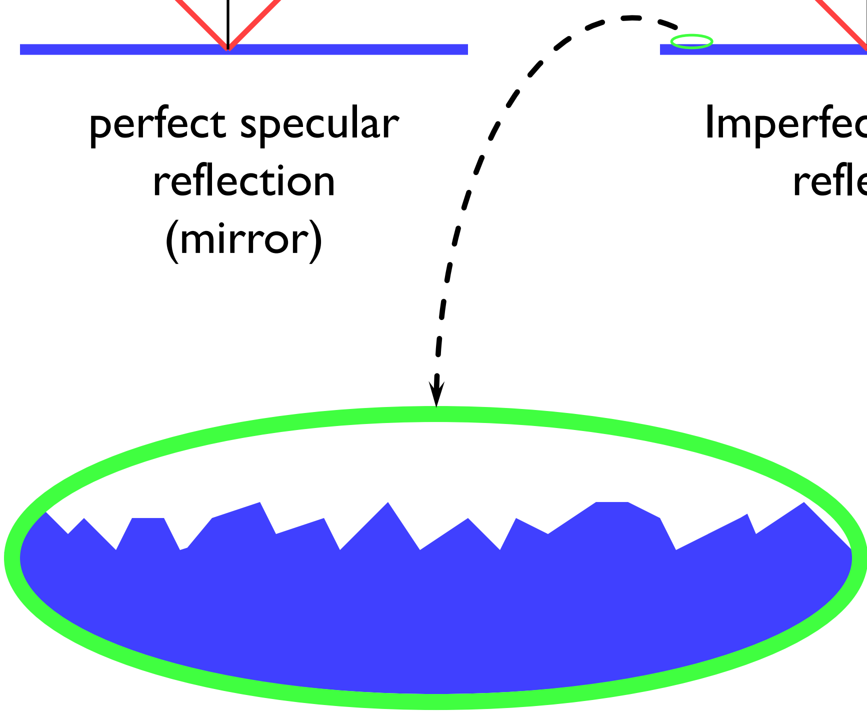
How do surfaces reflect light?



perfect specular reflection (mirror)

Imperfect specular reflection

diffuse reflection (Lambertian reflection)



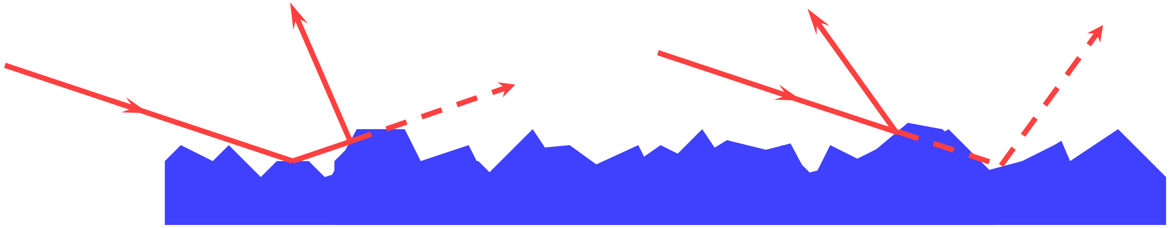
the surface of a specular reflector is facetted, each facet reflects perfectly but in a slightly different direction to the other facets

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

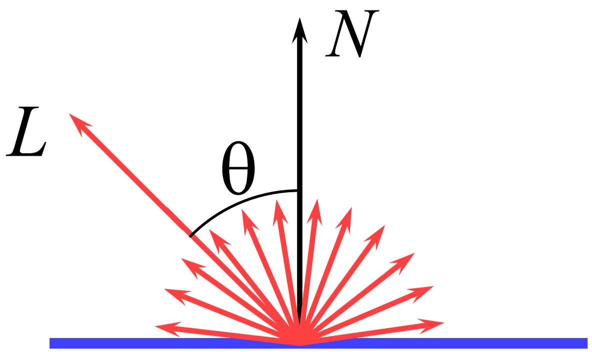


- plastics are good examples of surfaces with:
  - specular reflection in the light’s colour
  - diffuse reflection in the plastic’s colour

Calculating the shading of a surface

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

Diffuse shading calculation



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

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

N is the normal to the surface

I_l is the intensity of the light source

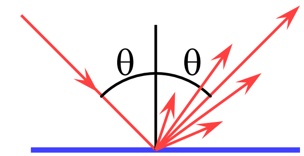
k_d is the proportion of light which is diffusely reflected by the surface

I is the intensity of the light reflected by the surface

use this equation to calculate the colour of a pixel

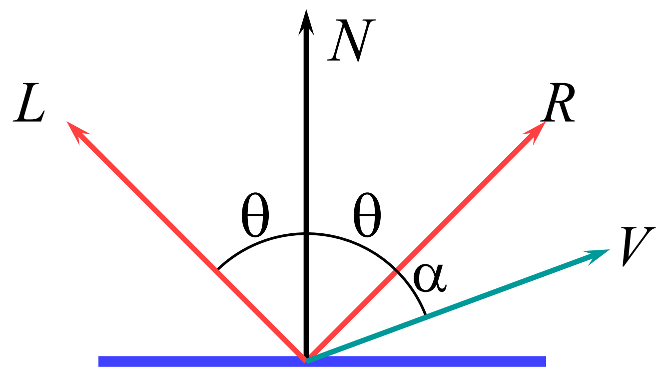
Diffuse shading: comments

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



Specular reflection

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



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

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

R is the vector of perfect reflection

N is the normal to the surface

V is a normalised vector pointing at the viewer

I_l is the intensity of the light source

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

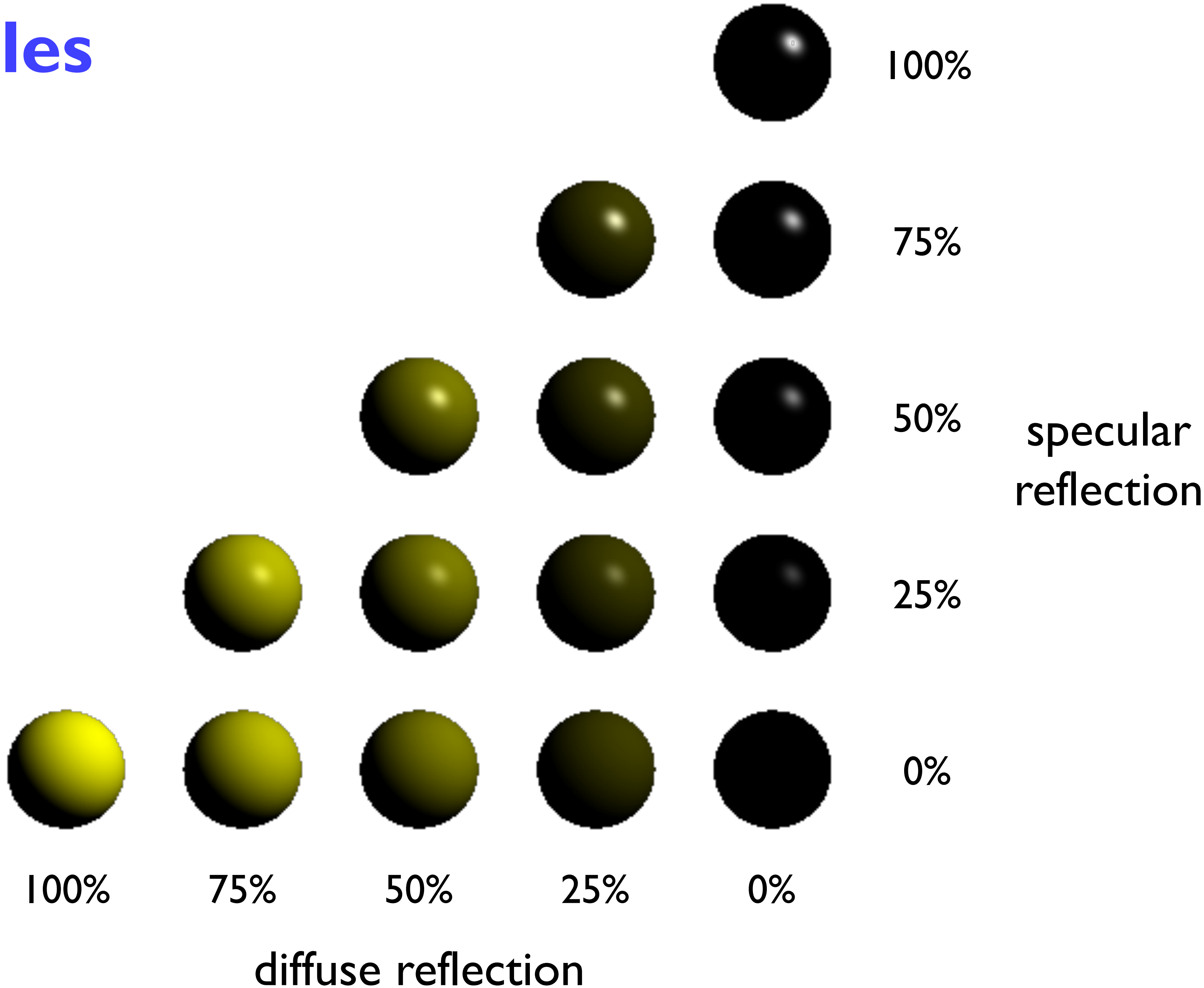
n is Phong's *ad hoc* "roughness" coefficient

I is the intensity of the specularly reflected light

n=1 n=3 n=7 n=20 n=40

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

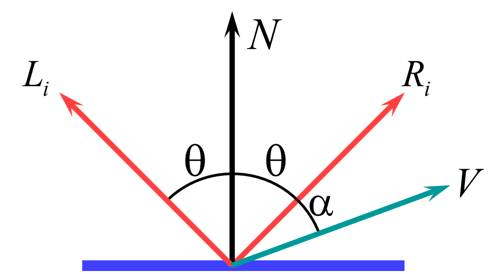
Examples


100%
75%
50%
25%
0%
specular reflection
100%
75%
50%
25%
0%
diffuse reflection

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

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



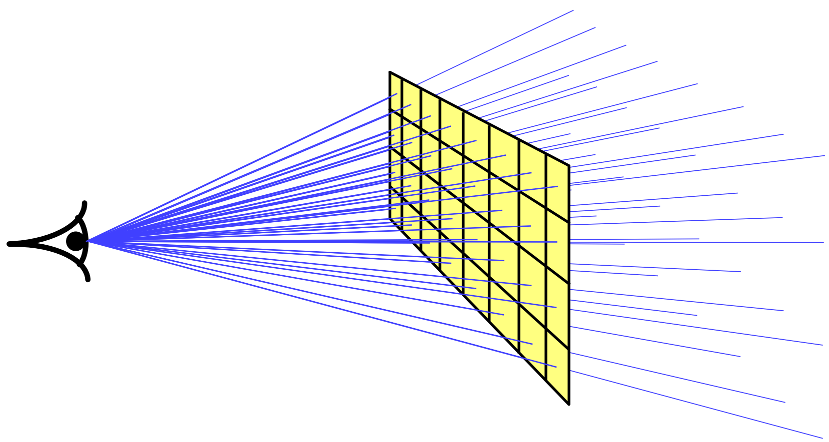
  - the more lights there are in the scene, the longer this calculation will take

The gross assumptions revisited

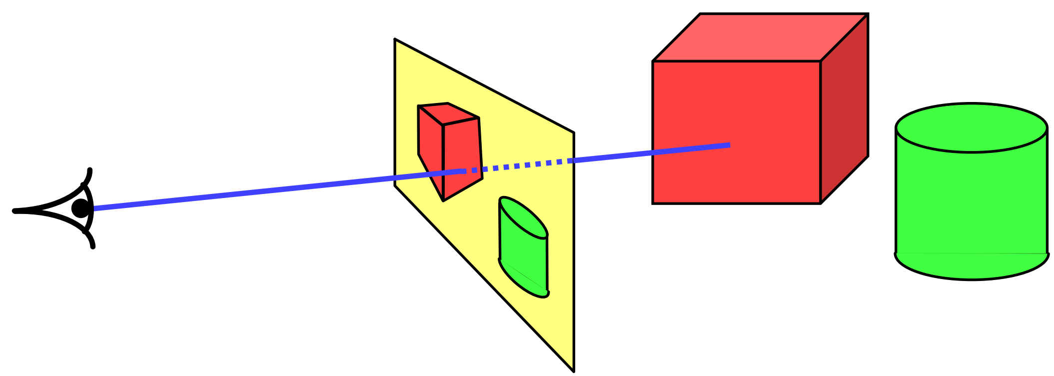
- diffuse reflection
- approximate specular reflection
- no shadows
  - need to do ray tracing or shadow mapping to get shadows
- lights at infinity
  - can add local lights at the expense of more calculation
    - need to interpolate the L vector
- no interaction between surfaces
  - cheat!
    - assume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

Ray tracing

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

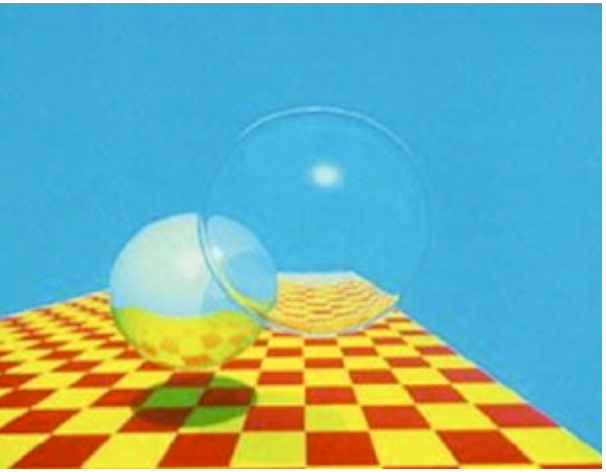


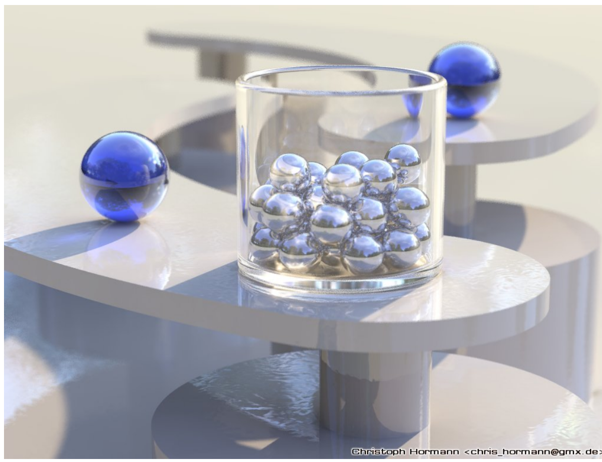
shoot a ray through each pixel



whatever the ray hits determines the colour of that pixel

Ray tracing: examples





ray tracing easily handles reflection, refraction, shadows and blur

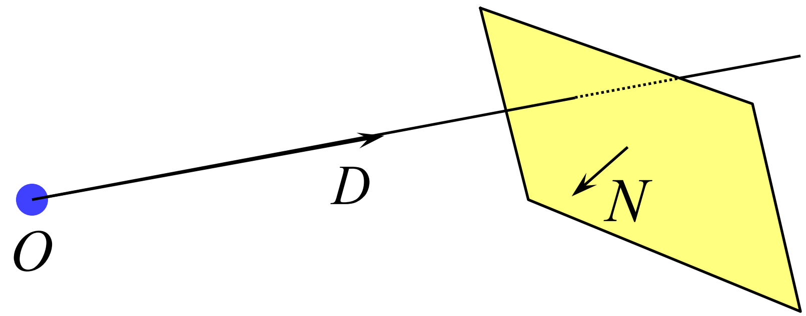
ray tracing is computationally expensive

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 I

- plane



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

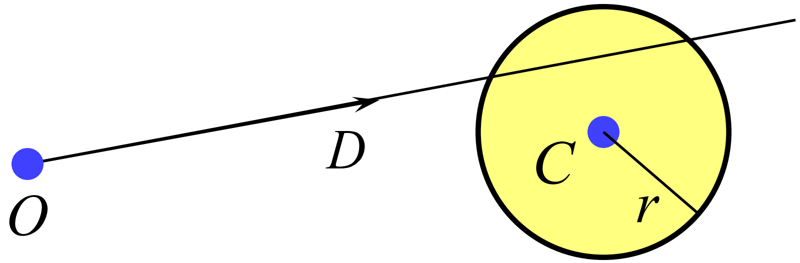
$$\text{plane: } P \cdot N + d = 0$$

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

- polygon or disc
  - intersection the ray with the plane of the polygon
    - as above
  - then check to see whether the intersection point lies inside the polygon
    - a 2D geometry problem (which is simple for a disc)

Intersection of a ray with an object 2

- sphere



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

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

$$a = D \cdot D$$

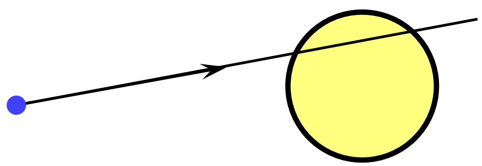
$$b = 2D \cdot (O - C)$$

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

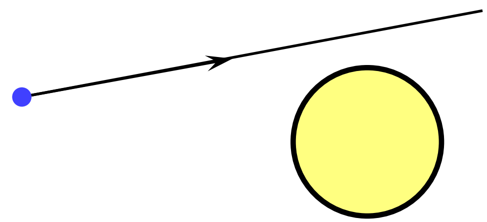
$$d = \sqrt{b^2 - 4ac}$$

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

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



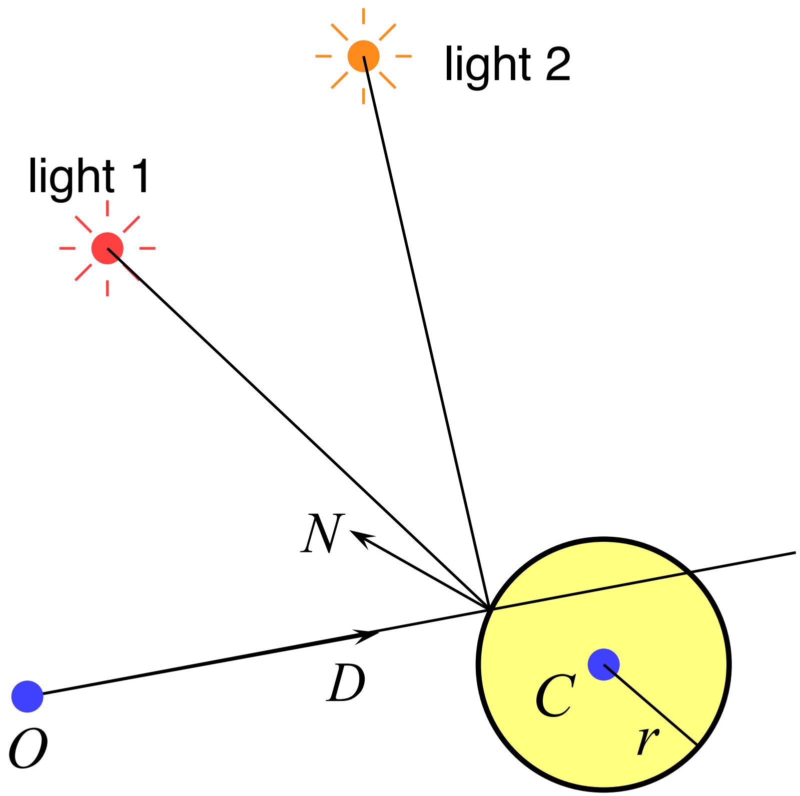
d real



d imaginary

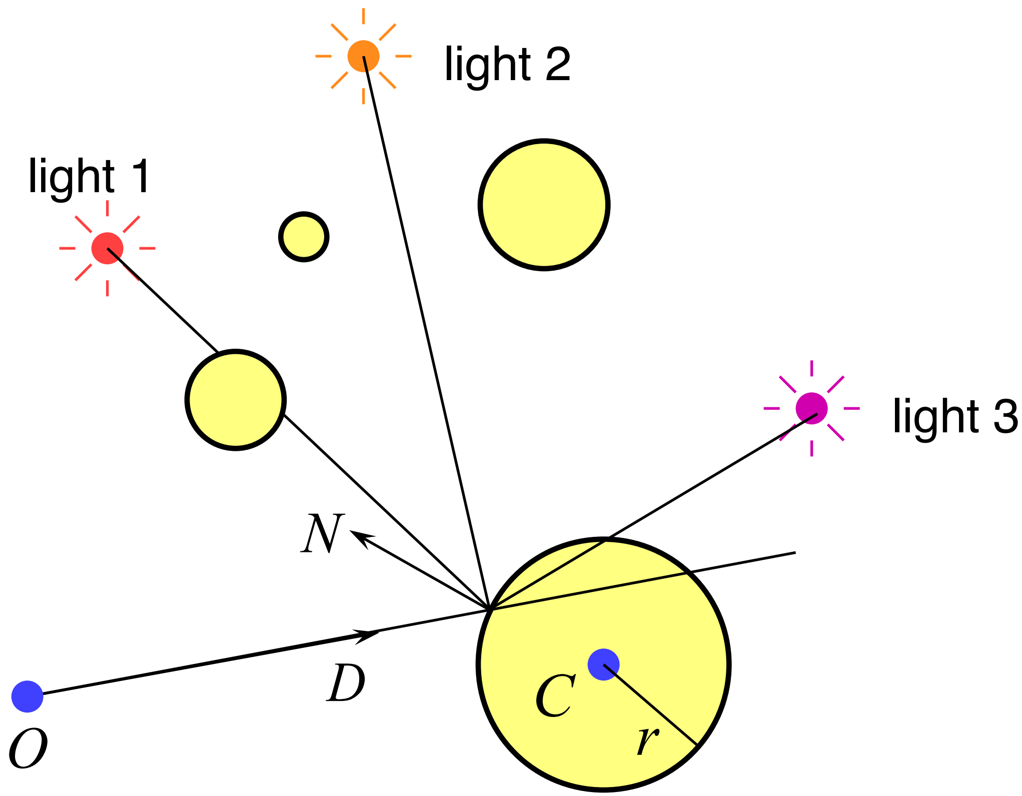
- cylinder, cone, torus
  - all similar to sphere
  - try them as an exercise

Ray tracing: shading



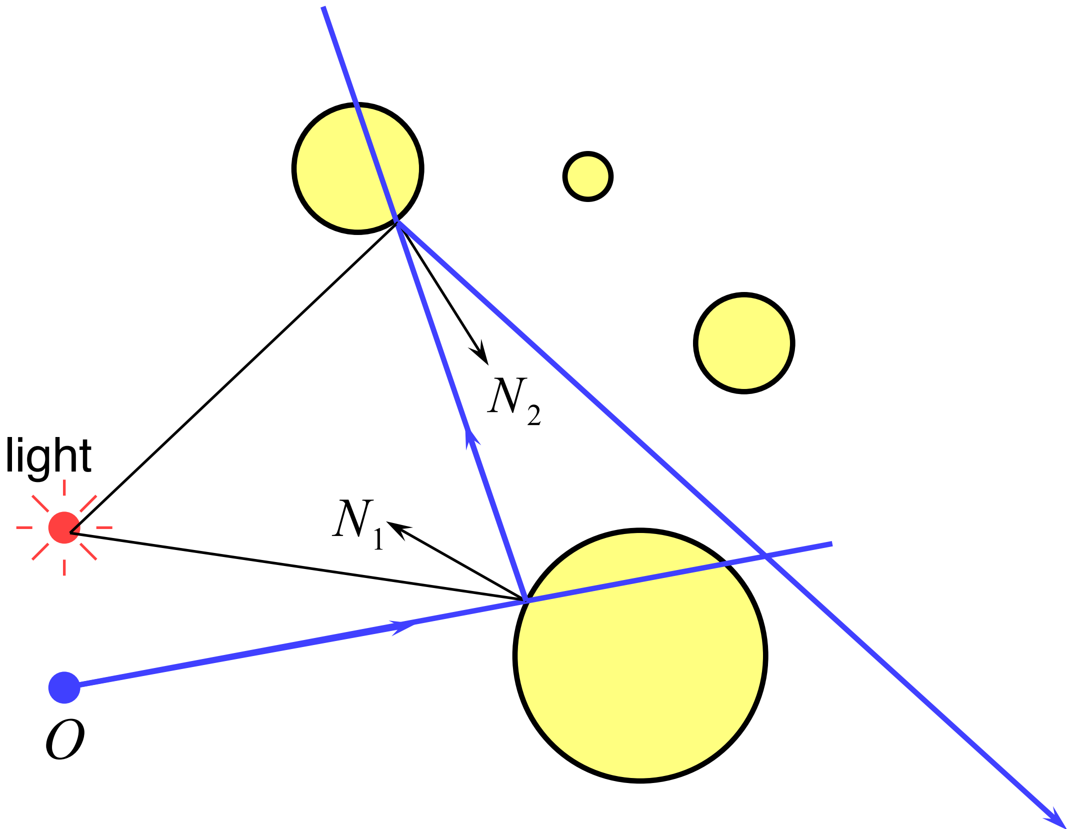
- once you have the intersection of a ray with the nearest object you can also:
  - calculate the normal to the object at that intersection point
  - shoot rays from that point to all of the light sources, and calculate the diffuse and specular reflections off the object at that point
    - this (plus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

Ray tracing: shadows



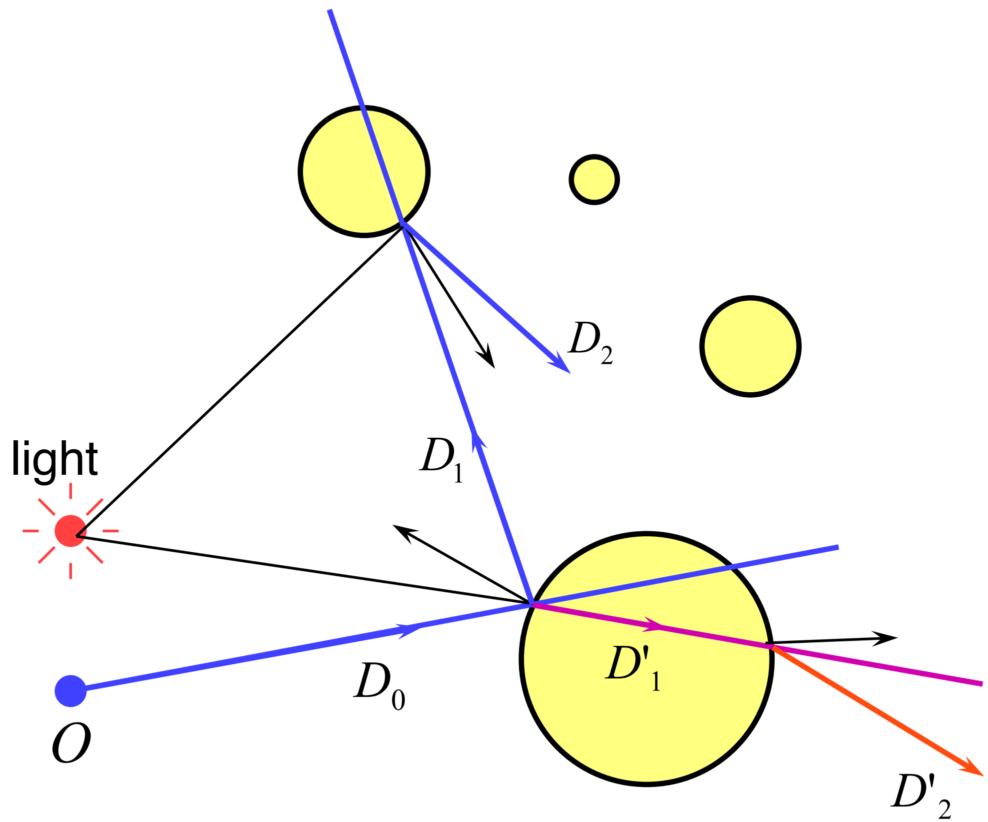
- because you are tracing rays from the intersection point to the light, you can check whether another object is between the intersection and the light and is hence casting a shadow
  - also need to watch for self-shadowing

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

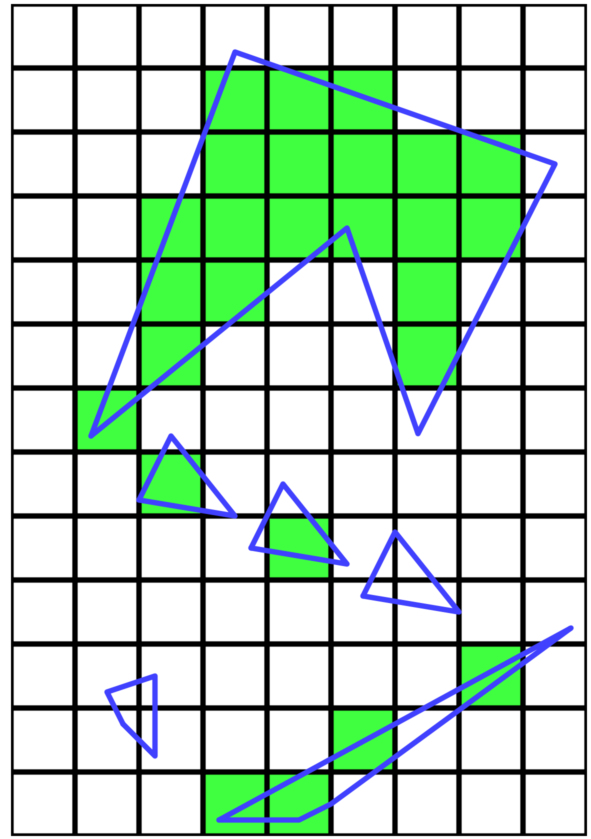
Ray tracing: transparency & refraction



- 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

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

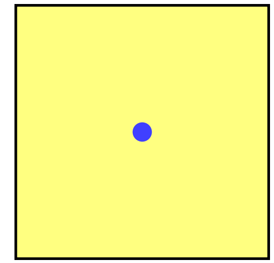


Anti-aliasing

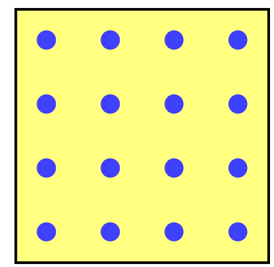
- these artefacts (and others) are jointly known as aliasing
- methods of ameliorating the effects of aliasing are known as *anti-aliasing*
    - in signal processing *aliasing* is a precisely defined technical term for a particular kind of artefact
    - in computer graphics its meaning has expanded 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

Sampling in ray tracing

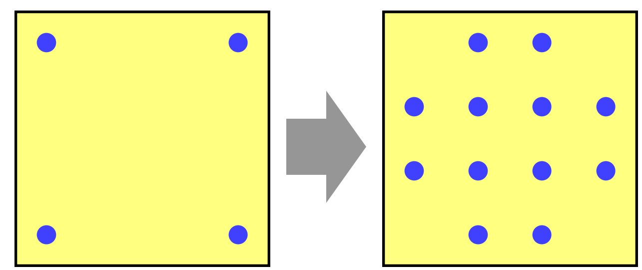
- single point
  - shoot a single ray through the pixel's centre



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

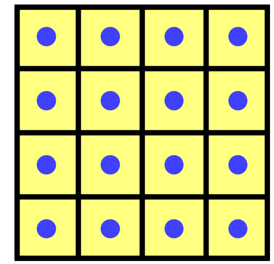


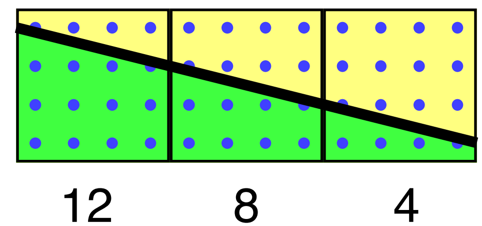
- adaptive super-sampling
  - shoot a few rays through the pixel, check the variance of the resulting values, if similar enough stop, otherwise shoot some more rays

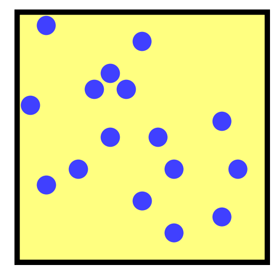


Types of super-sampling I

- 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

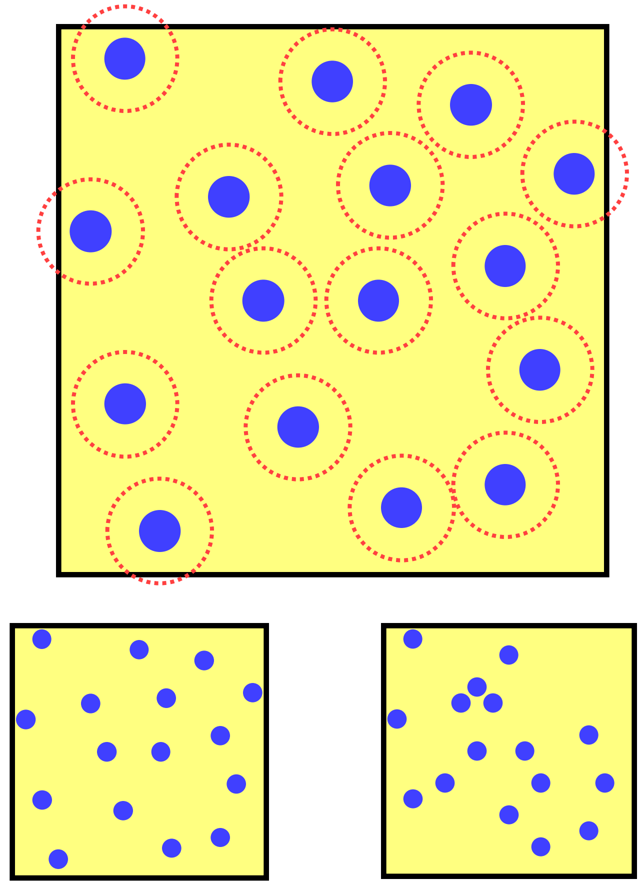






Types of super-sampling 2

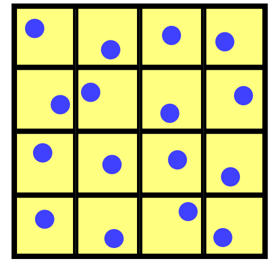
- Poisson disc
  - shoot N rays at random points in the pixel with the proviso that no two rays shall pass through the pixel closer than ε to one another
  - for N rays this produces a better looking image than pure random sampling
  - very hard to implement properly

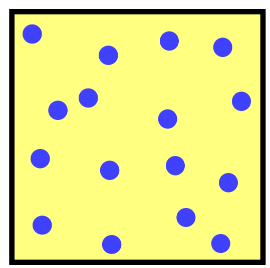


Poisson disc   pure random

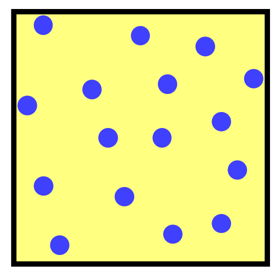
Types of super-sampling 3

- 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

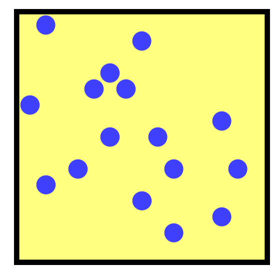




jittered



Poisson disc



pure random

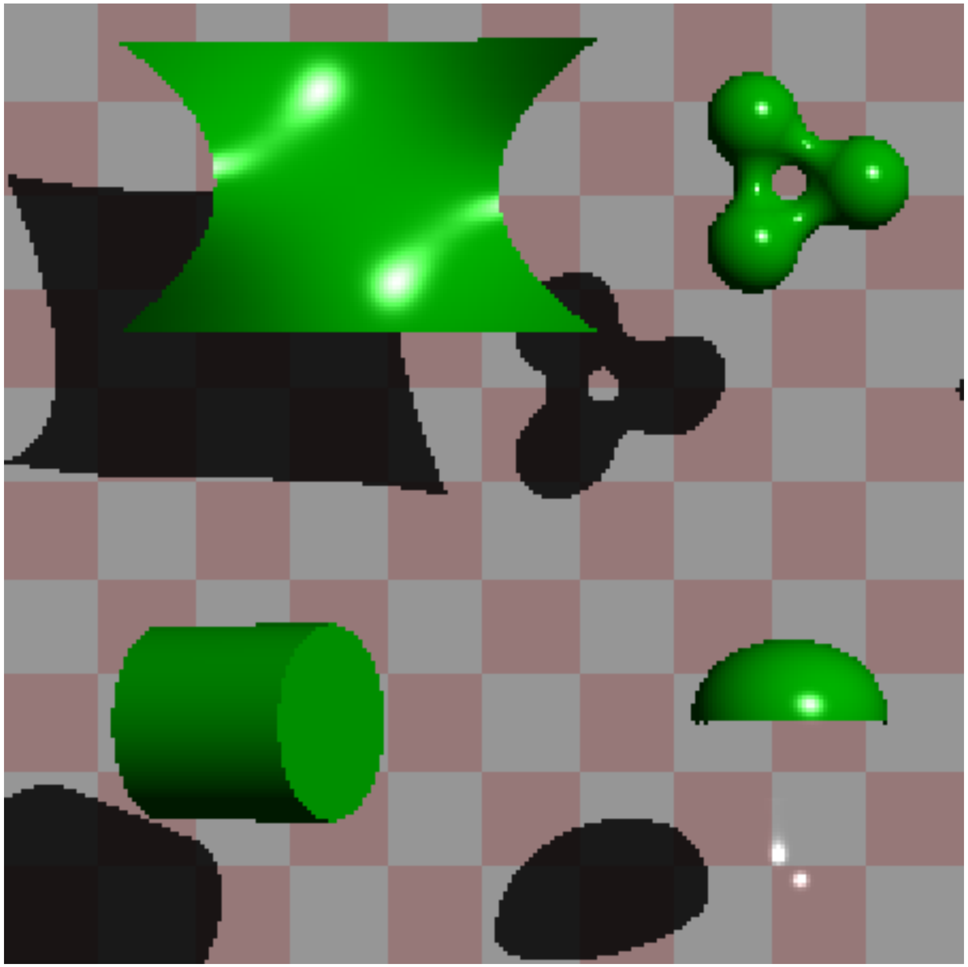
More reasons for wanting to take multiple samples per pixel

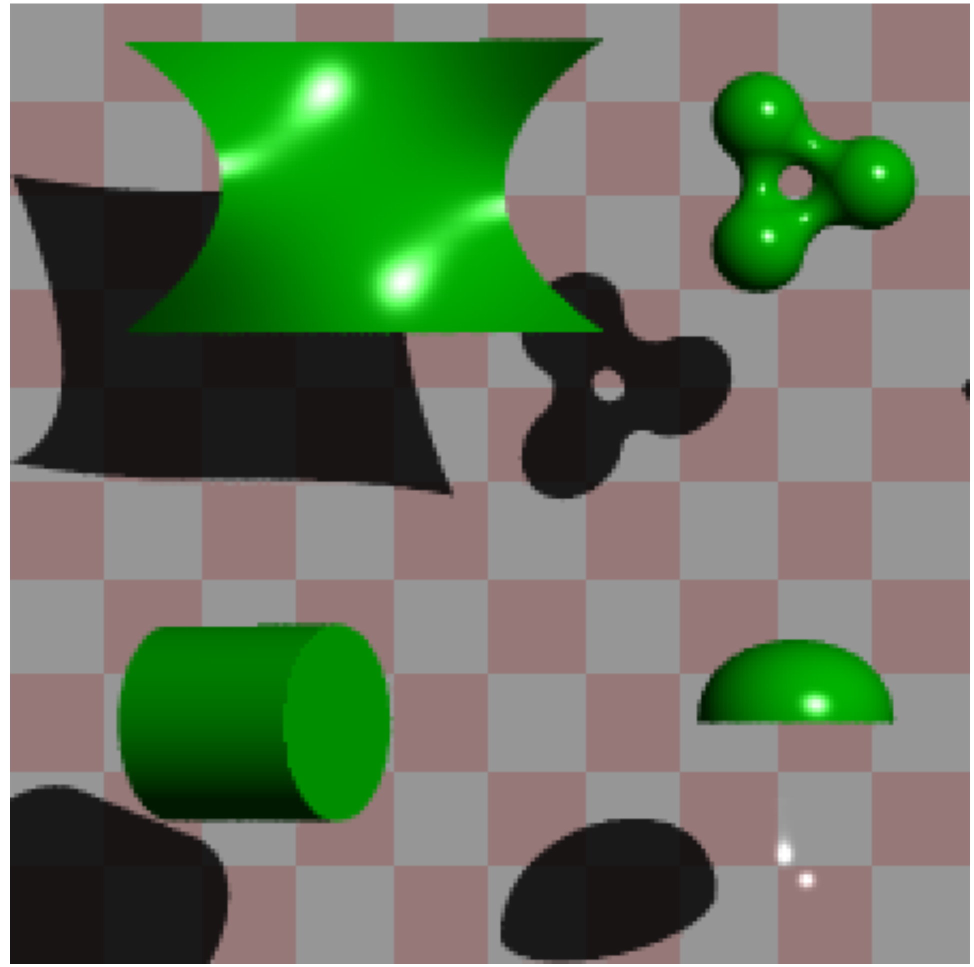
- super-sampling is only one reason why we might want to take multiple samples per pixel
- many effects can be achieved by distributing the multiple samples over some range
  - called *distributed* ray tracing
    - N.B. *distributed* means distributed over a range of values
- can work in two ways
  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

Anti-aliasing

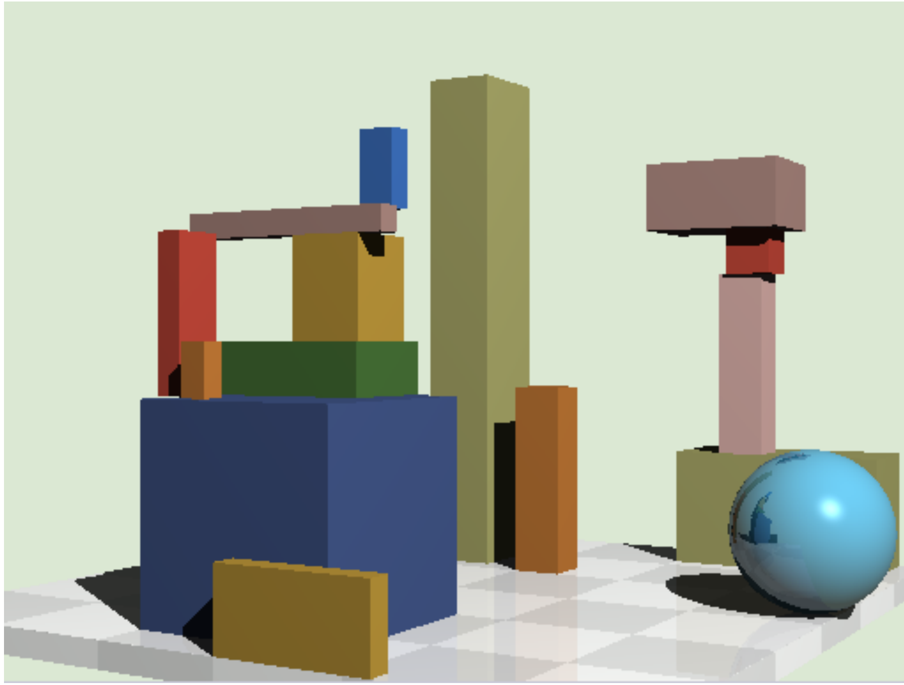

one sample per pixel


multiple samples per pixel

Area *vs* point light source

an area light source produces soft shadows

a point light source produces hard shadows



Finite aperture

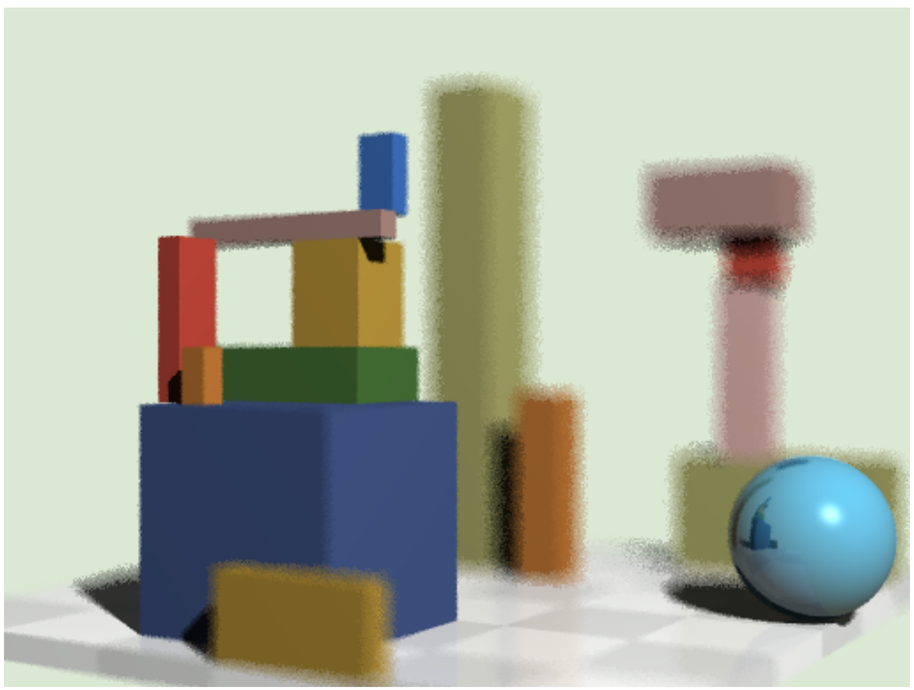
left, a pinhole camera

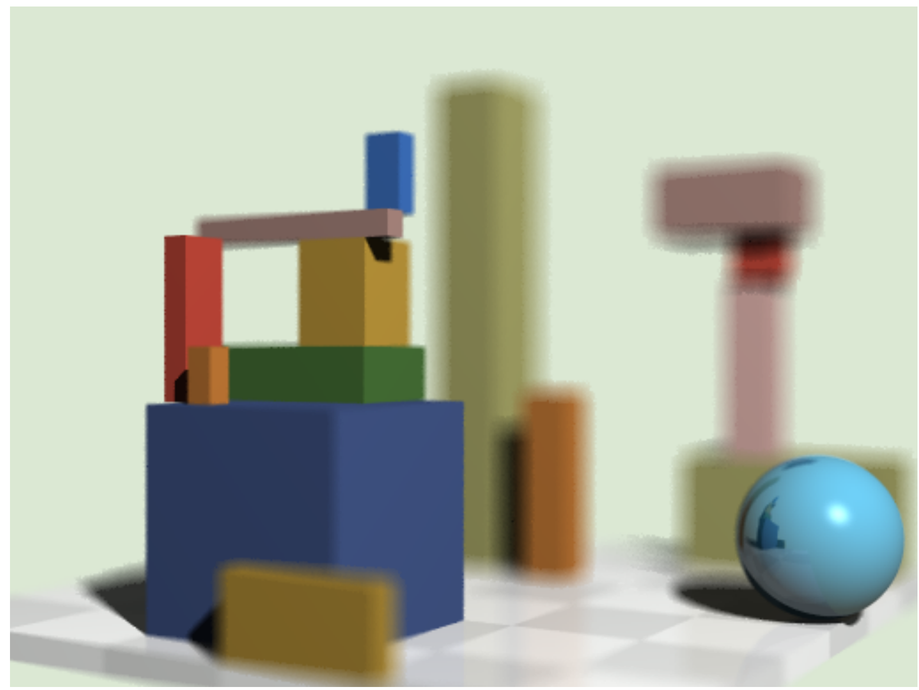
below, a finite aperture camera

below left, 12 samples per pixel

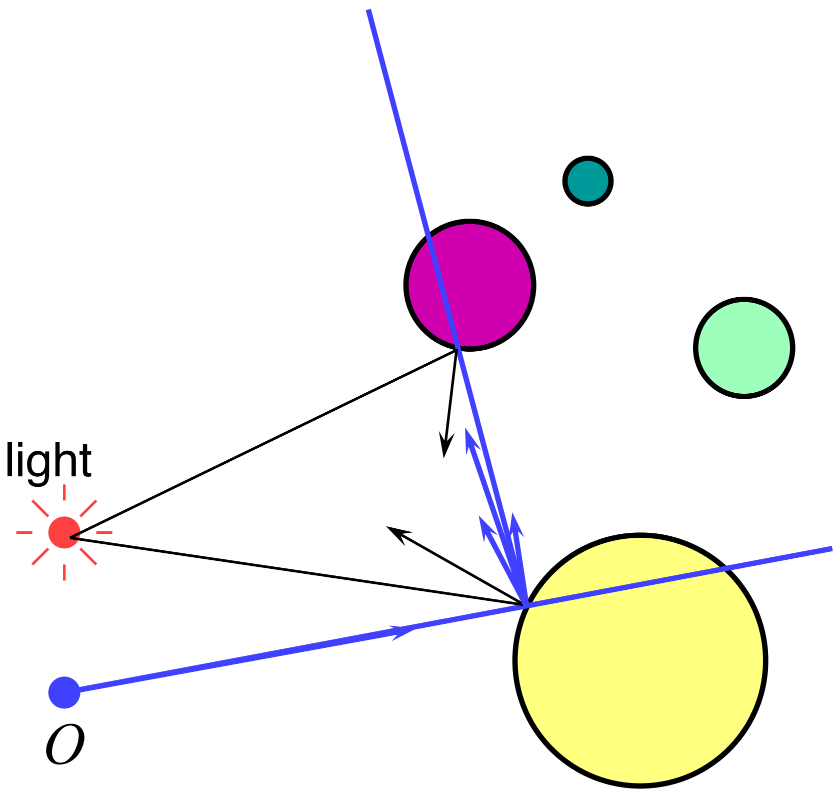
below right, 120 samples per pixel

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



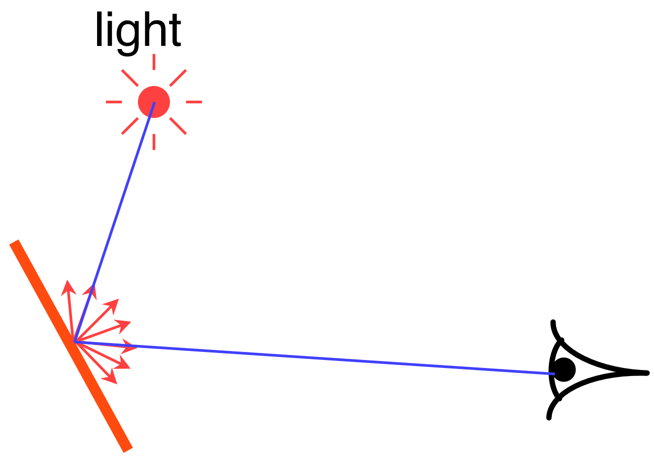


Distributed ray tracing for specular reflection

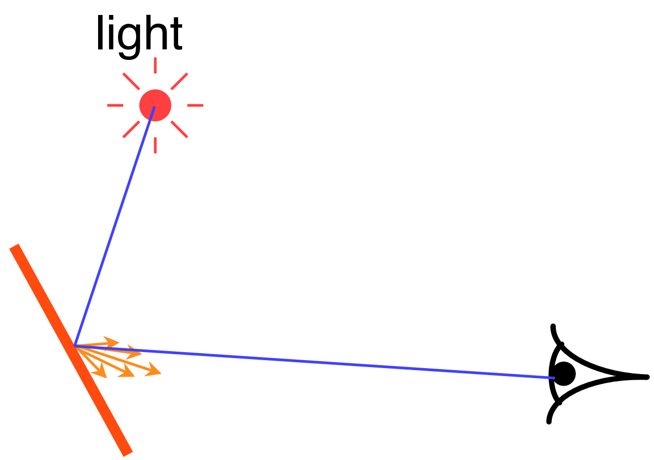


- previously we could only calculate the effect of perfect reflection
- we can now distribute the reflected rays over the range of directions from which specularly reflected light could come
- provides a method of handling some of the inter-reflections between objects in the scene
- requires a very large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

Handling direct illumination

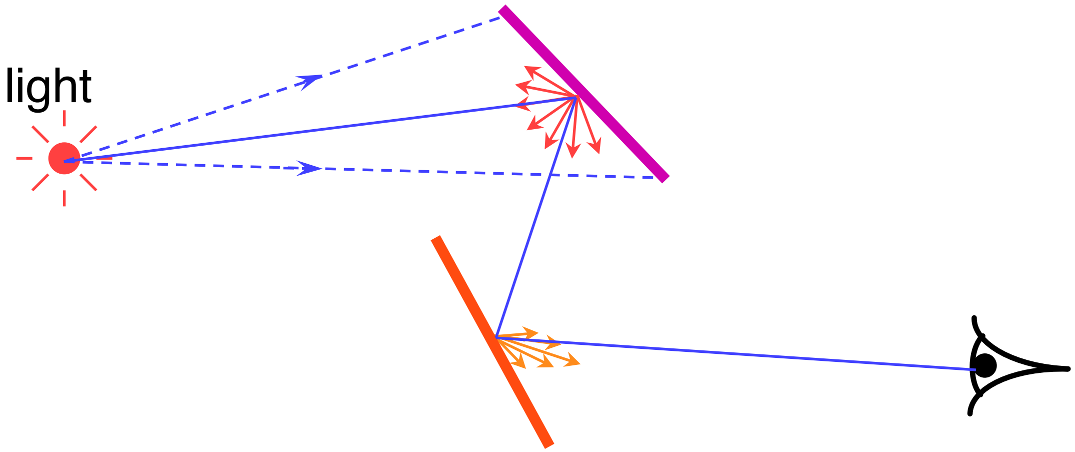


- diffuse reflection
  - handled by ray tracing and polygon scan conversion
  - assumes that the object is a perfect Lambertian reflector

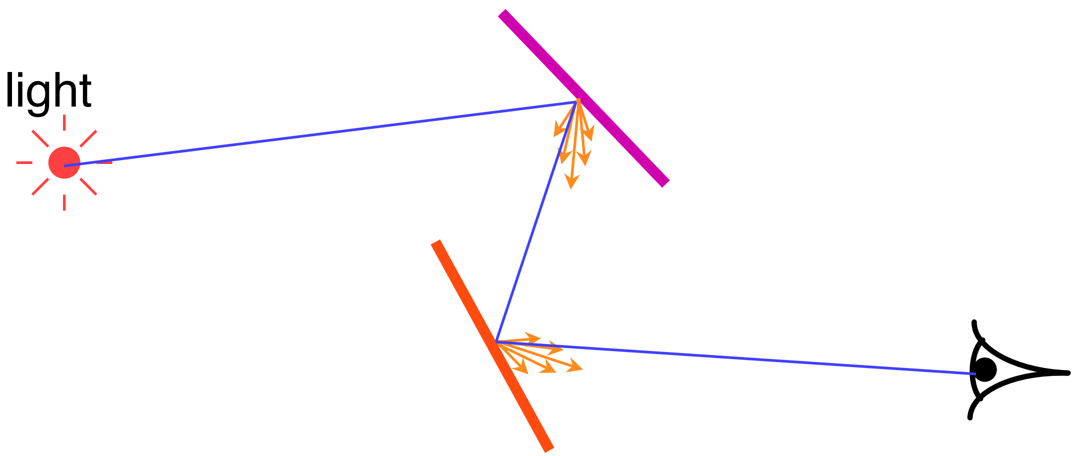


- specular reflection
  - also handled by ray tracing and polygon scan conversion
  - use Phong's approximation to true specular reflection

Handing indirect illumination: I

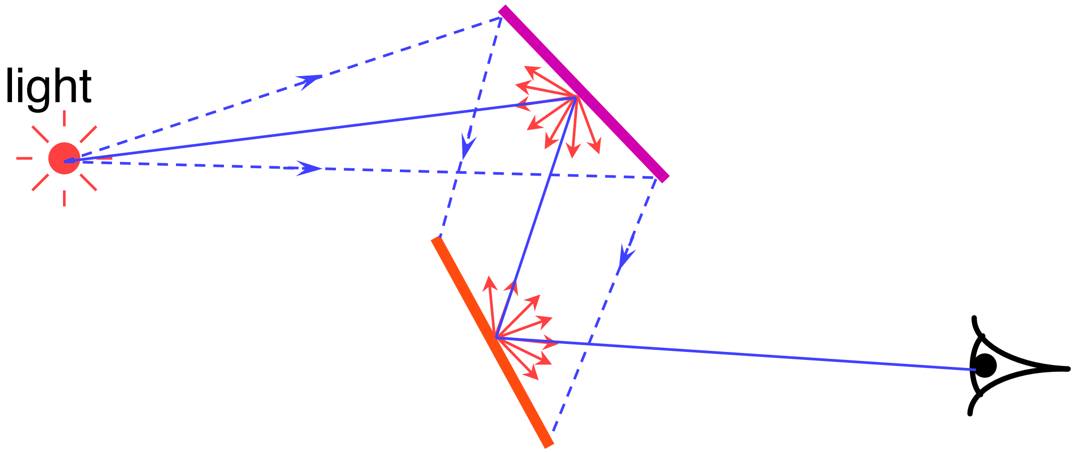


- diffuse to specular
  - handled by distributed ray tracing

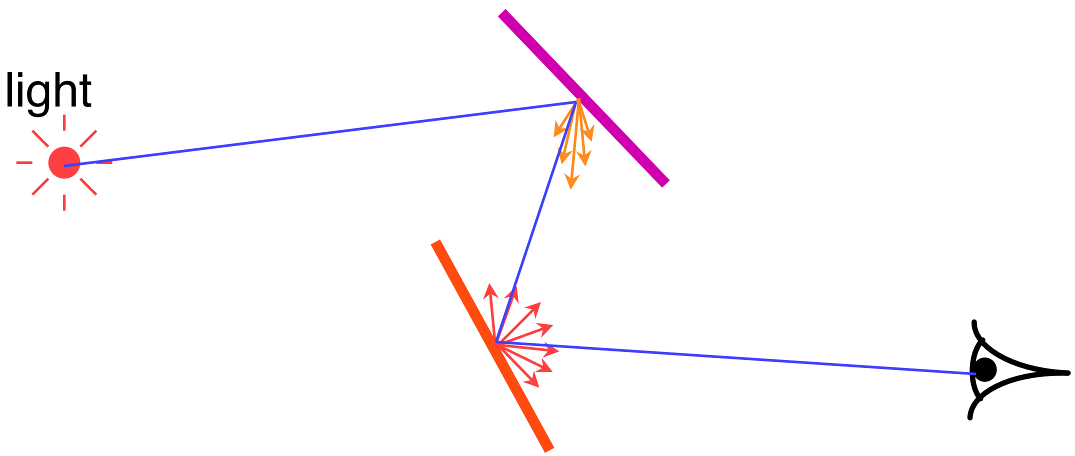


- specular to specular
  - also handled by distributed ray tracing

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

Multiple inter-reflection

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

Introduction to Computer Graphics

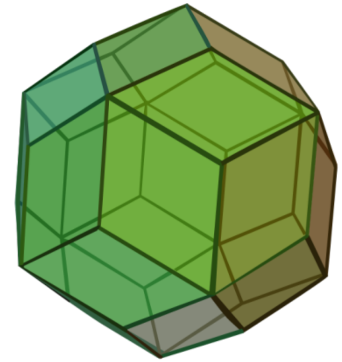
- Background
- Rendering
- **Graphics pipeline**
  - Polygonal mesh models
  - Transformations using matrices in 2D and 3D
  - Homogeneous coordinates
  - Projection: orthographic and perspective
- Graphics hardware and modern OpenGL
- Technology

Unfortunately…

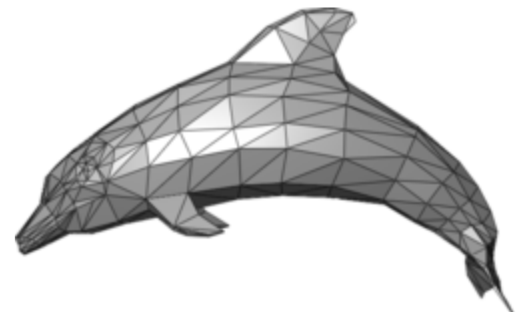
- Ray tracing is computationally expensive
  - used by hobbyists and for super-high visual quality
- Video games and user interfaces need something faster
- So:
  - Model surfaces as polyhedra – meshes of polygons
  - Use composition to build scenes
  - Apply perspective transformation and project into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

Three-dimensional objects

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



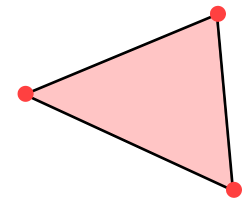
- Polygonal meshes
  - open or closed
  - manifold or non-manifold

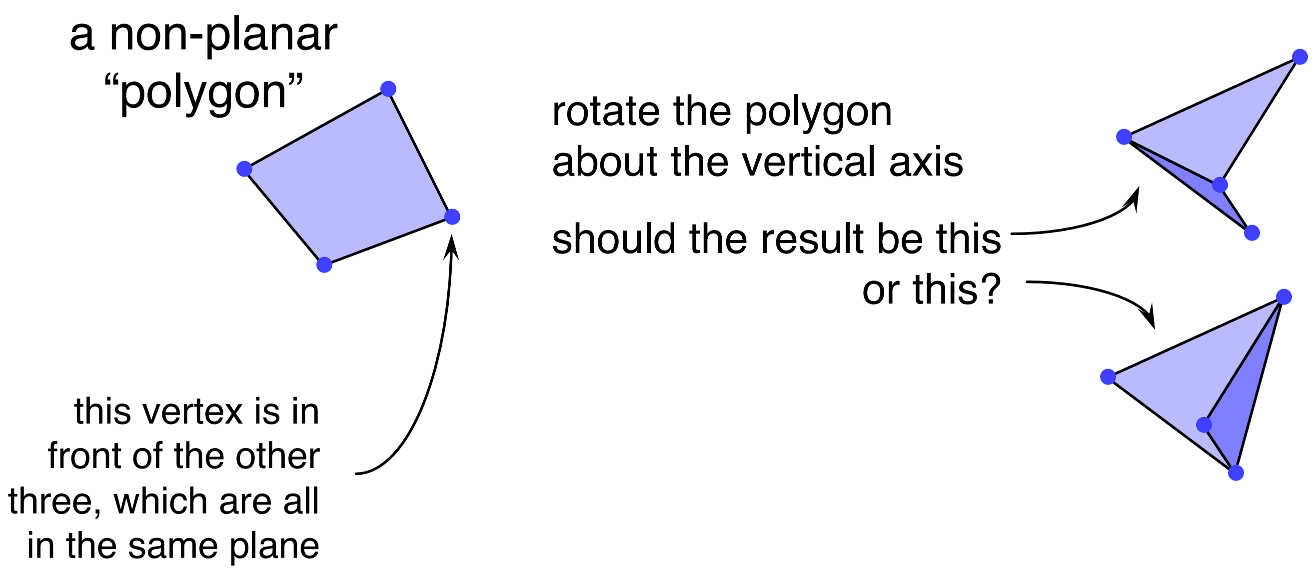


- Curved surfaces
  - must be converted to polygons to be drawn

Surfaces in 3D: polygons

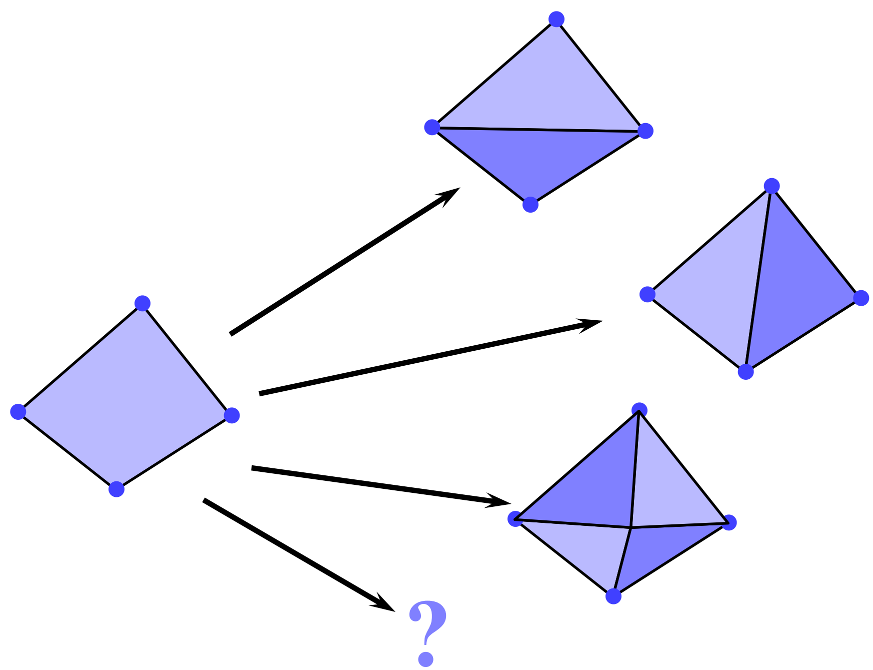
- Easier to consider planar polygons
  - 3 vertices (triangle) must be planar
  - > 3 vertices, not necessarily planar





Splitting polygons into triangles

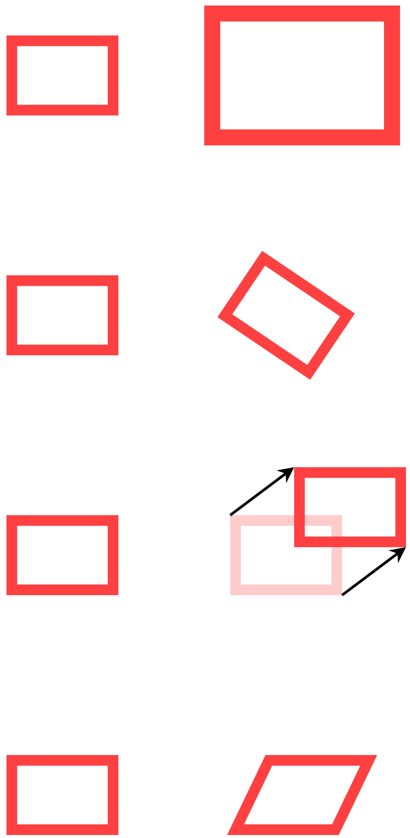
- Most Graphics Processing Units (GPUs) are optimised to draw triangles
- Split polygons with more than three vertices into triangles



which is preferable?

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:

scale

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

shear

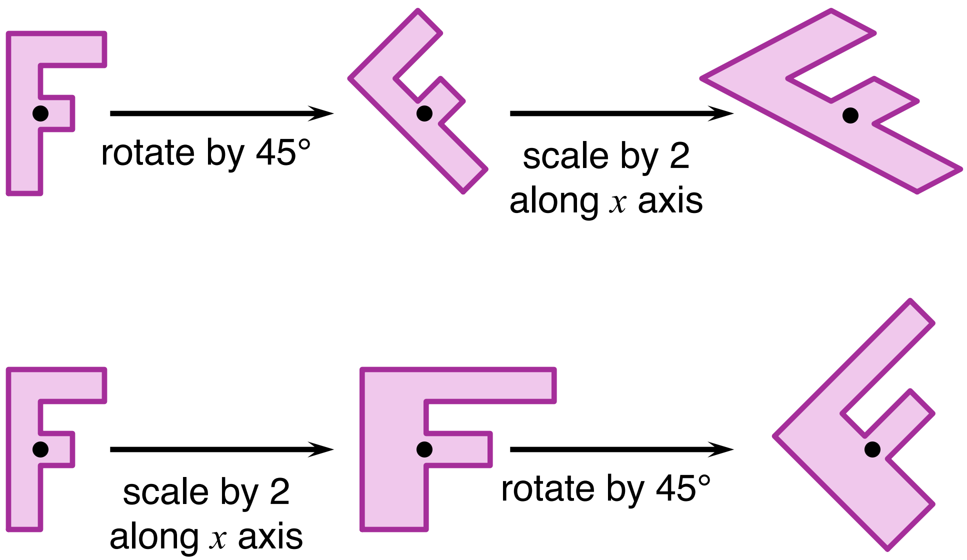
$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} 1 & a & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

scale shear both

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

Transformation are not commutative

- be careful of the order in which you concatenate transformations



rotate then scale

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

scale

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

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

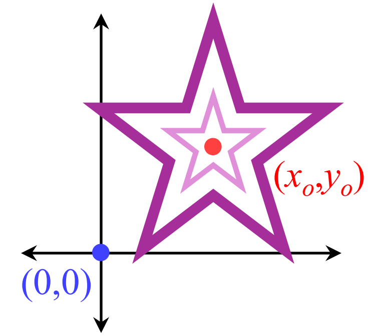
scale then rotate

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

rotate

Scaling about an arbitrary point

- scale by a factor $\boldsymbol{m}$ about point $(\boldsymbol{x_o},\boldsymbol{y_o})$
  - Œ translate point $(\boldsymbol{x_o},\boldsymbol{y_o})$ to the origin
  - ❷ scale by a factor $\boldsymbol{m}$ about the origin
  - ❸ translate the origin to $(\boldsymbol{x_o},\boldsymbol{y_o})$



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

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

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

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

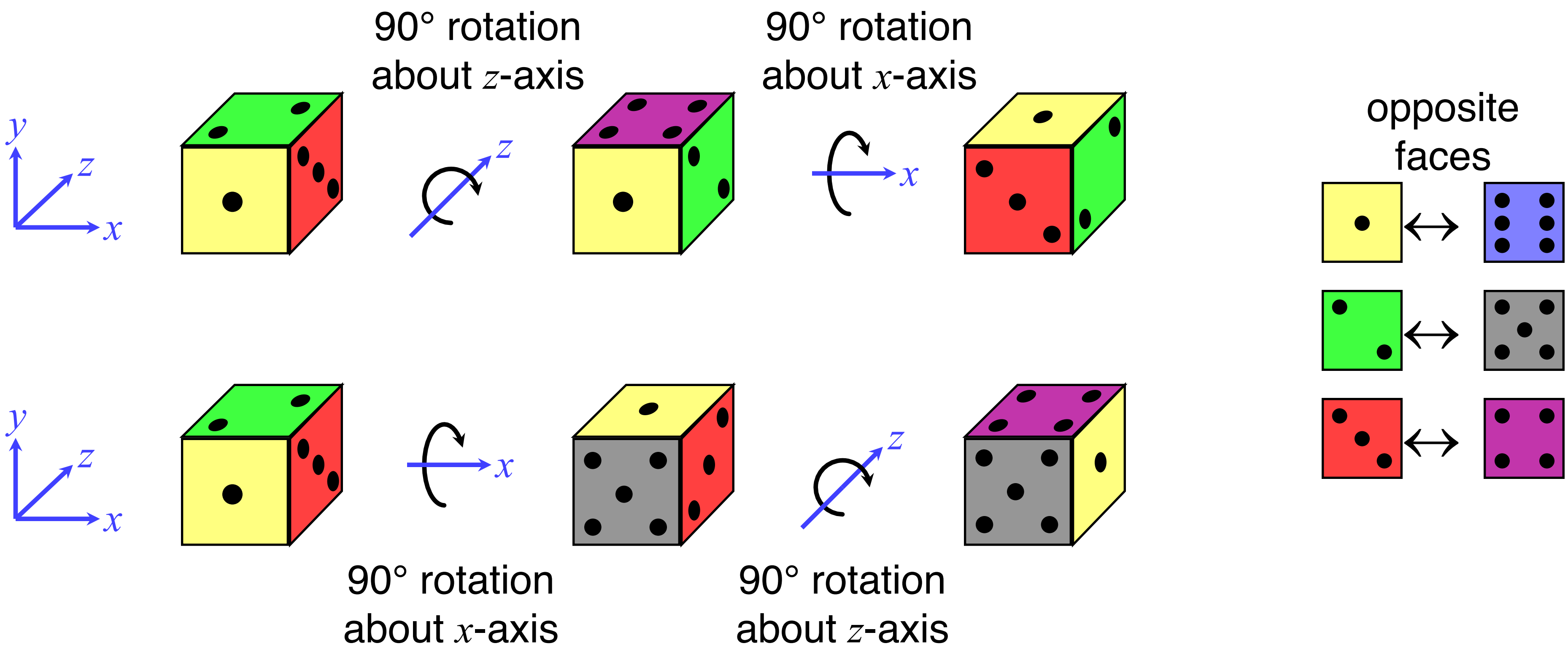
rotation about z-axis

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

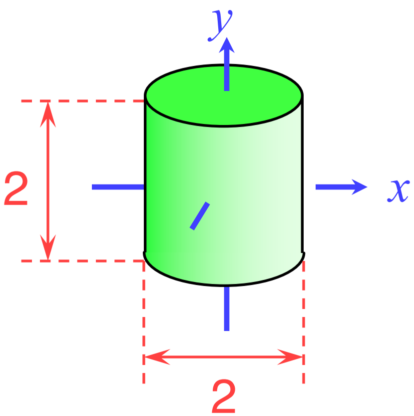
rotation about *y*-axis

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

3D transformations are not commutative



Model transformation 1



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

Model transformation 2

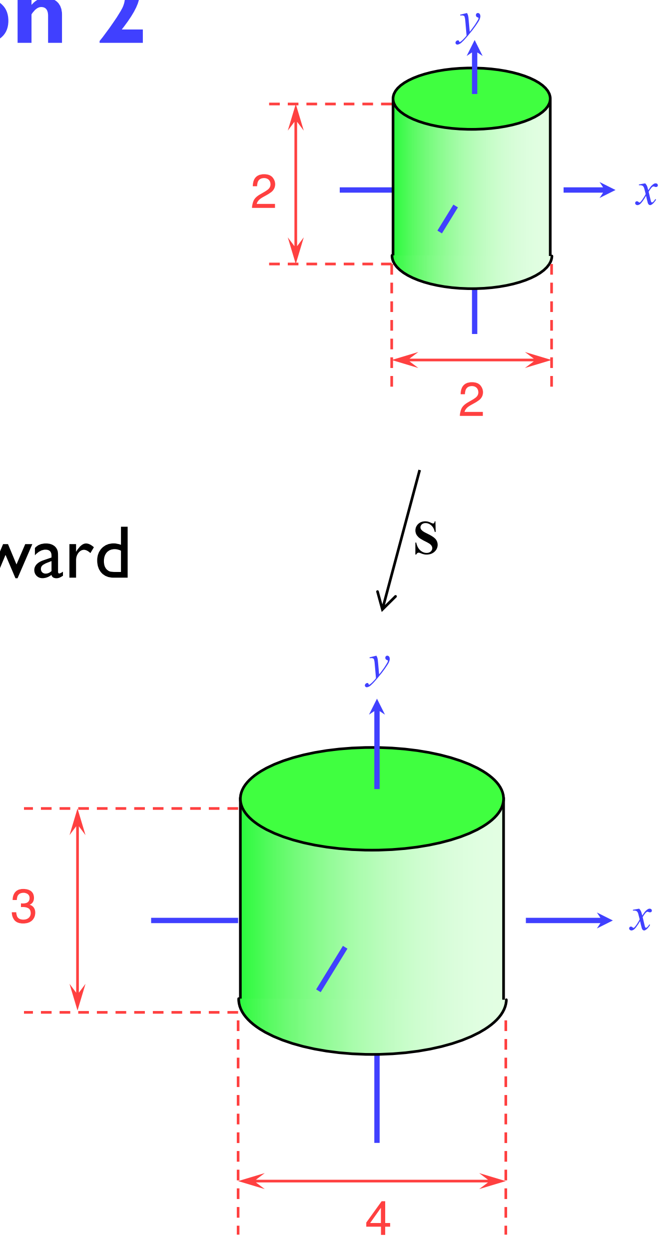
- order is important:
  - scale first
  - rotate
  - translate last
- scaling and translation are straightforward

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

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

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

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



Model transformation 3

- rotation is a multi-step process
  - break the rotation into steps, each of which is rotation about a principal axis
  - work these out by taking the desired orientation back to the original axis-aligned position

      - **the centres of its two ends located at (1,2,3) and (2,4,5)**

  - desired axis: (2,4,5)–(1,2,3) = (1,2,2)

  - original axis: y-axis = (0,1,0)

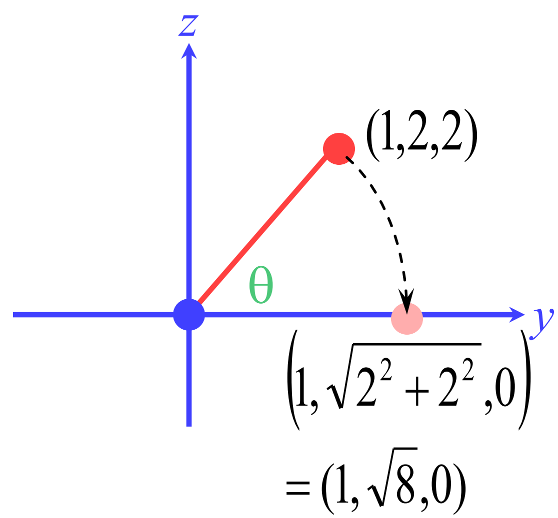
Model transformation 4

- desired axis: (2,4,5)–(1,2,3) = (1,2,2)
- original axis: y-axis = (0,1,0)

- zero the z-coordinate by rotating about the x-axis

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

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

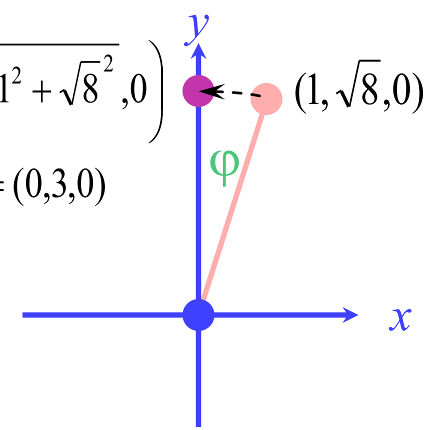


Model transformation 5

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

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

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



$$\left(0, \sqrt{1^2+\sqrt{8}^2}, 0\right) = (0,3,0)$$

Model transformation 6

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

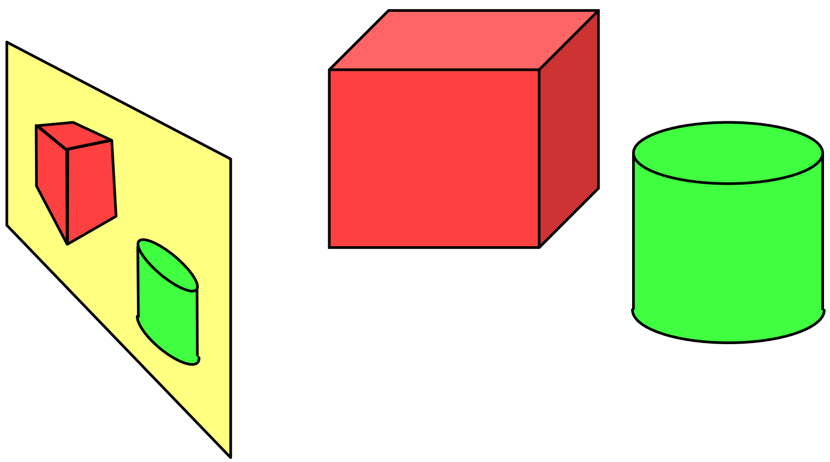
$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \mathbf{T} \times \mathbf{R}_1{}^{-1} \times \mathbf{R}_2{}^{-1} \times \mathbf{S} \times \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

Application: display multiple instances

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

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



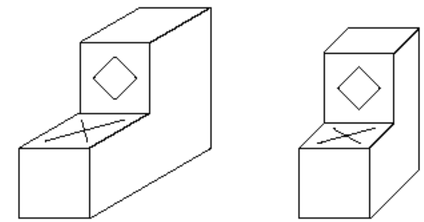
The 3D world is described as a set of (mathematical) objects

e.g. sphere radius (3.4)
centre (0,2,9)

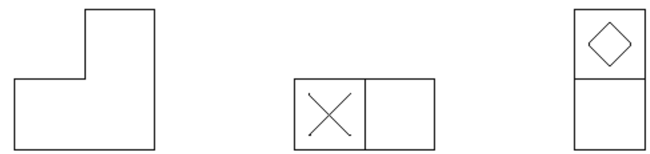
e.g. box size (2,4,3)
centre (7, 2, 9)
orientation (27º, 156º)

Types of projection

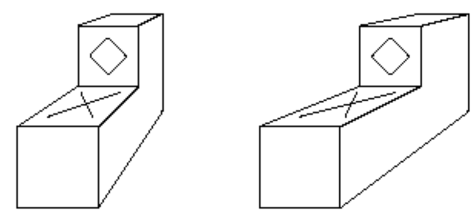
- parallel
  - e.g. $(x, y, z) \rightarrow (x, y)$
  - useful in CAD, architecture, etc
  - looks unrealistic
- perspective
  - e.g. $(x, y, z) \rightarrow (\frac{x}{z}, \frac{y}{z})$
  - things get smaller as they get farther away
  - looks realistic
    - this is how cameras work



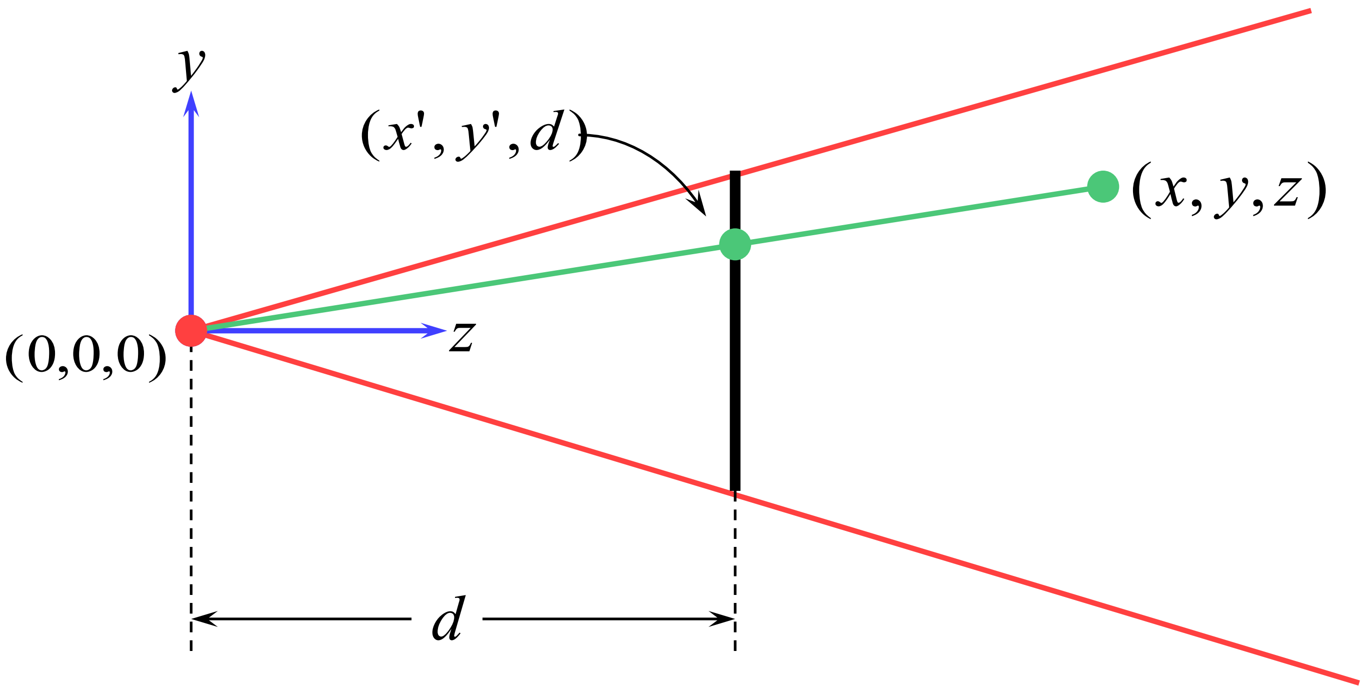
Cavalier projection   Cabinet projection



Parallel to X axis   Parallel to Y axis   Parallel to Z axis



Geometry of perspective projection



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

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

Projection as a matrix operation

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

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

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

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

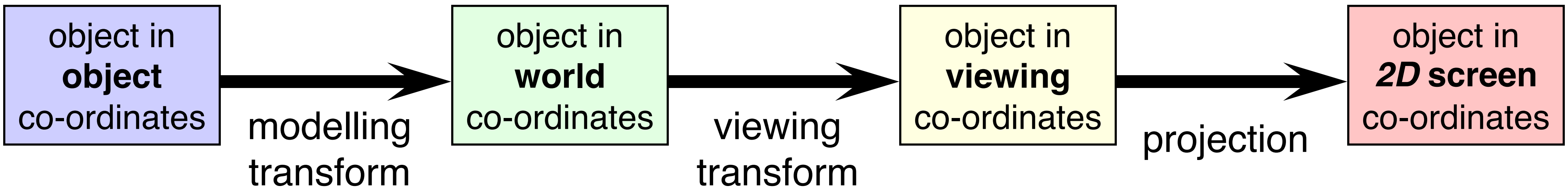
This is useful in the z-buffer algorithm where we need to interpolate $1/z$ values rather than z values.

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

Perspective projection with an arbitrary camera

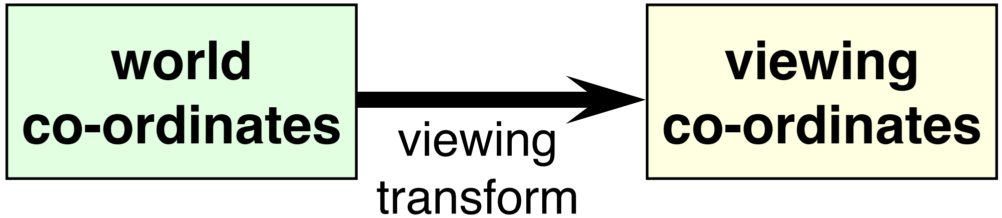
- we have assumed that:
  - screen centre at **(0,0,d)**
  - screen parallel to *xy*-plane
  - *z*-axis into screen
  - *y*-axis up and *x*-axis to the right
  - eye (camera) at origin **(0,0,0)**
- for an arbitrary camera we can either:
  - work out equations for projecting objects about an arbitrary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

A variety of transformations

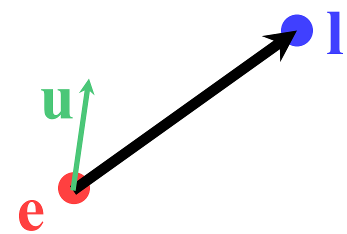


- the modelling transform and viewing transform can be multiplied together to produce a single matrix taking an object directly from object co-ordinates into viewing co-ordinates
- either or both of the modelling transform and viewing transform matrices can be the identity matrix
  - e.g. objects can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

Viewing transformation I



- 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 transformation 2

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

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

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

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

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

Viewing transformation 3

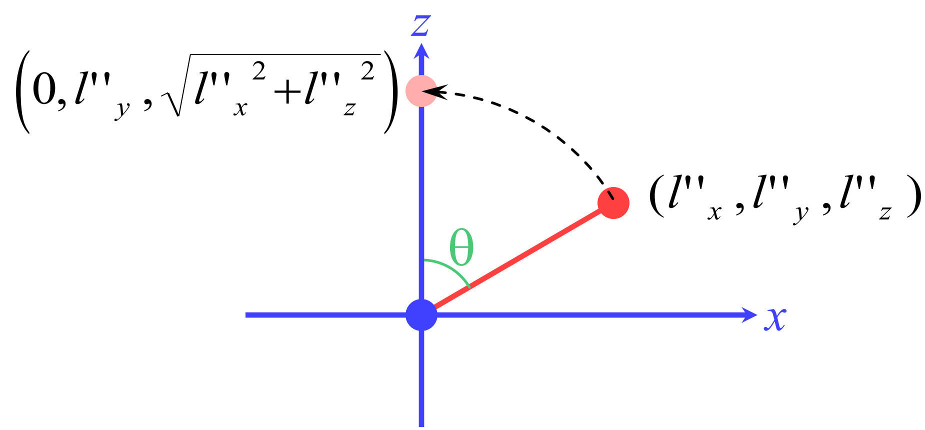
- need to align line $\overline{\mathbf{el}}$ with *z*-axis
  - first transform e and l into new co-ordinate system

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

  - then rotate e''l'' into *yz*-plane, rotating about *y*-axis

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

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



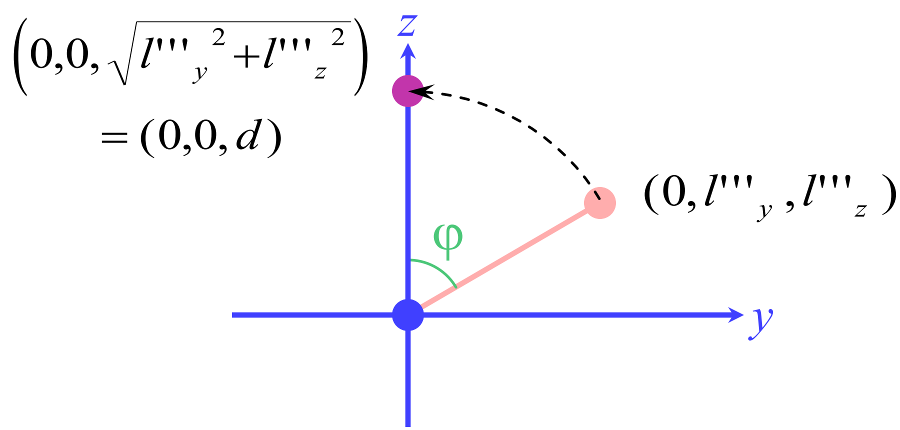
Viewing transformation 4

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

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

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

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



Viewing transformation 5

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

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

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

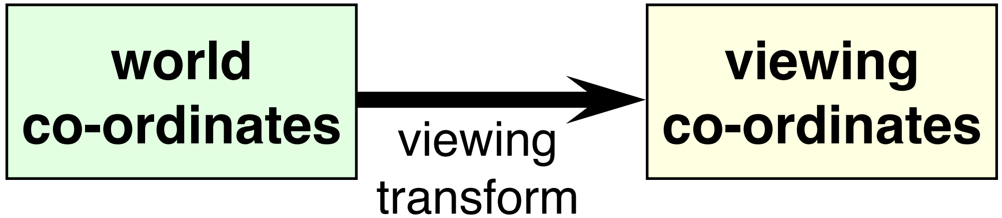
$$\psi = \arccos \frac{u''''_y}{\sqrt{u''''^2_x + u''''^2_y}}$$

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

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

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

Viewing transformation 6



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

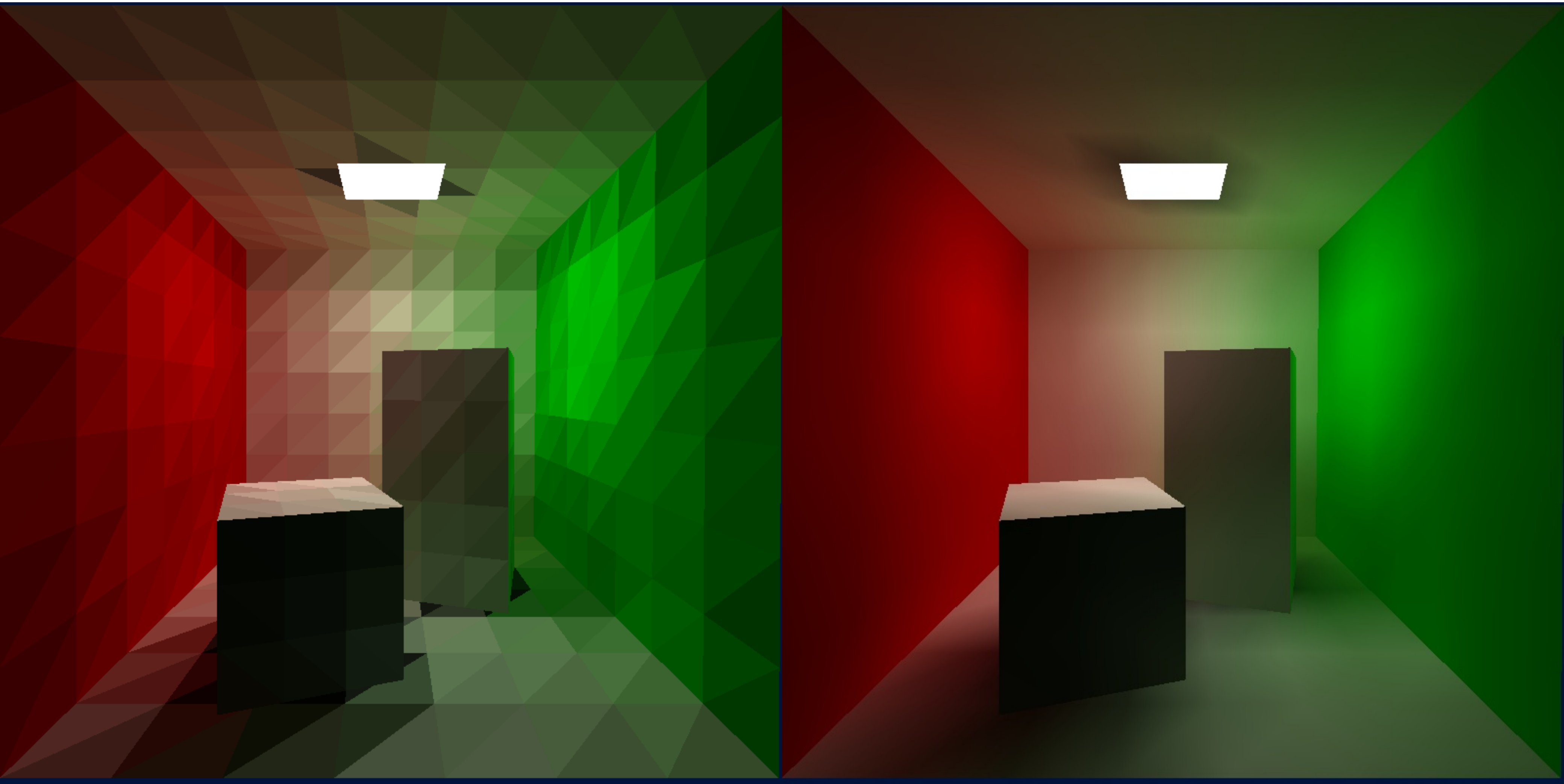
$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \mathbf{R}_3 \times \mathbf{R}_2 \times \mathbf{R}_1 \times \mathbf{S} \times \mathbf{T} \times \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

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

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

Illumination & shading

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



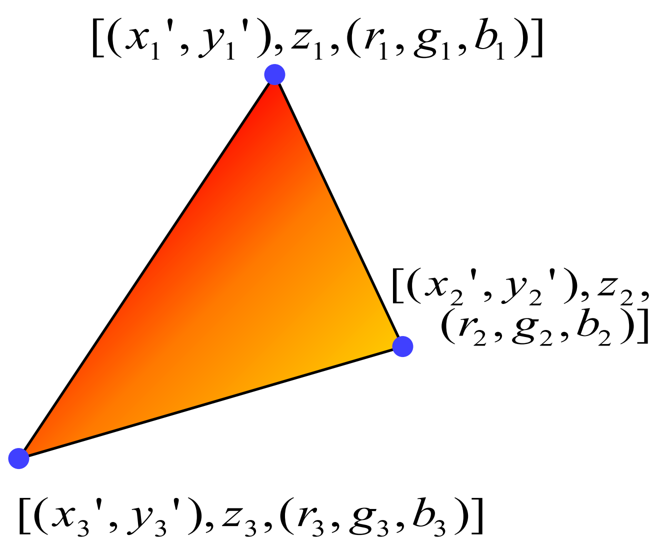
Illumination & shading

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

Gouraud shading

- for a polygonal model, calculate the diffuse illumination at each ***vertex***
  - calculate the normal at the vertex, and use this to calculate the diffuse illumination at that point
  - normal can be calculated directly if the polygonal model was derived from a curved surface

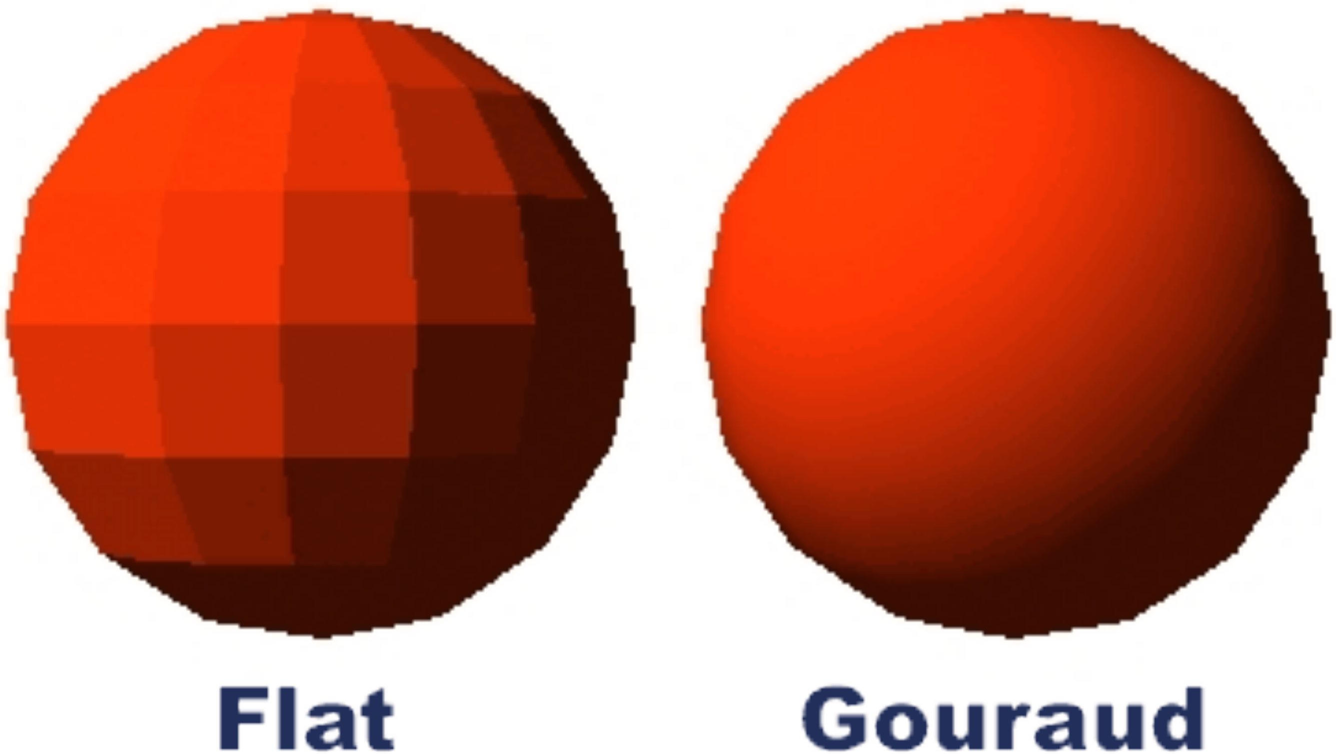
- interpolate the colour between the vertices across the polygon
- surface will look smoothly curved
  - rather than looking like a set of polygons
  - surface outline will still look polygonal



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

Flat *vs* Gouraud shading

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

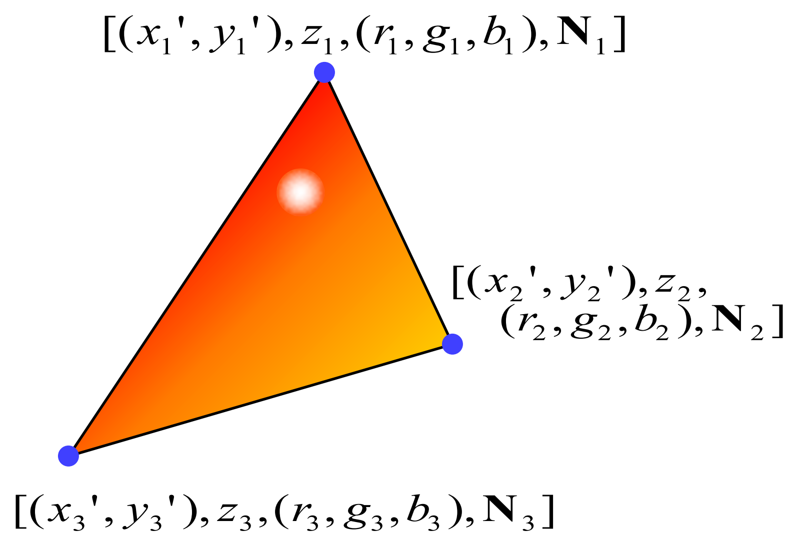


http://computer.howstuffworks.com/question484.htm

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

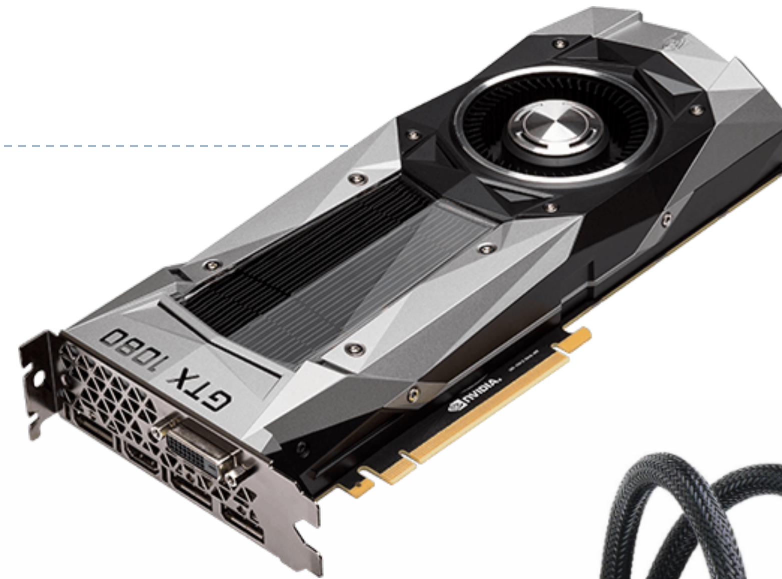


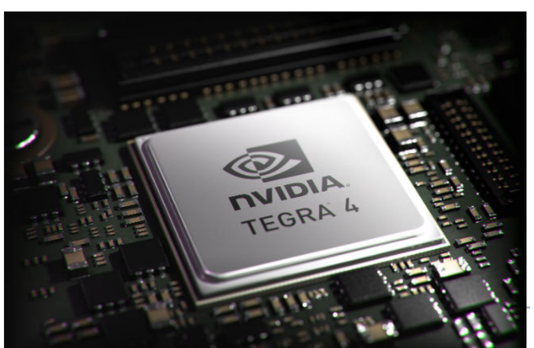
Introduction to Computer Graphics

- Background
- Rendering
- Graphics pipeline
- **Graphics hardware and modern OpenGL**
  - **GPU & APIs**
  - **Example OpenGL code**
  - **OpenGL Rendering pipeline**
  - **GLSL**
  - **Transformations & vertex shaders**
  - **Raster buffers**
  - **Textures**
- Technology

What is a GPU?

- Graphics Processing Unit
- Like CPU (Central Processing Unit) but for processing graphics
- Optimized for floating point operations on large arrays of data
  - Vertices, normals, pixels, etc.

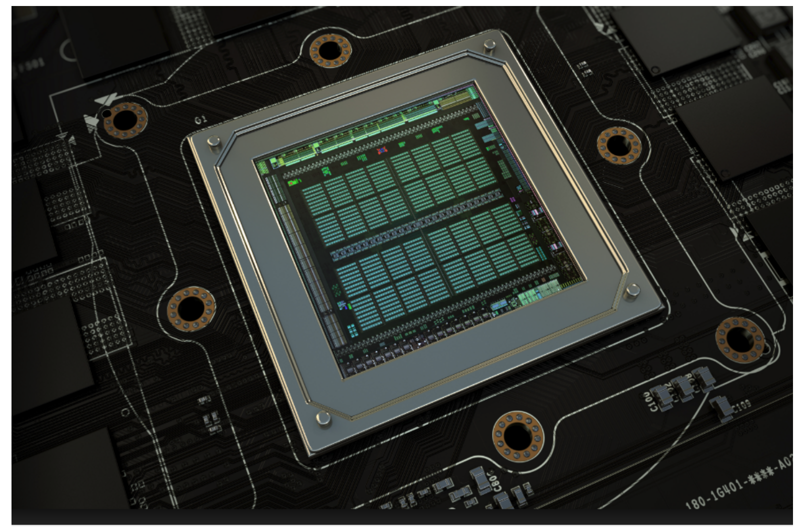




Transistor count

Intell 8-core Core i7 Haswell-E

2,600,000,000 transistors



Nvidia GeForce GTX Titan X

8,000,000,000 transistors

What does a GPU do

- Performs all low-level tasks & a lot of high-level tasks
  - Clipping, rasterisation, hidden surface removal, …
    - Essentially draws millions of triangles very efficiently
  - Procedural shading, texturing, animation, simulation, …
  - Video rendering, de- and encoding, deinterlacing, ...
  - Physics engines
- Full programmability at several pipeline stages
  - In the recent years GPUs became like CPU
    - fully programmable
    - but optimized for massively parallel operations

What makes GPU so fast?

- 3D rendering can be very efficiently parallelized
  - Millions of pixels
  - Millions of triangles
  - Many operations executed at the same time
- This is why modern GPUs
  - Contain between hundreds and thousands of SIMD processors
    - Single Instruction Multiple Data – operate on large arrays of data
  - >>400 GB/s memory access
    - This is much higher bandwidth than CPU
    - But peak performance can be expected for very specific operations

GPU APIs (Application Programming Interfaces)

OpenGL

- Multi-platform
- Open standard API
- Focus on general 3D applications
  - Open GL driver manages the resources

DirectX

- Microsoft Windows / Xbox
- Proprietary API
- Focus on games
  - Application manages resources

- Nearly the same functionality
- Similar performance

One more API

- Vulkan – cross platform, open standard
- Low-overhead API for high performance 3D graphics
- Compared to OpenGL / DirectX
  - Reduces CPU load
  - Better support of multi-CPU-core architectures
  - Finer control of GPU
- But
  - The code for drawing a few primitives can take 1000s line of code
  - Intended for game engines and code that must be very well optimized

GPU for general computing

- OpenGL and DirectX are not meant to be used for general purpose computing
  - Example: physical simulation
- CUDA – NVidia's architecture for parallel computing
  - C-like programming language
  - With special API for parallel instructions
  - Requires NVidia GPU
- OpenCL – Similar to CUDA, but open standard
  - Can run on both GPU and CPU
  - Supported by AMD, Intel and Nvidia, Qualcomm, Apple, …

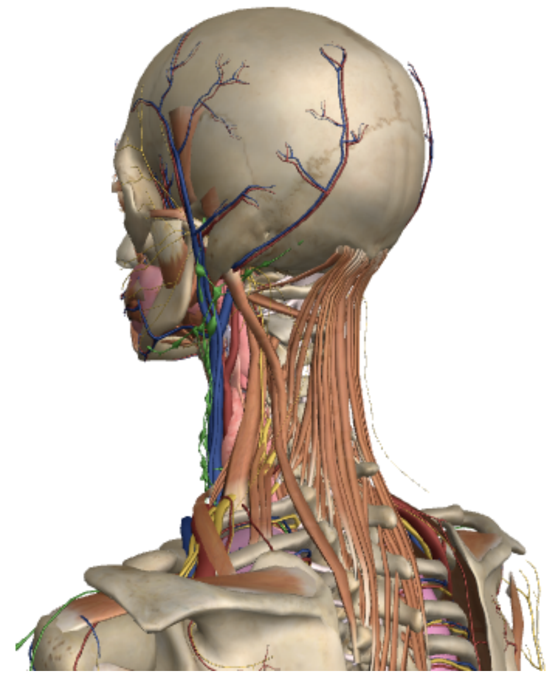
GPU and mobile devices

- OpenGL ES 1.0-3.2
  - Stripped version of OpenGL
  - Removed functionality that is not strictly necessary on mobile devices
- Devices
  - iOS: iPad, iPhone, iPod Touch
  - Android phones
  - PlayStation 3
  - Nintendo 3DS
  - and many more

OpenGL ES 2.0 rendering (iOS)

WebGL

- JavaScript library for 3D rendering in a web browser
- WebGL 1.0 - based on OpenGL ES 2.0
- Most modern browsers support WebGL
  - Microsoft browsers are lagging behind
- Potentially could be used to create 3D games in a browser
  - and replace Adobe Flash



http://zygotebody.com/

OpenGL in Java

- Standard Java API does not include OpenGL interface
- But several wrapper libraries exist
  - Java OpenGL – JOGL
  - Lightweight Java Game Library - LWJGL
- We will use LWJGL 3
  - Seems to be better maintained
  - Access to other APIs (OpenCL, OpenAL, …)

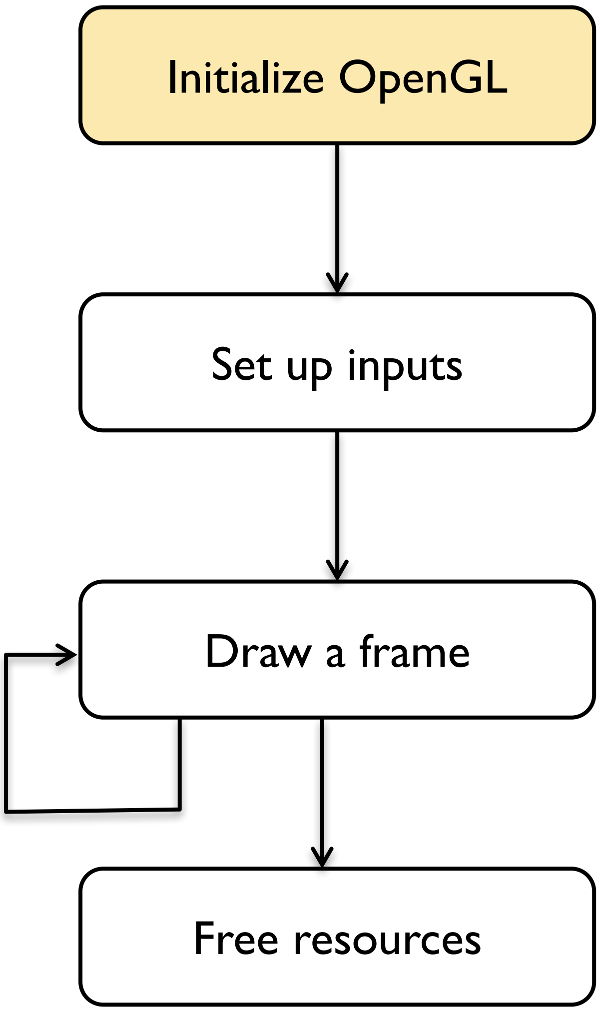
- We also need a linear algebra library
  - JOML – Java OpenGL Math Library
  - Operations on 2, 3, 4-dimensional vectors and matrices

OpenGL History

- Proprietary library IRIS GL by SGI
- OpenGL 1.0 (1992)
- OpenGL 1.2 (1998)
- OpenGL 2.0 (2004)
  - GLSL
  - Non-power-of-two (NPOT) textures
- OpenGL 3.0 (2008)
  - Major overhaul of the API
  - Many features from previous versions depreciated
- OpenGL 3.2 (2009)
  - Core and Compatibility profiles
  - Geometry shaders
- OpenGL 4.0 (2010)
  - Catching up with Direct3D 11
- OpenGL 4.5 (2014)

OpenGL example code - overview

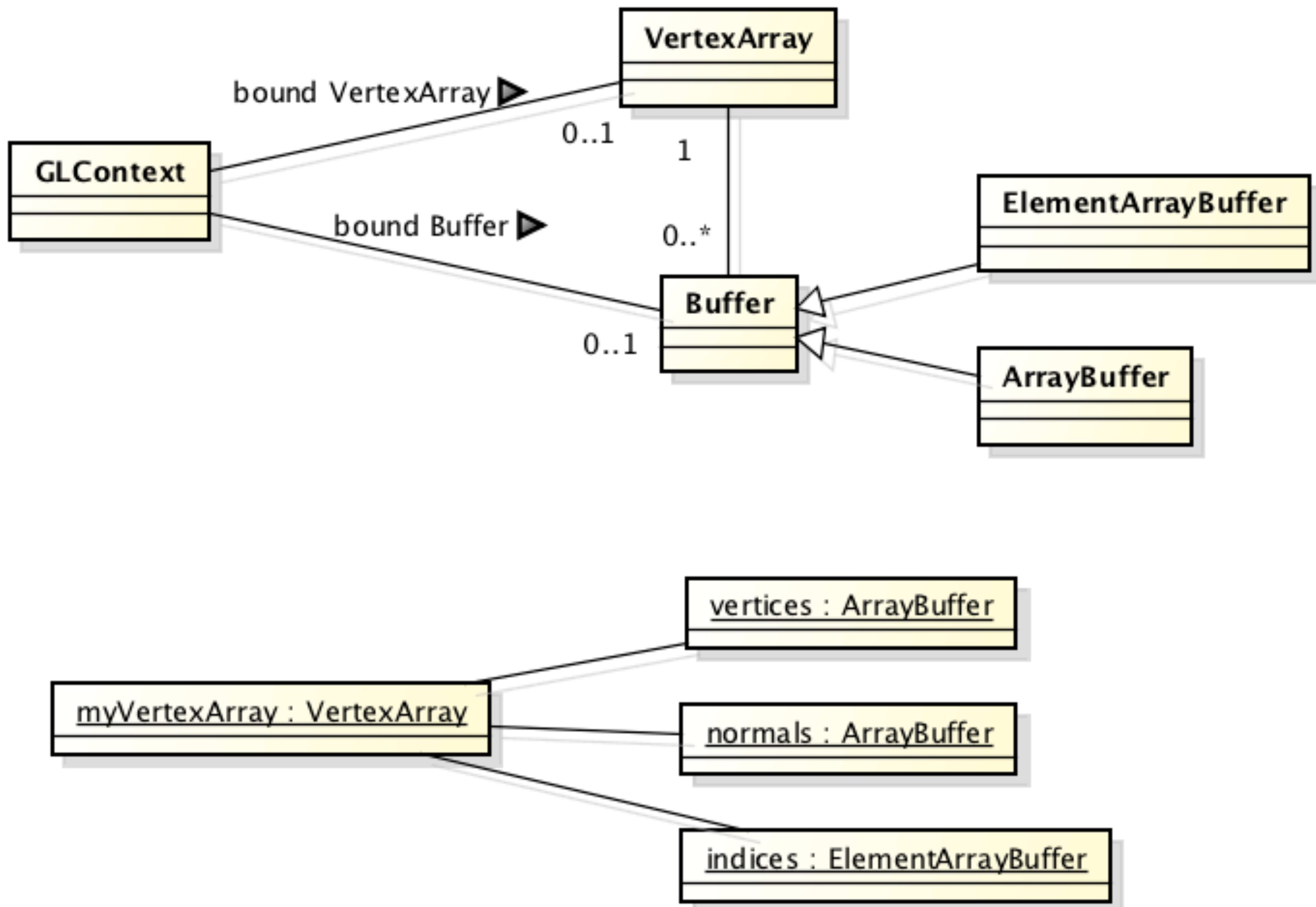
Let us draw some triangles



- Initialize rendering window & OpenGL context
- **Send the geometry (vertices, triangles, normals) to the GPU**
- Load and compile Shaders

- Clear the screen buffer
- Set the model-view-projection matrix
- Render geometry
- Flip the screen buffers

Geometry objects in OpenGL (OO view)



OpenGL as a state-machine

If OpenGL was OO API:

```
VertexArray va = new VertexArray();

ArrayBuffer vertices = new ArrayBuffer( my_data );

va.add( vertices );
```

But it is not, and you must do:

```
int va = glGenVertexArrays();
glBindVertexArray(va); // va becomes "active" VertexArray

int vertices = glGenBuffers();
glBindBuffer(GL_ARRAY_BUFFER, vertices);  // This adds vertices to currently bound VertexArray
```

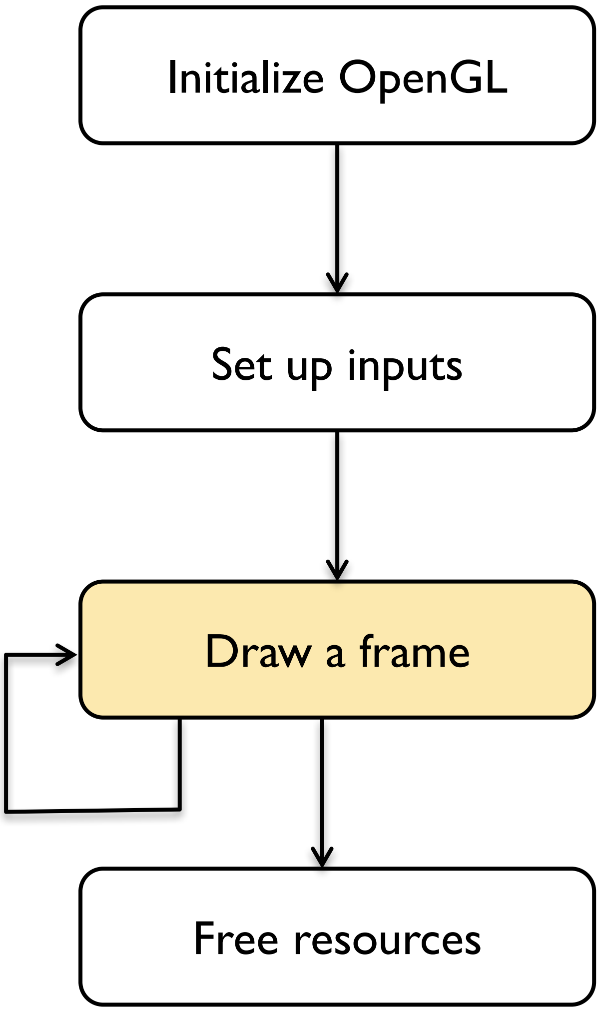
A more complete example

```
int vertexArrayObj = glGenVertexArrays();  // Create a name
glBindVertexArray(vertexArrayObj);          // Bind a VertexArray

float[] vertPositions = new float[] { -1, -1, 0,  0, 1, 0,  1, -1, 0 }; // x, y, z, x, y, z …
// Java specific code for transforming float[] into an OpenGL-friendly format
FloatBuffer vertex_buffer = BufferUtils.createFloatBuffer(vertPositions.length);
vertex_buffer.put(vertPositions); // Put the vertex array into the CPU buffer
vertex_buffer.flip();             // "flip" is used to change the buffer from read to write mode

int vertex_handle = glGenBuffers();     // Get an OGL name for a buffer object
glBindBuffer(GL_ARRAY_BUFFER, vertex_handle);  // Bring that buffer object into existence on GPU
glBufferData(GL_ARRAY_BUFFER, vertex_buffer, GL_STATIC_DRAW);  // Load the GPU buffer object with data
```

Note on LWJGL

- The OpenGL functions and constants can be found in the LWJGL packages:
  - org.lwjgl.opengl.GL11
  - org.lwjgl.opengl.GL15
  - org.lwjgl.opengl.GL20
  - org.lwjgl.opengl.GL30
  - …
- For simplicity, package names are omitted in all examples shown in these slides

Let us draw some triangles



- Initialize rendering window & OpenGL context
- Send the geometry (vertices, triangles, normals) to the GPU
- Load and compile Shaders

- Clear the screen buffer
- Set the model-view-projection matrix
- Render geometry
- Flip the screen buffers

Rendering 1 of 2

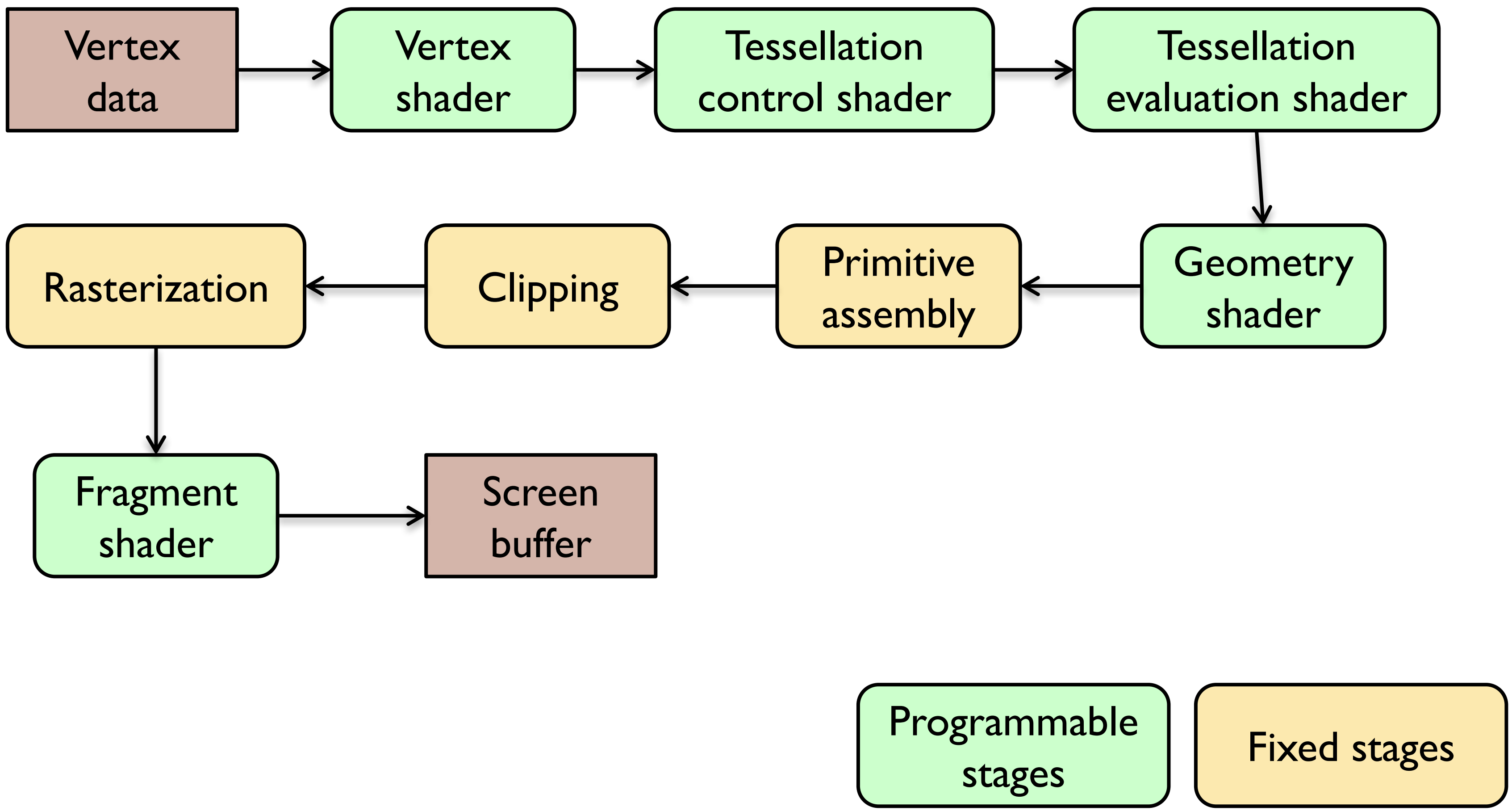
```
// Step 1: Pass a new model-view-projection matrix to the vertex shader
Matrix4f mvp_matrix; // Model-view-projection matrix
mvp_matrix = new
Matrix4f(camera.getProjectionMatrix()).mul(camera.getViewMatrix());

int mvp_location = glGetUniformLocation(shaders.getHandle(), "mvp_matrix");
FloatBuffer mvp_buffer = BufferUtils.createFloatBuffer(16);
mvp_matrix.get(mvp_buffer);
glUniformMatrix4fv(mvp_location, false, mvp_buffer);

// Step 2: Clear the buffer
glClearColor(1.0f, 1.0f, 1.0f, 1.0f); // Set the background colour to dark grey
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

Rendering 2 of 2

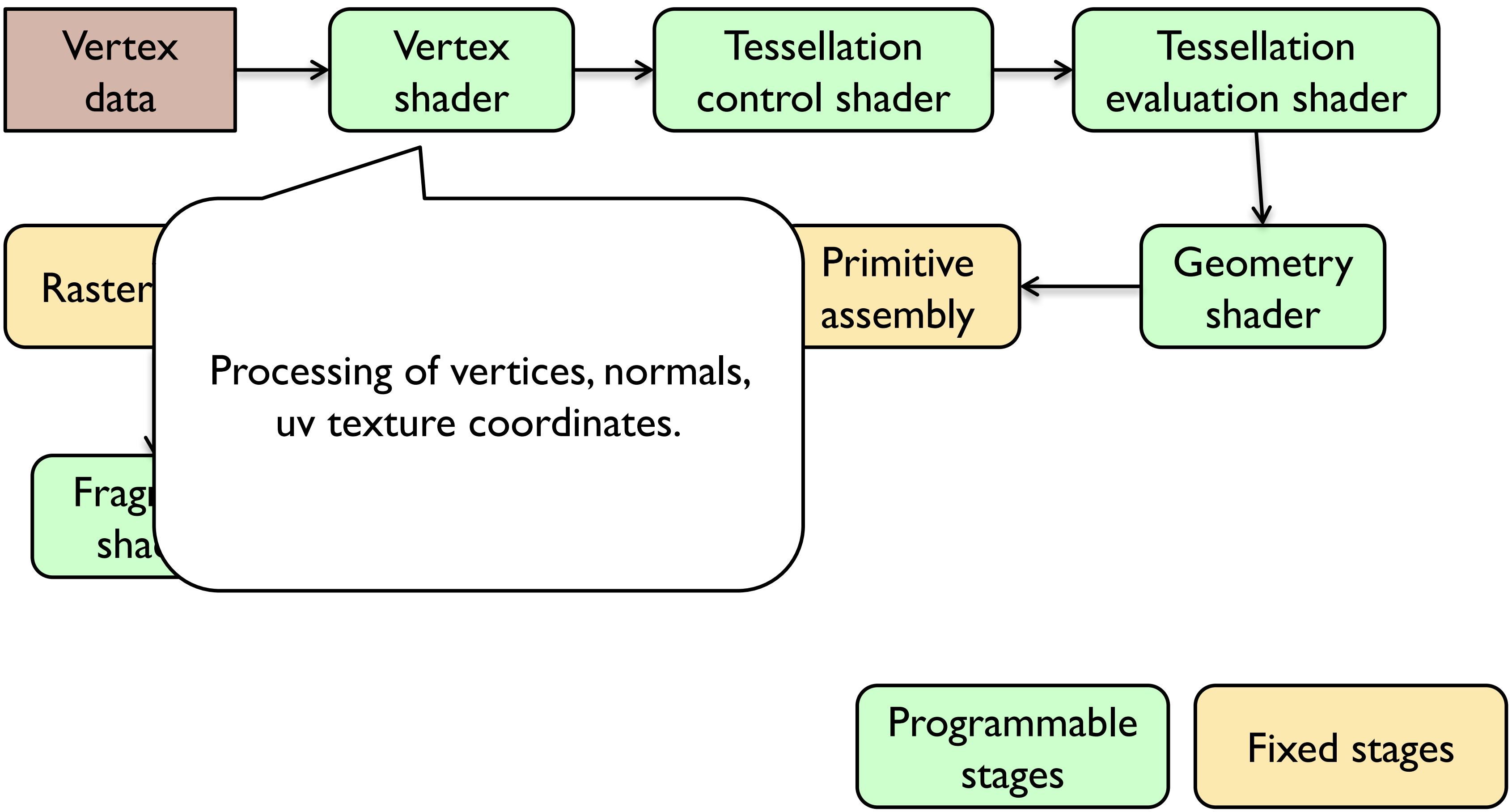
```
// Step 3: Draw our VertexArray as triangles
glBindVertexArray(vertexArrayObj); // Bind the existing VertexArray object
glDrawElements(GL_TRIANGLES, no_of_triangles, GL_UNSIGNED_INT, 0); // Draw it as triangles
glBindVertexArray(0); // Remove the binding

// Step 4: Swap the draw and back buffers to display the rendered image
glfwSwapBuffers(window);
glfwPollEvents();
```

OpenGL rendering pipeline

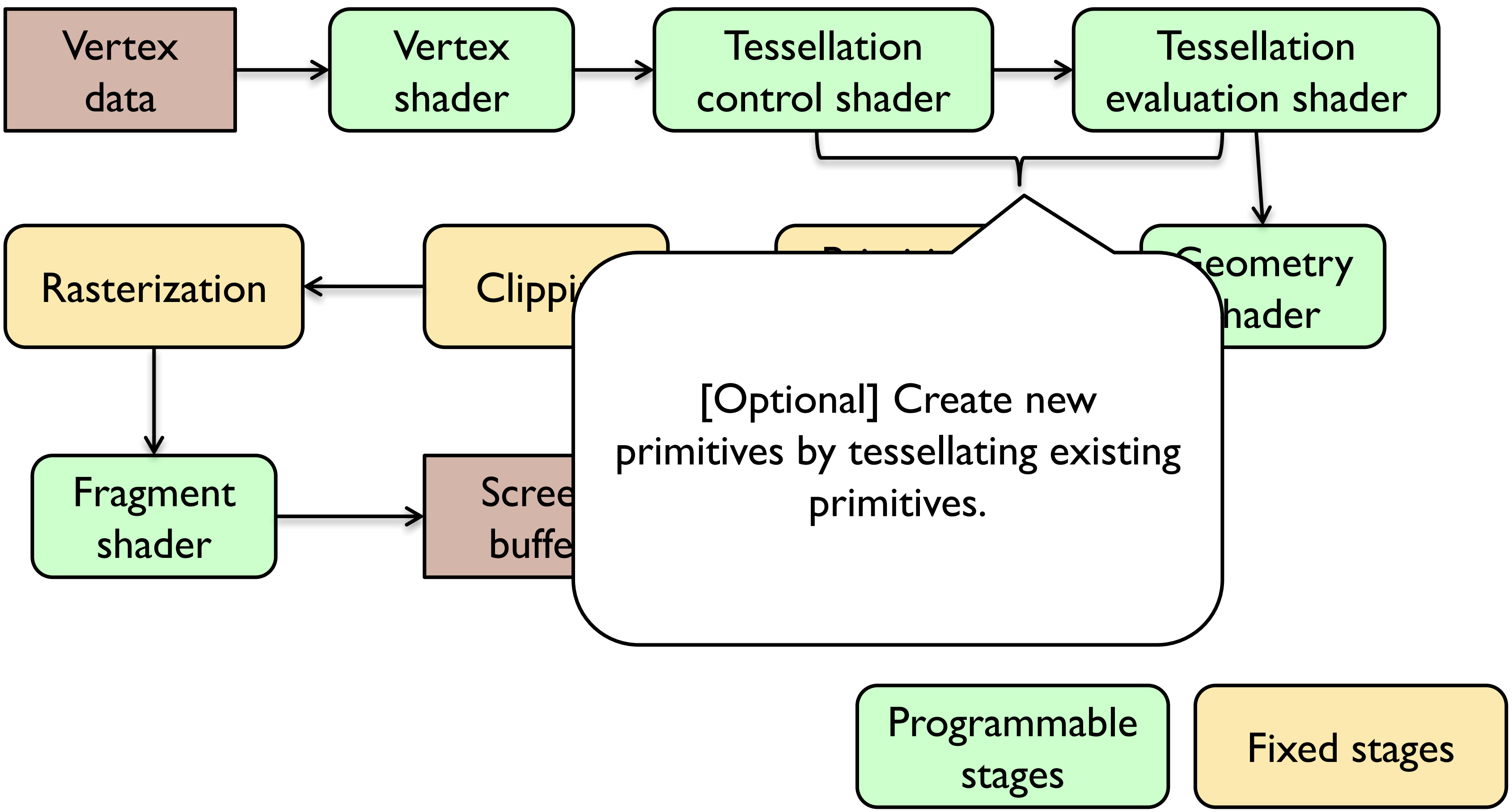
OpenGL rendering pipeline



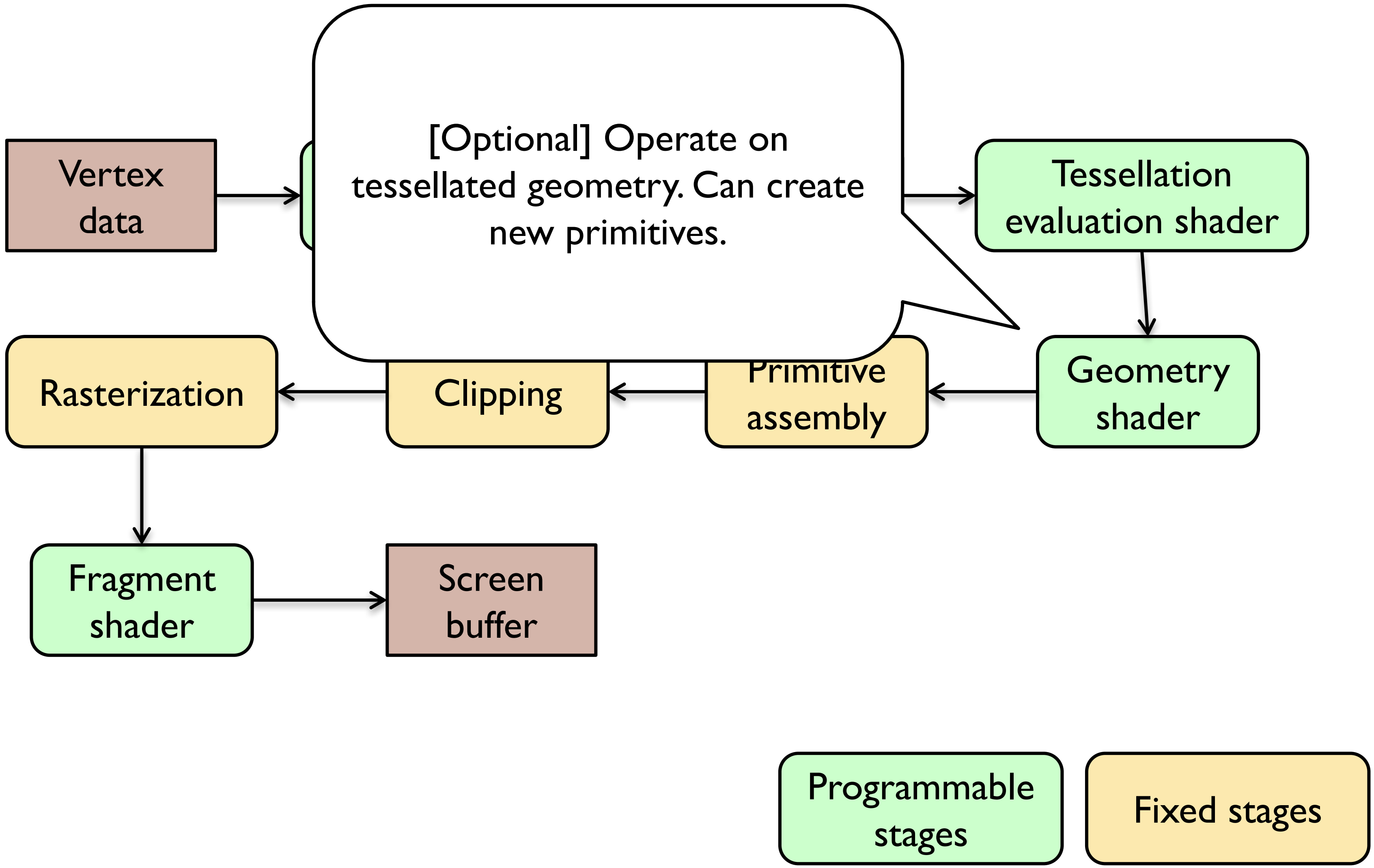
OpenGL rendering pipeline



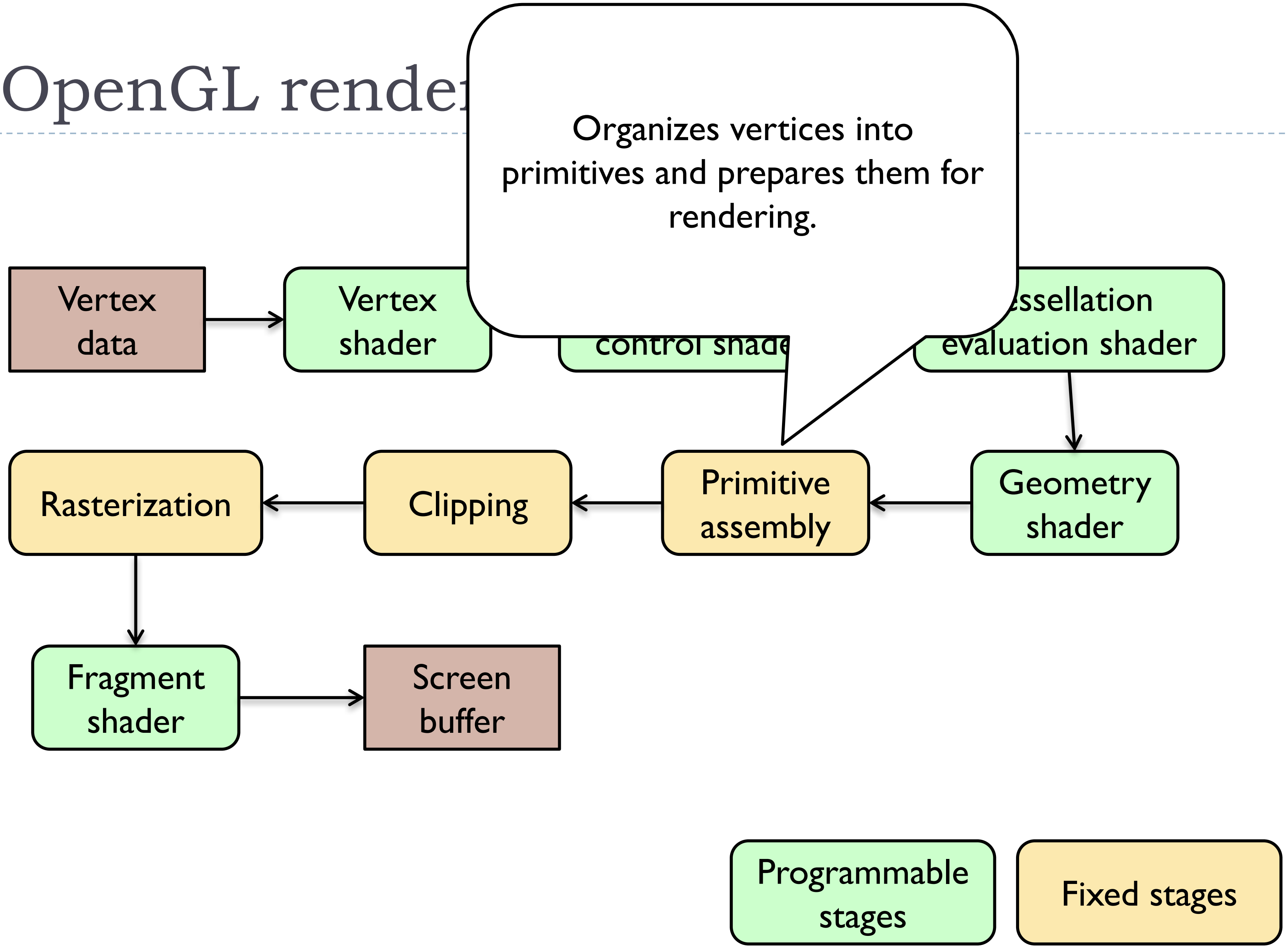
OpenGL rendering pipeline



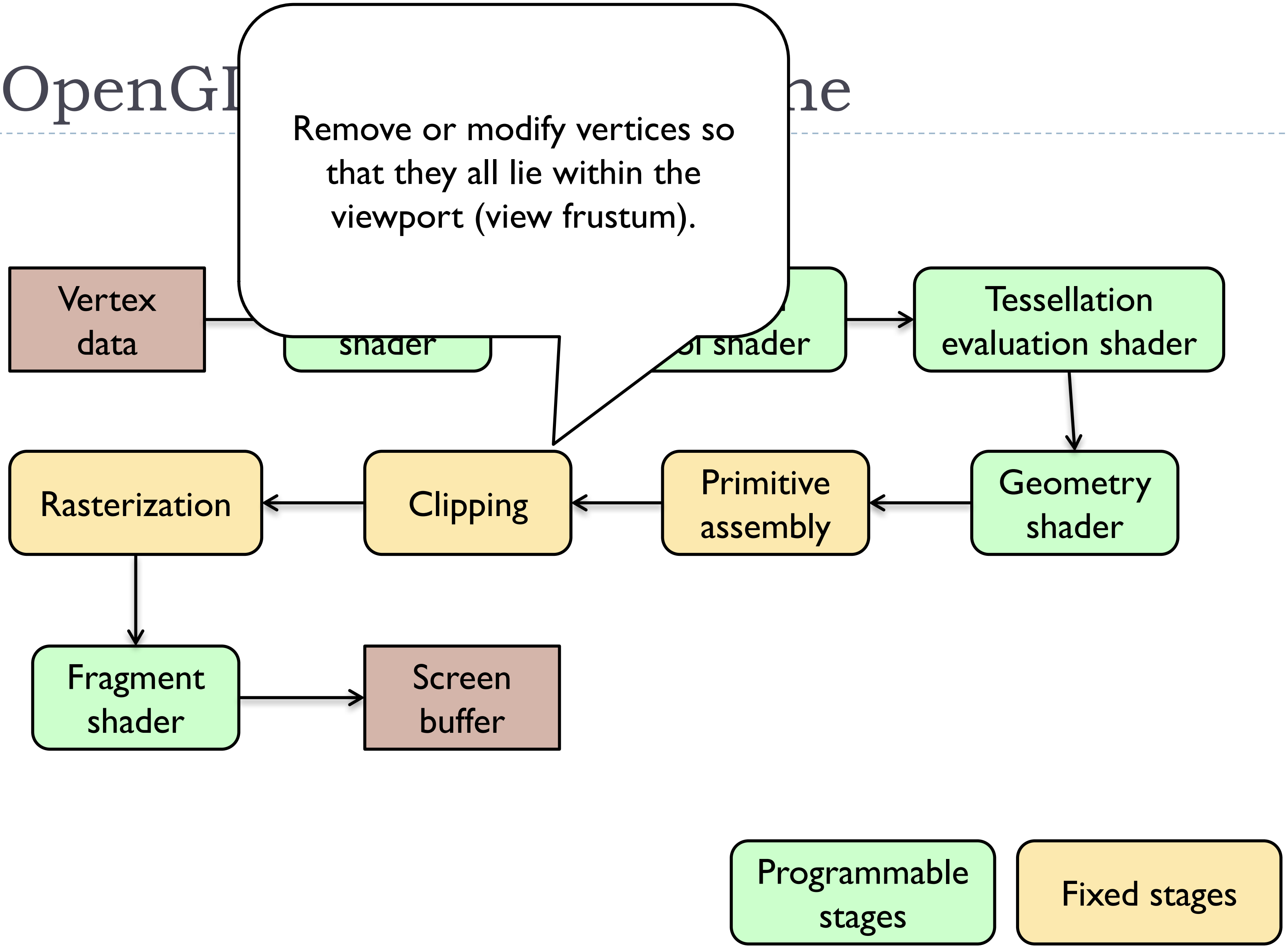
OpenGL rendering pipeline



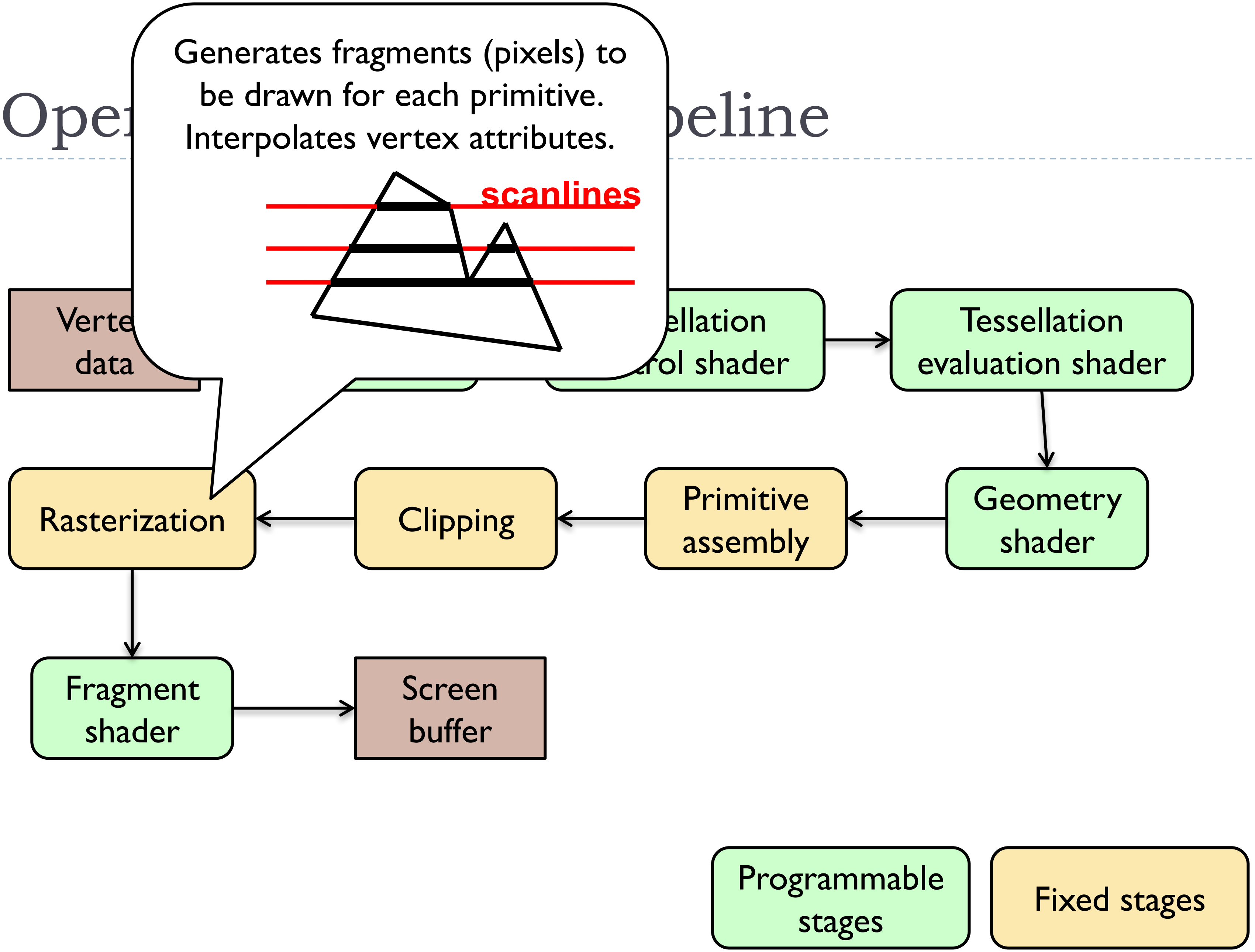
OpenGL rende



OpenGL … ne

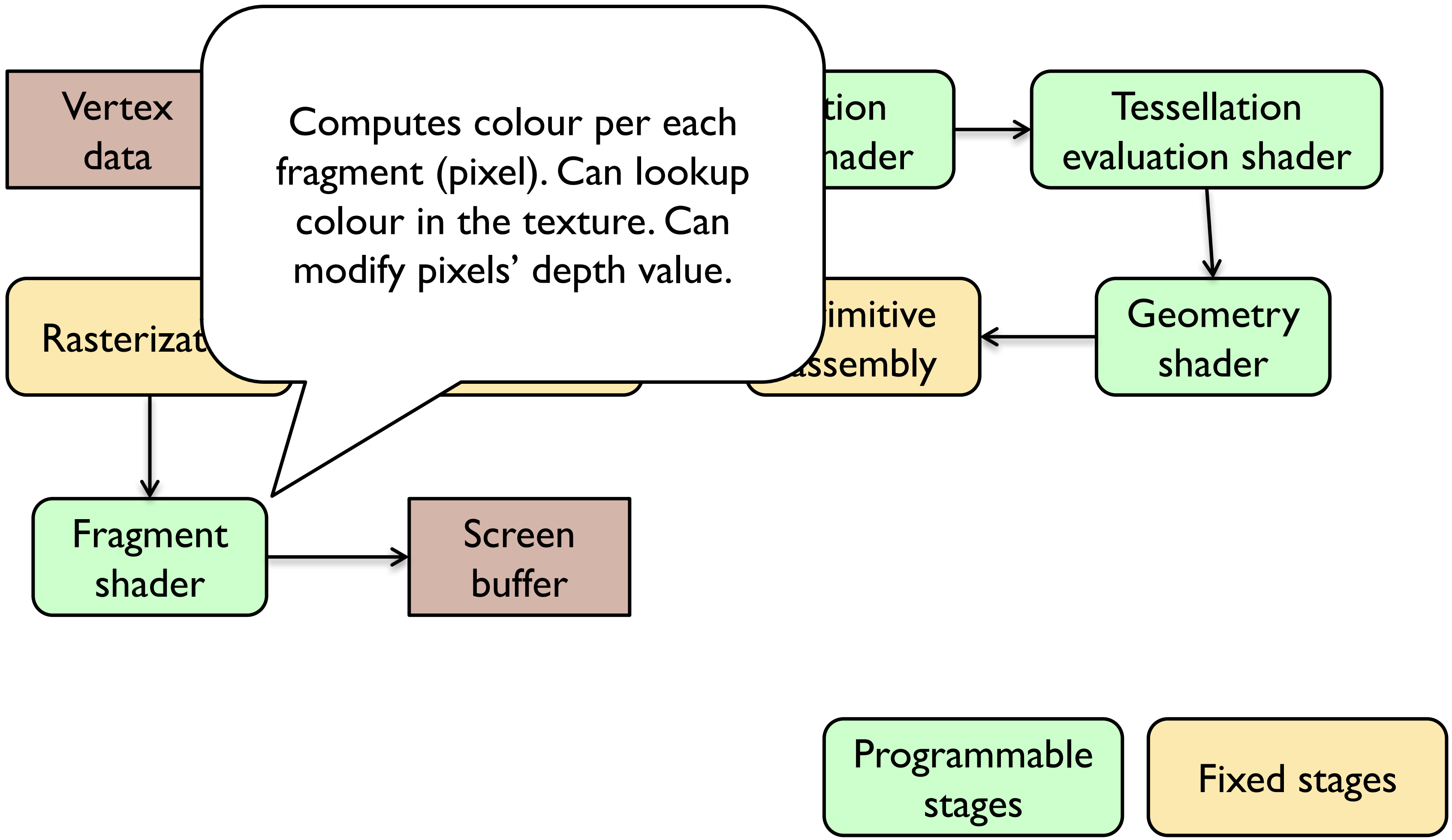

Remove or modify vertices so that they all lie within the viewport (view frustum).
Vertex data
shader
shader
Tessellation evaluation shader
Rasterization
Clipping
Primitive assembly
Geometry shader
Fragment shader
Screen buffer
Programmable stages
Fixed stages

Ope… eline



Generates fragments (pixels) to be drawn for each primitive. Interpolates vertex attributes.
scanlines
Verte data
ellation rol shader
Tessellation evaluation shader
Rasterization
Clipping
Primitive assembly
Geometry shader
Fragment shader
Screen buffer
Programmable stages
Fixed stages

OpenGL rendering pipeline



GLSL - fundamentals

Shaders

- Shaders are small programs executed on a GPU
  - Executed for each vertex, each pixel (fragment), etc.
- They are written in GLSL (OpenGL Shading Language)
  - Similar to C++ and Java
  - Primitive (int, float) and aggregate data types (ivec3, vec3)
  - Structures and arrays
  - Arithmetic operations on scalars, vectors and matrices
  - Flow control: if, switch, for, while
  - Functions

Example of a vertex shader

```
#version 330
in vec3 position;              // vertex position in local space
in vec3 normal;                // vertex normal in local space
out vec3 frag_normal;          // fragment normal in world space
uniform mat4 mvp_matrix;       // model-view-projection matrix

void main()
{
    // Typicaly normal is transformed by the model matrix
    // Since the model matrix is identity in our case, we do not modify normals
    frag_normal = normal;

    // The position is projected to the screen coordinates using mvp_matrix
    gl_Position = mvp_matrix * vec4(position, 1.0);
}
```

Why is this piece of code needed?

Data types

- Basic types
  - float, double, int, uint, bool
- Aggregate types
  - float: vec2, vec3, vec4; mat2, mat3, mat4
  - double: dvec2, dvec3, dvec4; dmat2, dmat3, dmat4
  - int: ivec2, ivec3, ivec4
  - uint: uvec2, uvec3, uvec4
  - bool: bvec2, bvec3, bvec4

```
vec3 V = vec3( 1.0, 2.0, 3.0 );    mat3 M = mat3( 1.0, 2.0, 3.0,
                                                  4.0, 5.0, 6.0,
                                                  7.0, 8.0, 9.0 );
```

Indexing components in aggregate types

- Subscripts: rgba, xyzw, stpq (work exactly the same)
  - float red = color.r;
  - float v_y = velocity.y;

  but also
  - float red = color.x;
  - float v_y = velocity.g;
- With 0-base index:
  - float red = color[0];
  - float m22 = M[1][1]; // second row and column of matrix M

Swizzling

You can select the elements of the aggregate type:

- vec4 rgba_color(1.0, 1.0, 0.0, 1.0);
- vec3 rgb_color = rgba_color.rgb;
- vec3 bgr_color = rgba_color.bgr;
- vec3 luma = rgba_color.ggg;

Arrays

- Similar to C

```
float lut[5] = float[5]( 1.0, 1.42, 1.73, 2.0, 2.23 );
```

- Size can be checked with “length()”

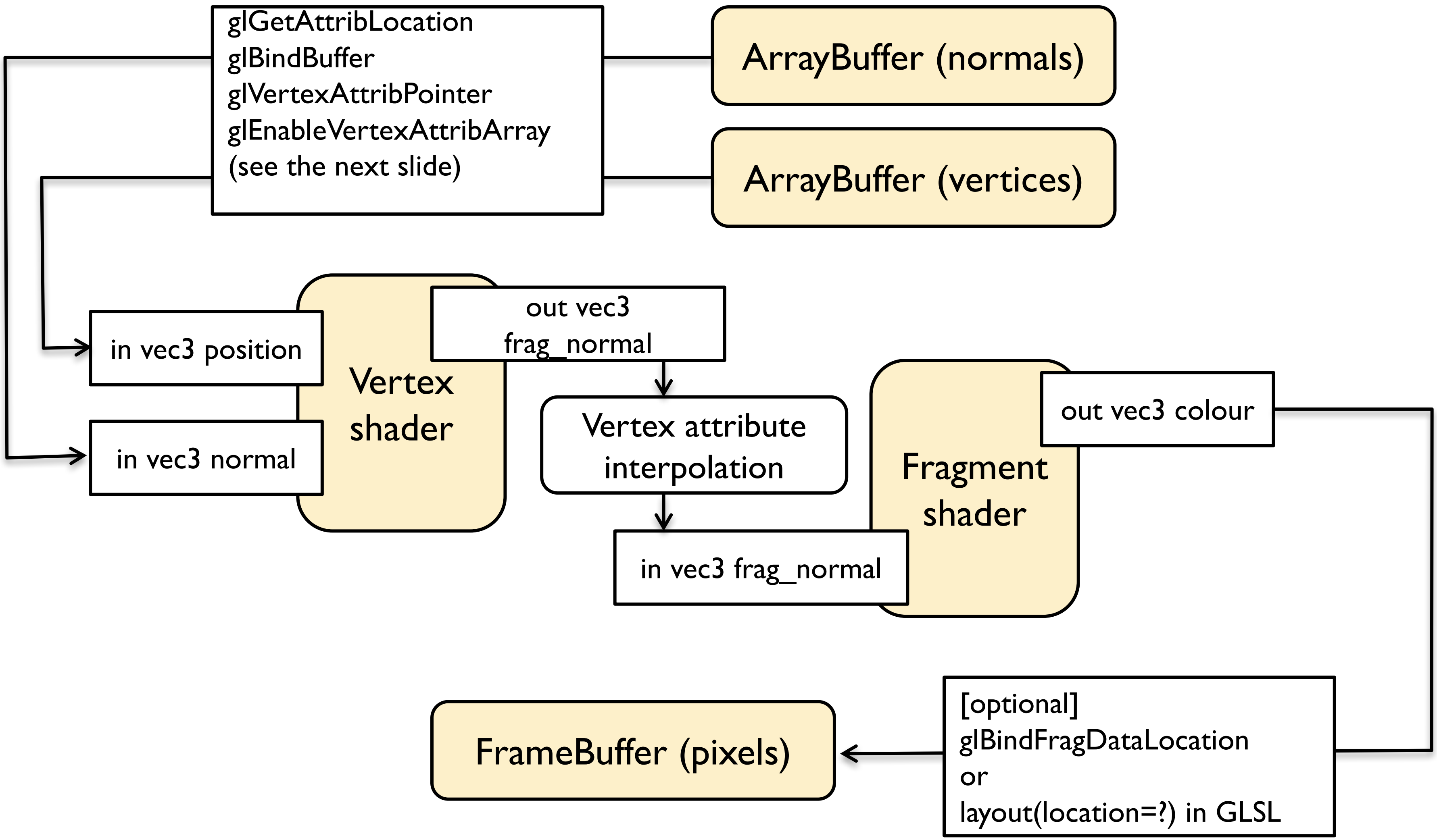
```
for( int i = 0; i < lut.length(); i++ ) {
   lut[i] *= 2;
}
```

Storage qualifiers

- `const` – read-only, fixed at compile time
- `in` – input to the shader
- `out` – output from the shader
- `uniform` – parameter passed from the application (Java), constant for the primitive
- `buffer` – shared with the application
- `shared` – shared with local work group (compute shaders only)

- Example: `const float pi=3.14;`

Shader inputs and outputs



How to specify input to a vertex shader?

```
// Get the locations of the "position" vertex attribute variable in our shader
int position_loc = glGetAttribLocation(shaders_handle, "position");
// If the vertex attribute found
if (position_loc != -1) {
  // Activate the ArrayBuffer that should be accessed in the shader
  glBindBuffer(GL_ARRAY_BUFFER, vertex_handle);
  // Specifies where the data for "position" variable can be accessed
  glVertexAttribPointer(position_loc, 3, GL_FLOAT, false, 0, 0);
  // Enable that vertex attribute variable
  glEnableVertexAttribArray(position_loc);
}
```

Passing uniform(s) to a shader

- In shader:

```
uniform mat4 mvp_matrix; // model-view-projection matrix
```

- In Java:

```
Matrix4f mvp_matrix; // Matrix to be passed to the shader
...
int mvp_location = glGetUniformLocation(shaders.getHandle(),
                   "mvp_matrix");
FloatBuffer mvp_buffer = BufferUtils.createFloatBuffer(16);
mvp_matrix.get(mvp_buffer);
glUniformMatrix4fv(mvp_location, false, mvp_buffer);
```

Name of the method depends on the data type.
For example, `glUniform3fv` for `Vector3f`

GLSL Operators

- Arithmetic: + - ++ --
  - Multiplication:
    - `vec3 * vec3` – element-wise
    - `mat4 * vec4` – matrix multiplication (with a column vector)
- Bitwise (integer): `<<, >>, &, |, ^`
- Logical (bool): `&&, ||, ^^`
- Assignment:

```
float a=0;
a += 2.0; // Equivalent to a = a + 2.0
```

- See the quick reference guide at: https://www.opengl.org/documentation/glsl/

GLSL Math

- Trigonometric:
  - `radians( deg ), degrees( rad ), sin, cos, tan, asin, acos, atan, sinh, cosh, tanh, asinh, acosh, atanh`
- Exponential:
  - `pow, exp, log, exp2, log2, sqrt, inversesqrt`
- Common functions:
  - `abs, round, floor, ceil, min, max, clamp,` …
- And many more

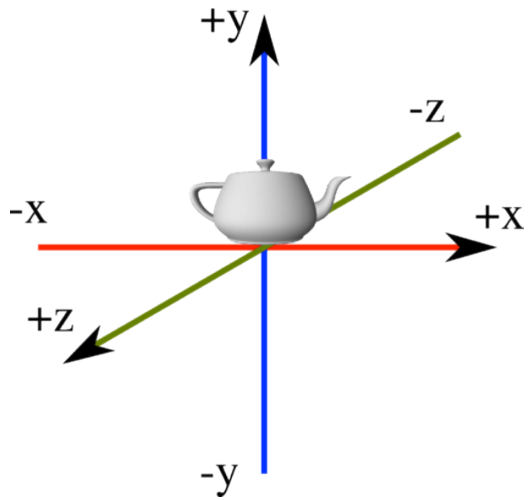
- See the quick reference guide at: https://www.opengl.org/documentation/glsl/

GLSL flow control

```
if( bool ) {
  // true
} else {
  // false
}

switch( int_value ) {
  case n:
    // statements
    break;
  case m:
    // statements
    break;
  default:
}
for( int i = 0; i<10; i++ ) {
   ...
}

while( n < 10 ) {
 ...
}

do {
 ...
} while ( n < 10 )
```

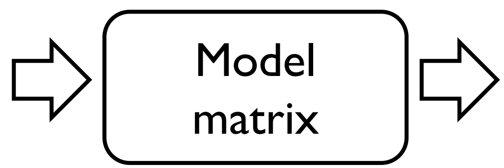
Transformations (Vertex shaders)

Model, View, Projection matrices

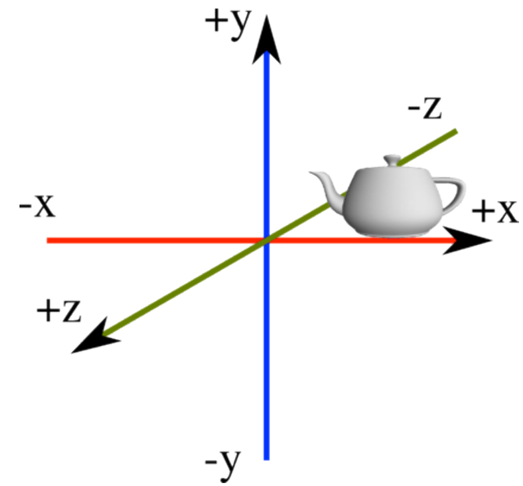


Object coordinates

Object centred at the origin

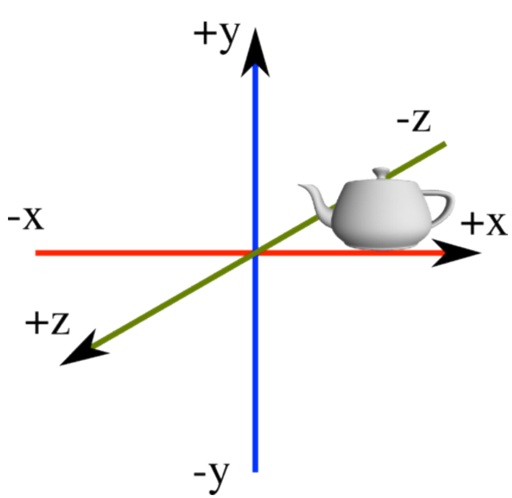


To position each object in the scene. Could be different for each object.



World coordinates

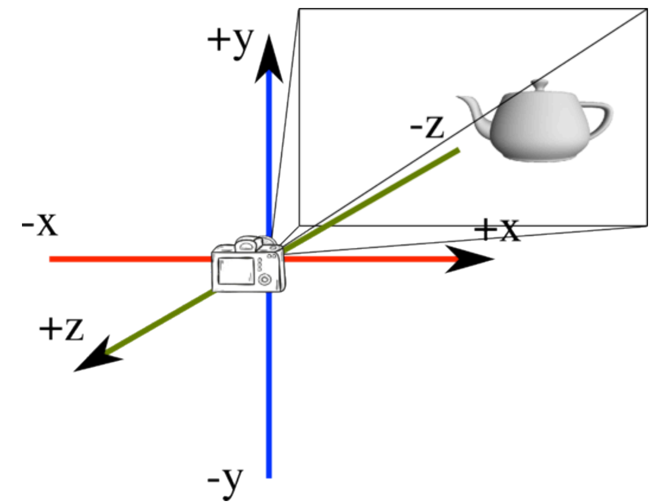
Model, View, Projection matrices



World coordinates

View matrix

To position all objects relative to the camera

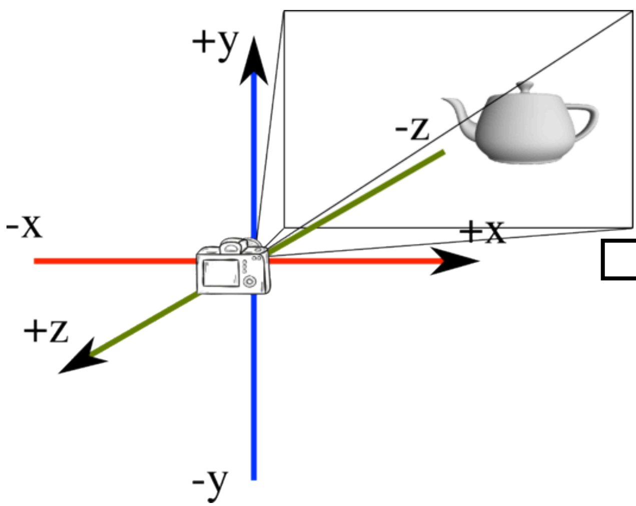


View (camera) coordinates

Camera at the origin, pointing at -z

Model, View, Projection matrices

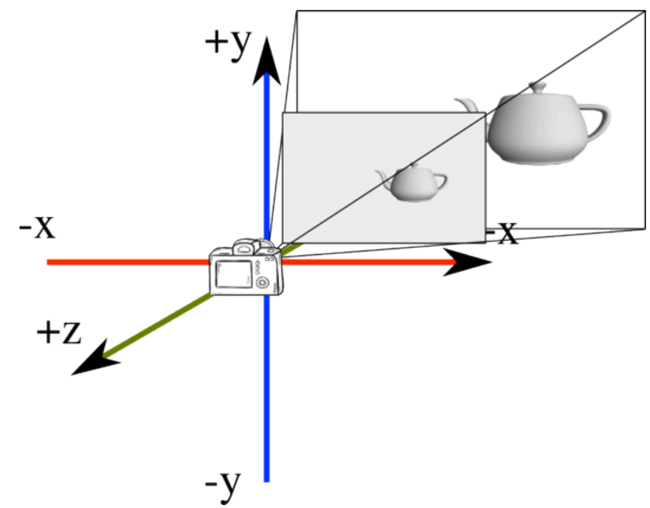
The default OpenGL coordinate system is right-handed



View (camera) coordinates

Projection matrix

To project 3D coordinates to a 2D plane. Note that *z* coordinate is retained for depth testing.



Screen coordinates

x and y must be in the range -1 and 1

All together

Screen coordinates

x_s/w_s and y_s/w_s must be between -1 and 1

$$\begin{bmatrix} x_s \\ y_s \\ z_s \\ w_s \end{bmatrix} = P \cdot V \cdot M \cdot \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

3D world vertex coordinates

Projection, view and model matrices

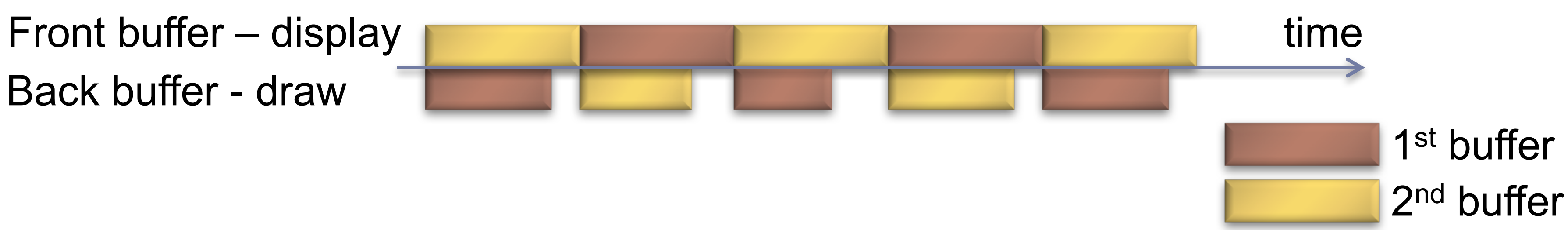
Raster buffers (colour, depth, stencil)

Render buffers in OpenGL

Colour:	GL_FRONT	GL_BACK	Four components: RGBA Typically 8 bits per component
In stereo:	GL_FRONT_LEFT	GL_FRONT_RIGHT	
	GL_BACK_LEFT	GL_BACK_RIGHT	
Depth:	DEPTH	To resolve occlusions (see Z-buffer algorithm) Single component, usually >8 bits	
Stencil:	STENCIL	To block rendering selected pixels Single component, usually 8 bits.	

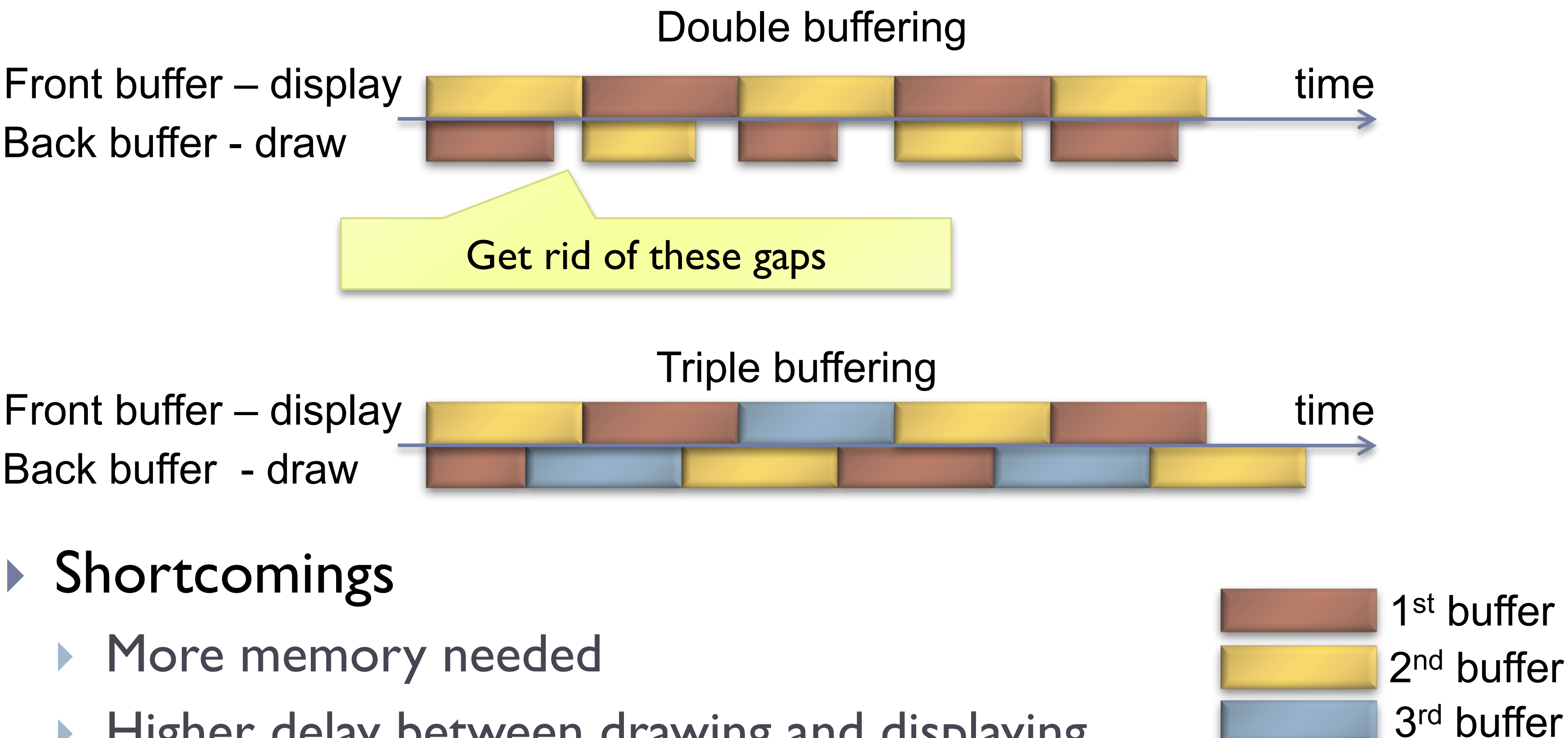
Double buffering

- To avoid flicker, tearing
- Use two buffers (rasters):
  - Front buffer – what is shown on the screen
  - Back buffer – not shown, GPU draws into that buffer
- When drawing is finished, swap front- and back-buffers



Triple buffering

- Do not wait for swapping to start drawing the next frame

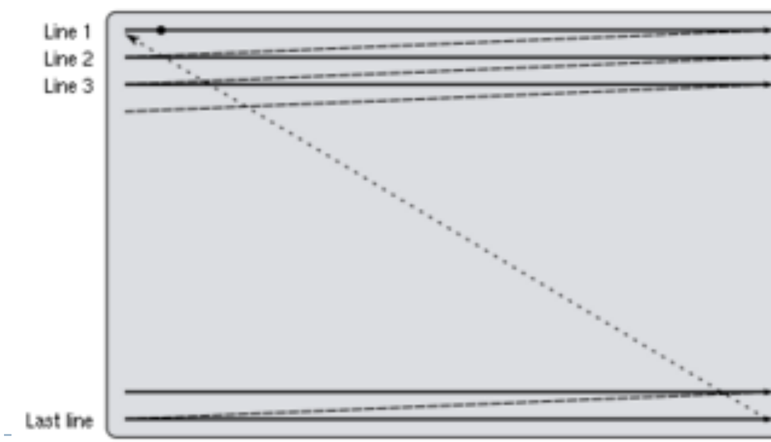


- Shortcomings
  - More memory needed
  - Higher delay between drawing and displaying a frame

Vertical Synchronization: V-Sync

- Pixels are copied from colour buffer to monitor raw-by-raw
- If front & back buffer are swapped during this process:
  - Upper part of the screen contains previous frame
  - Lower part of the screen contains current frame
  - Result: tearing artefact
- Solution: When V-Sync is enabled
  - `glwfSwapInterval(1);`

`glSwapBuffers()` waits until the last raw is copied to the display.



Z-Buffer - algorithm

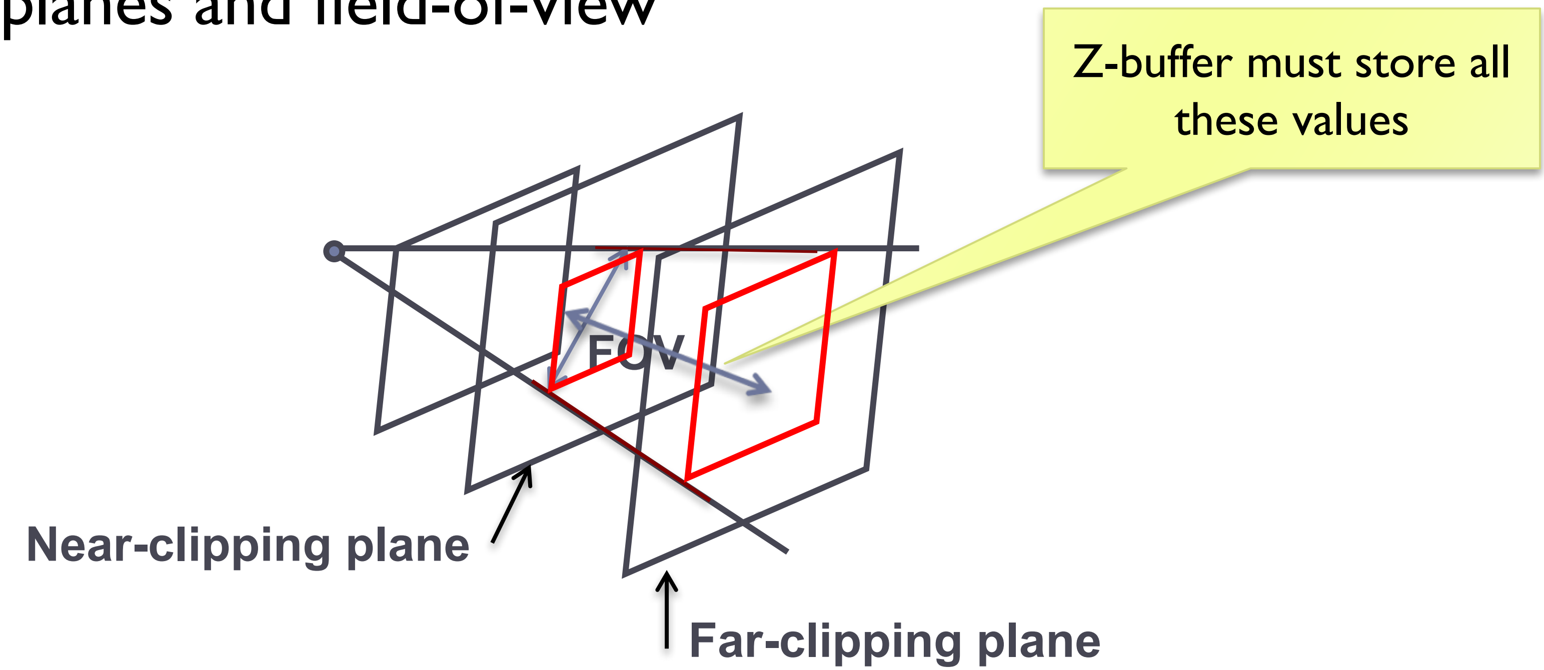
- Initialize the depth buffer and image buffer for all pixels

  color(x, y) = Background_Color,

  depth(x, y) = z_far // position of the far clipping plane
- For every triangle in a scene
  - For every fragment (x, y) representing this triangle
    - Calculate z for current (x, y)
    - `if (z < depth(x, y))`
      - `depth (x, y) = z`
      - `color (x, y) = Polygon_Color (x, y)`

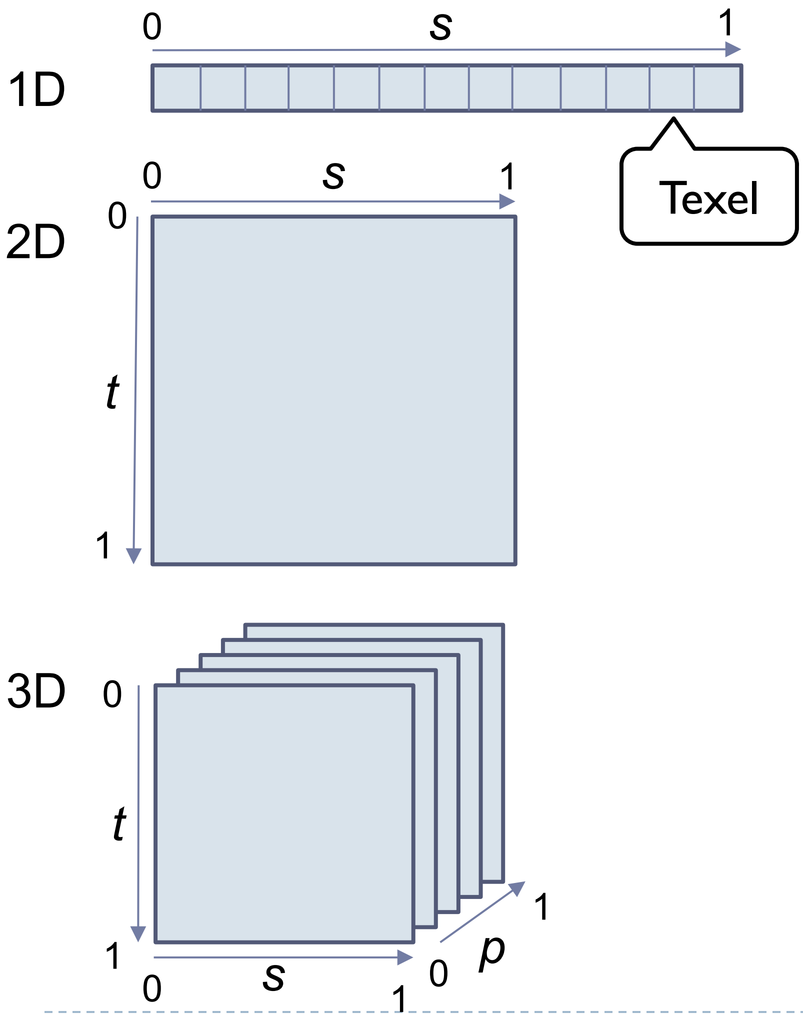
View frustum

- Controlled by camera parameters: near-, far-clipping planes and field-of-view



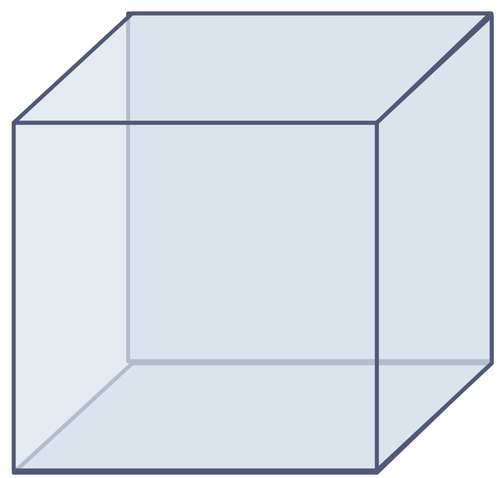
Textures

(Most important) OpenGL texture types

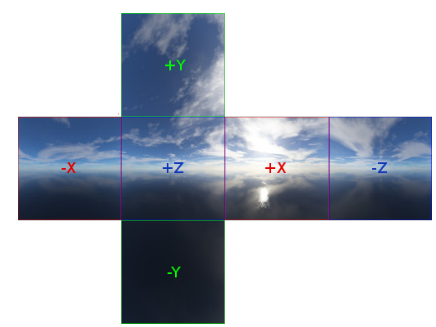


Texture can have any size but the sizes that are powers of two (POT, 2^n) may give better performance.

CUBE_MAP

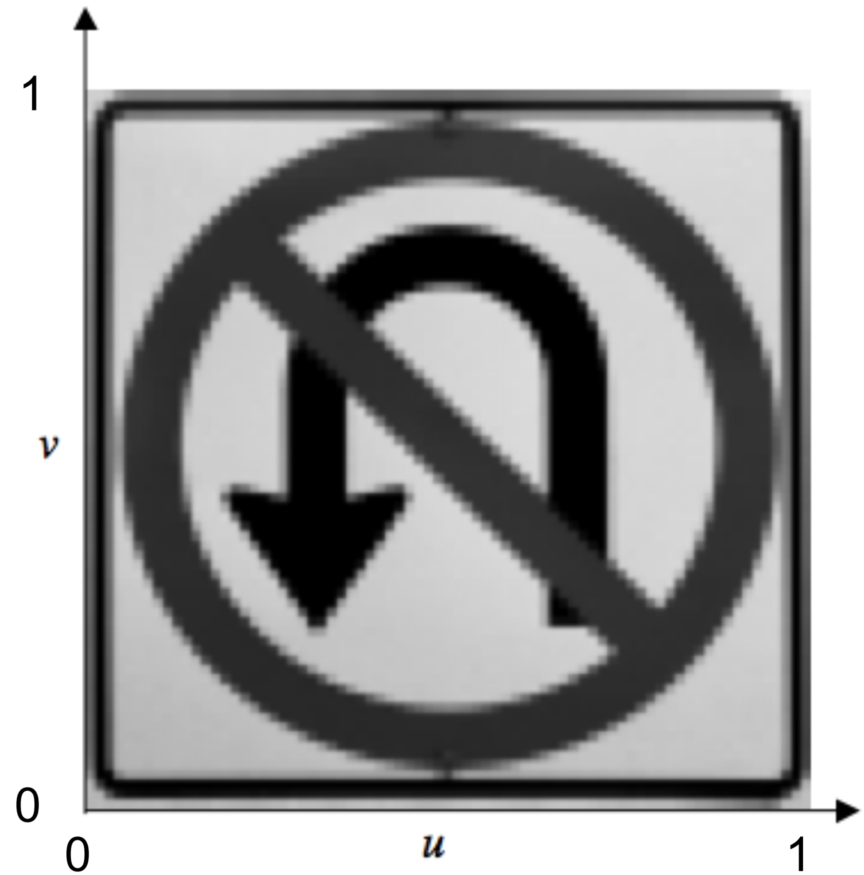


Used for environment mapping



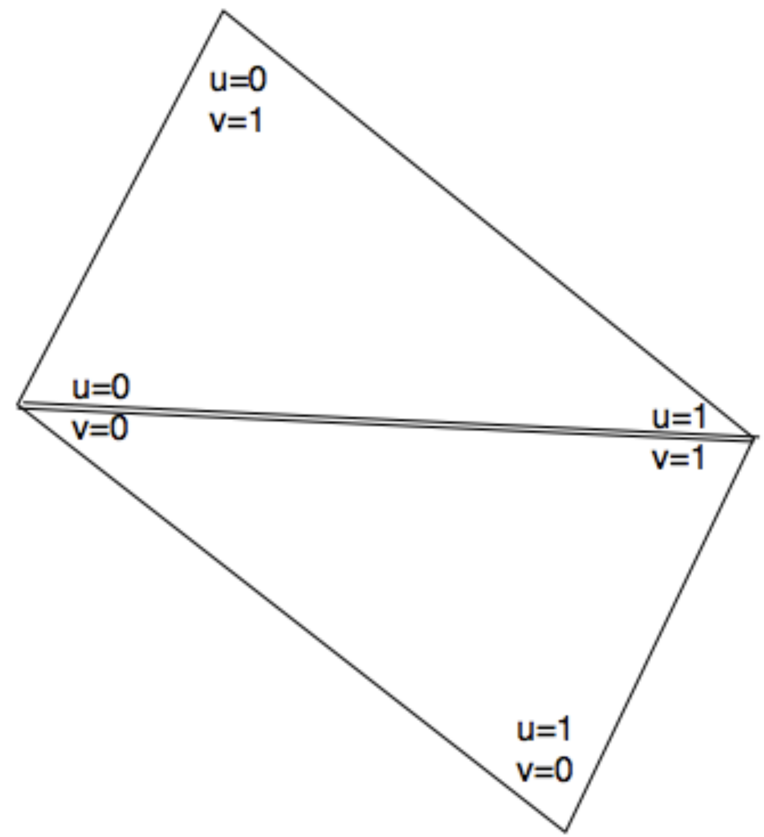
Texture mapping

- 1. Define your texture function (image) T(u,v)
- (u,v) are texture coordinates



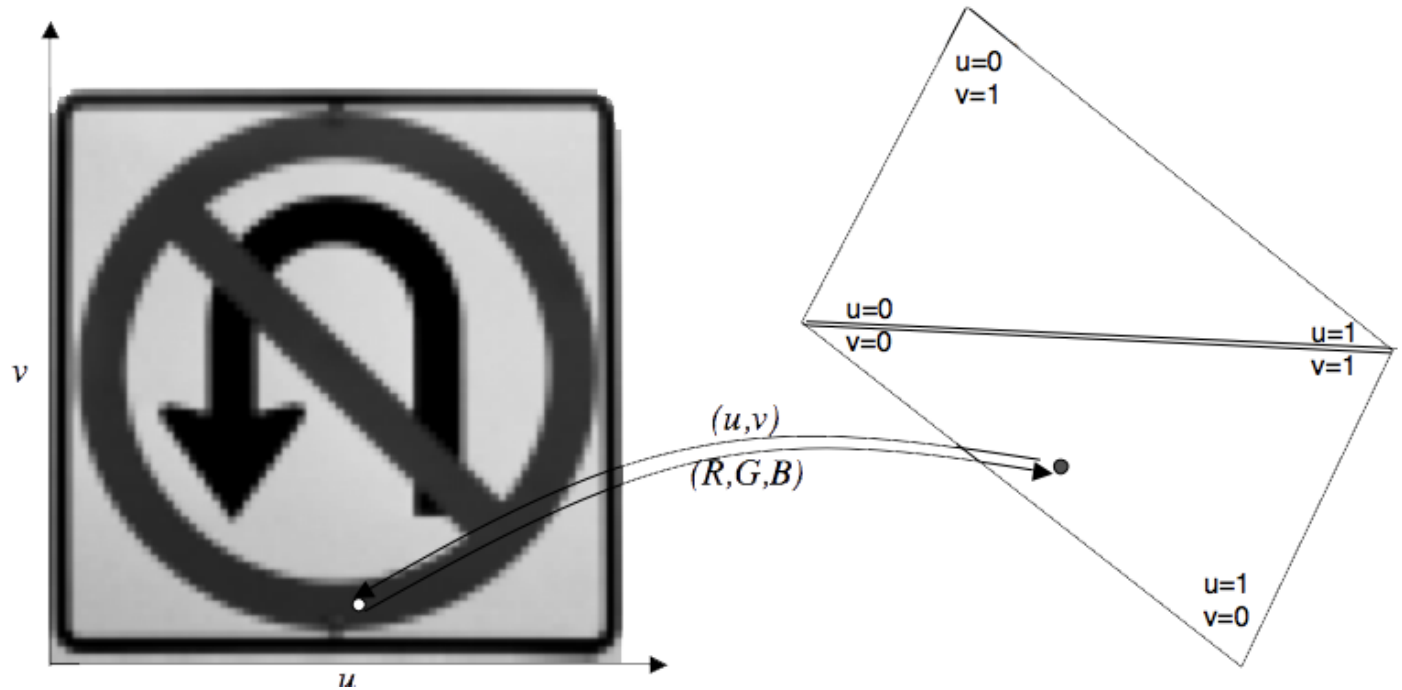
Texture mapping

- 2. Define the correspondence between the vertices on the 3D object and the texture coordinates

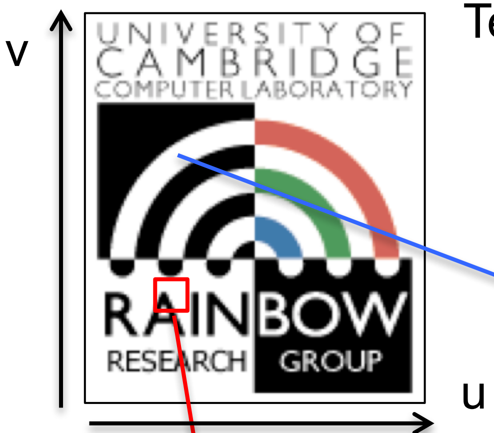


Texture mapping

- 3. When rendering, for every surface point compute texture coordinates. Use the texture function to get texture value. Use as color or reflectance.

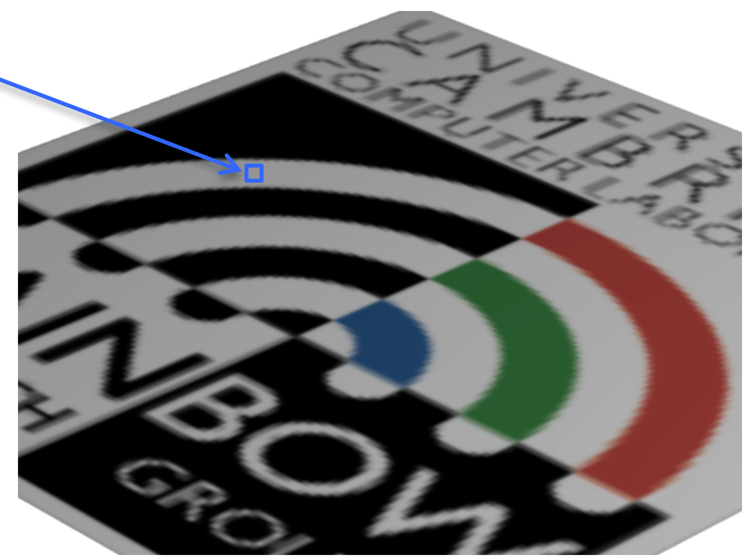


Sampling



Texture

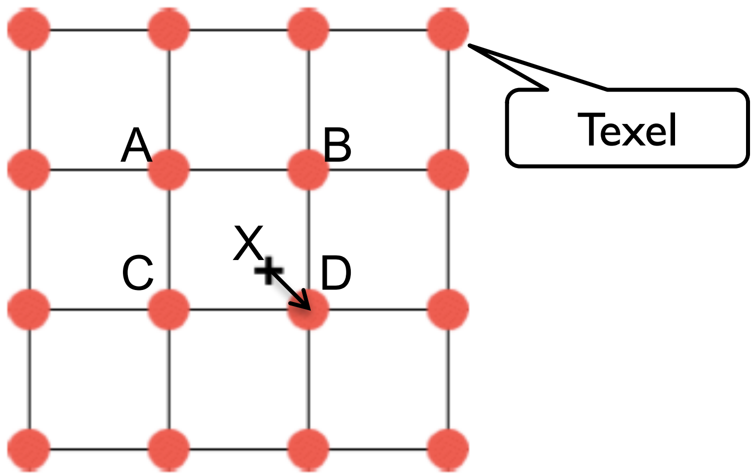
Up-sampling
More pixels than texels
Values need to be interpolated



Down-sampling
Fewer pixels than texels
Values need to be averaged over an area of the texture (usually using a mipmap)

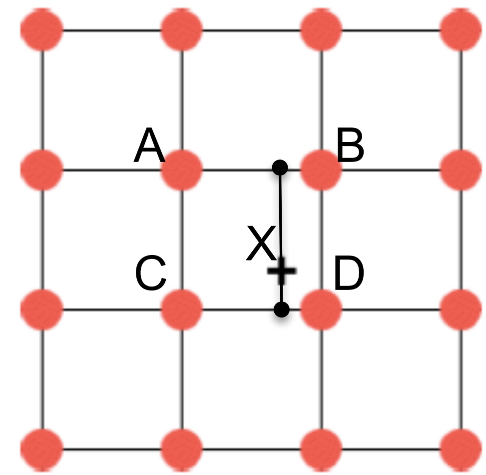
Nearest neighbor vs. bilinear interpolatim

Nearest neighbour

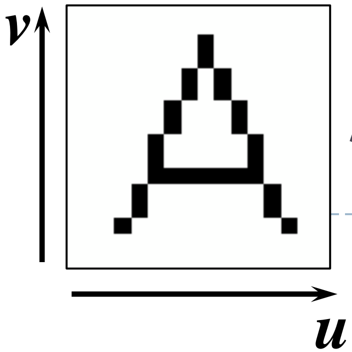


Pick the nearest texel: D

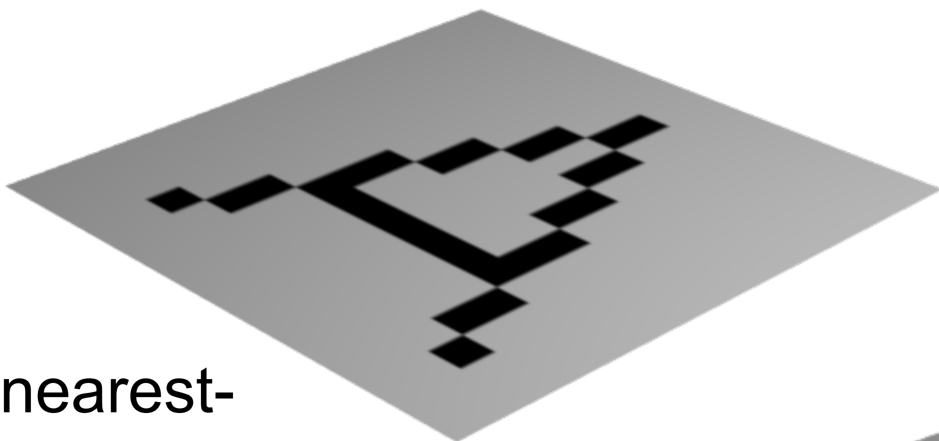
Bilinear interpolation



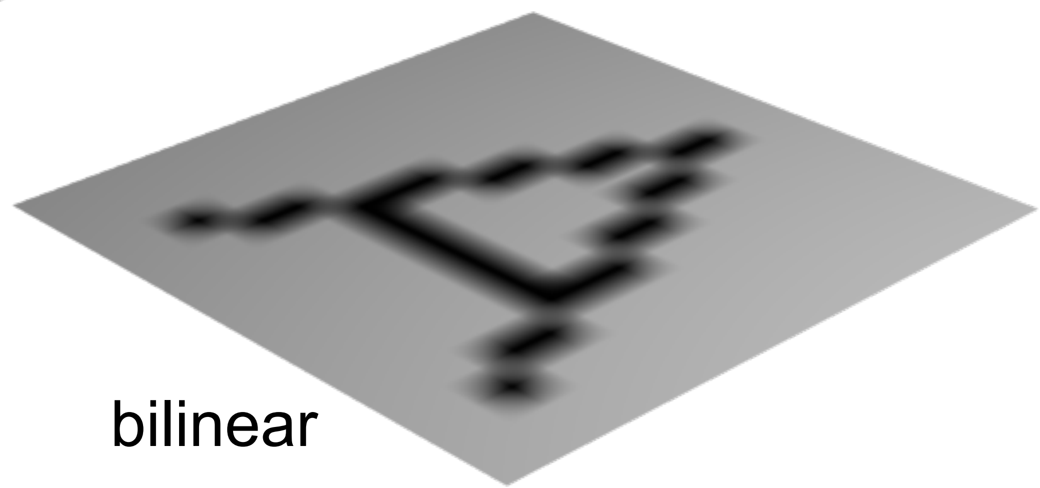
Interpolate first along x-axis between AB and CD, then along y-axis between the interpolated points.



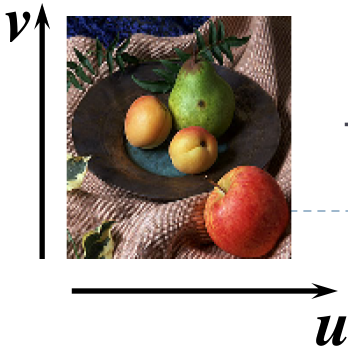
Texture mapping examples



nearest-neighbour



bilinear



Up-sampling

nearest-neighbour

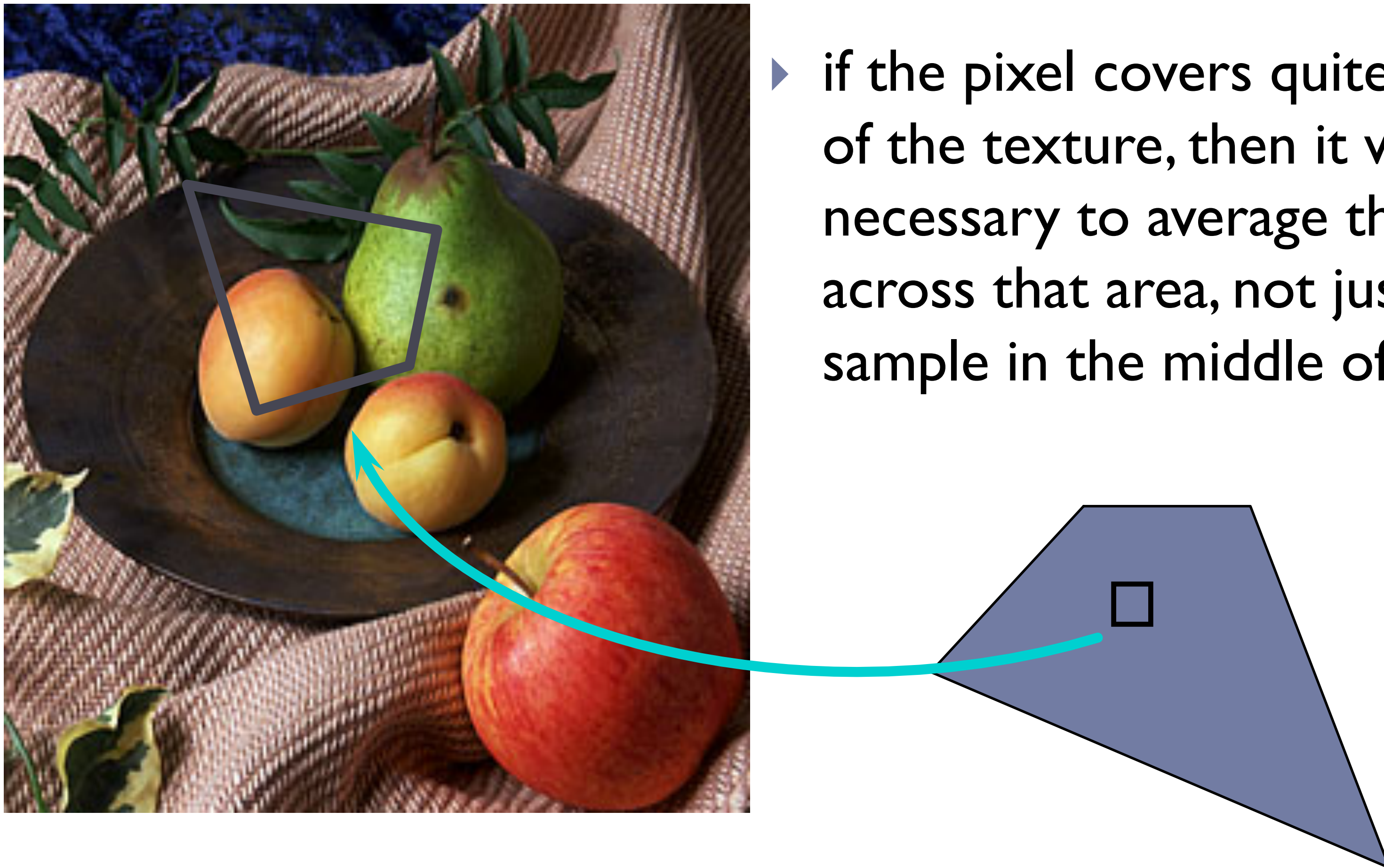
blocky artefacts

bilinear

blurry artefacts

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

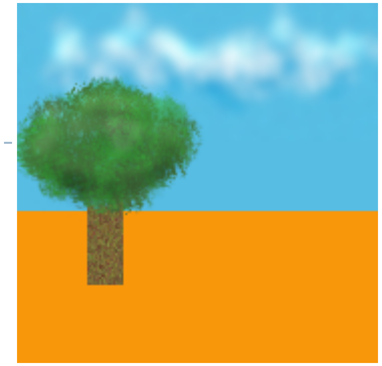
Down-sampling

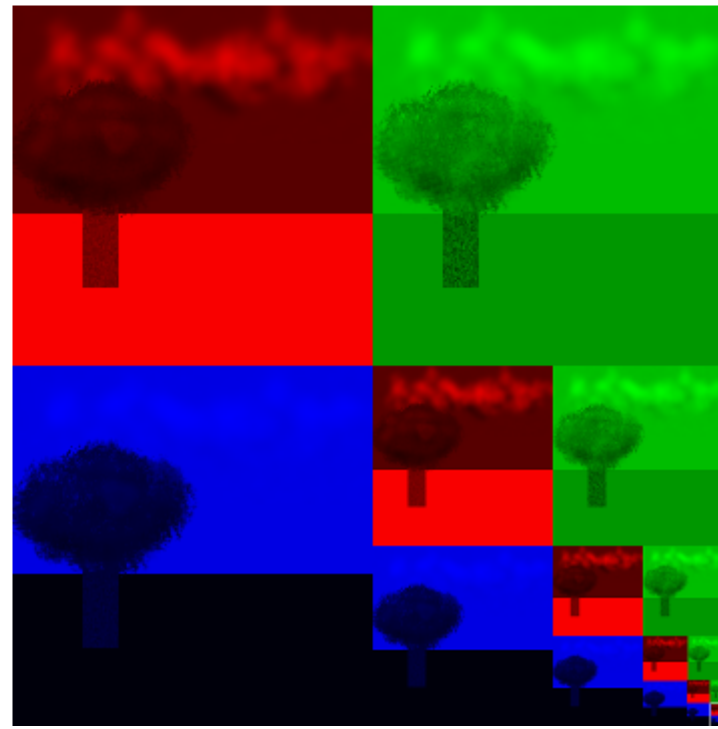


- if the pixel covers quite a large area of the texture, then it will be necessary to average the texture across that area, not just take a sample in the middle of the area

Mipmap

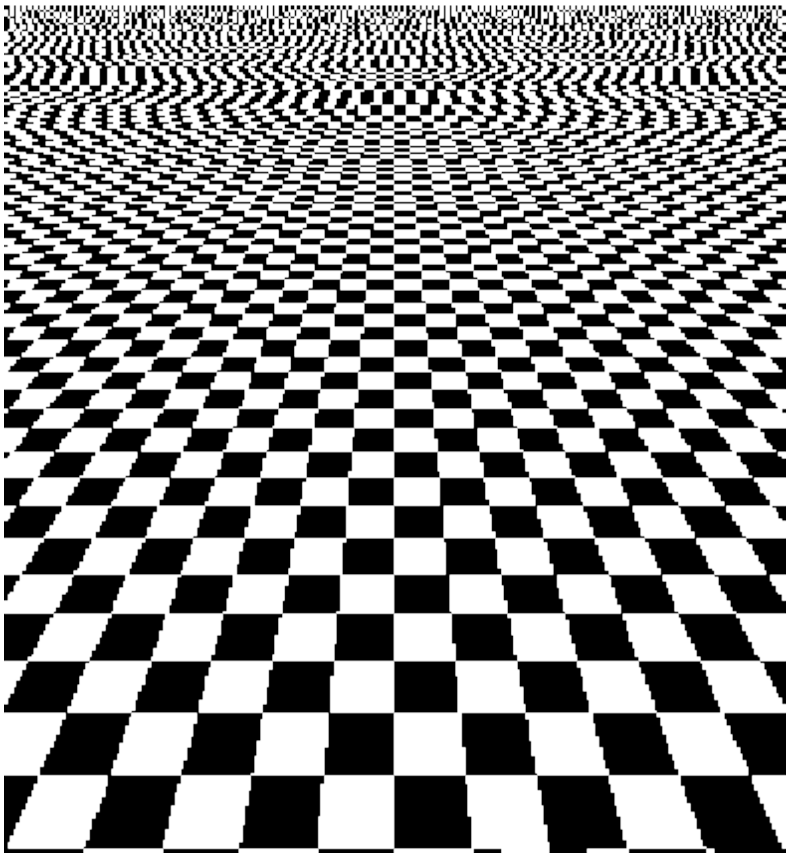
- Textures are often stored at multiple resolutions as a mipmap
  - Each level of the pyramid is half the size of the lower level
- It provides pre-filtered texture (area-averaged) when screen pixels are larger than the full resulution texels
- Mipmap requires just 1/3 of the original texture size to store



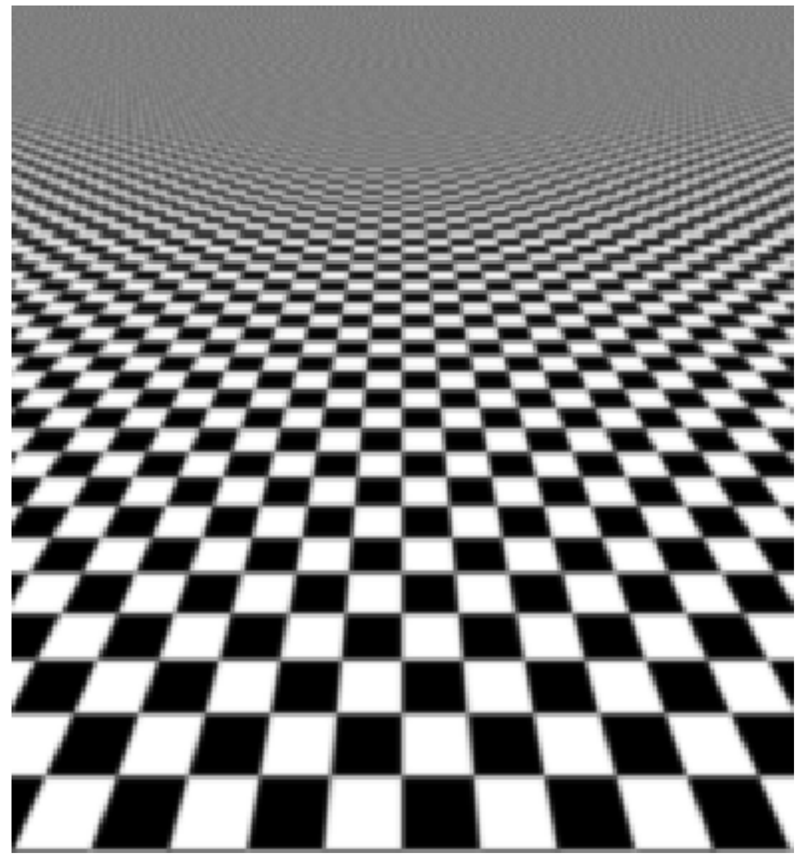


Down-sampling

without area averaging

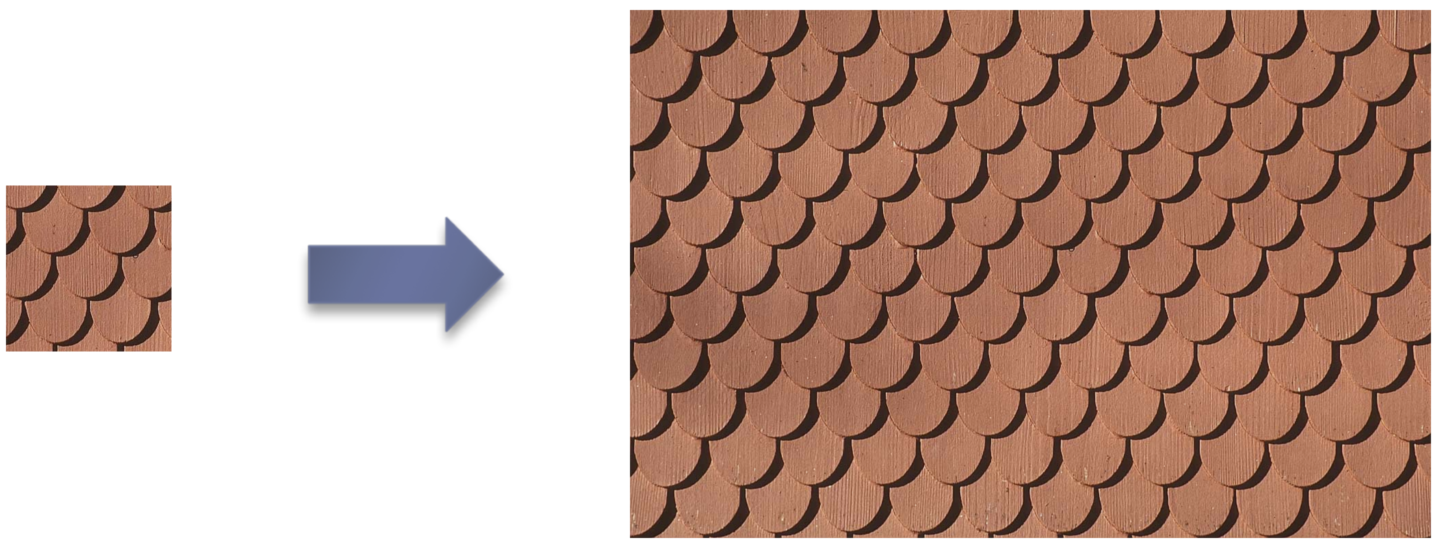


with area averaging



Texture tiling

- Repetitive patterns can be represented as texture tiles.
- The texture folds over, so that
  - T(u=1.1, v=0) = T(u=0.1, v=0)



Texture atlas

- A single texture is often used for multiple surfaces and objects

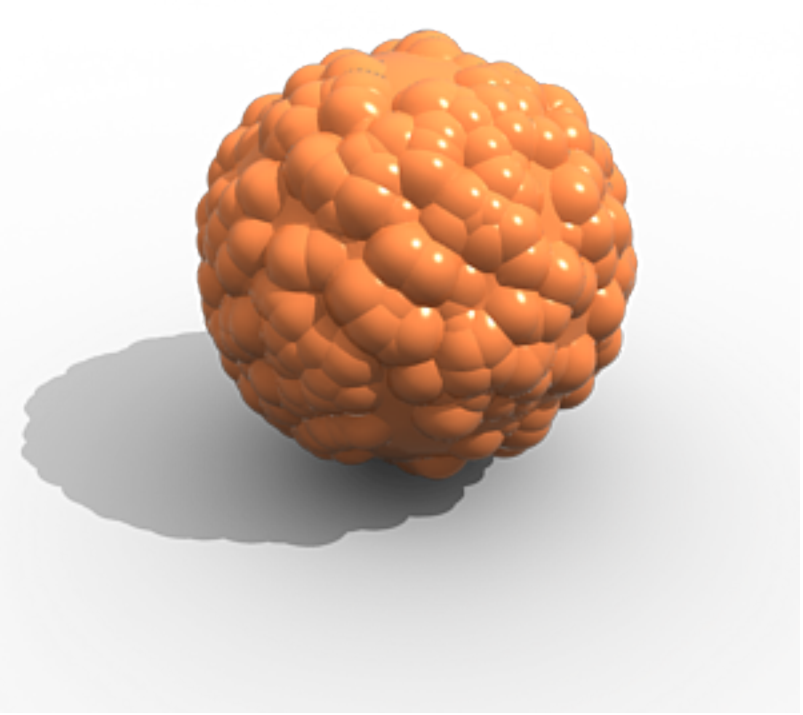
Example from: http://awshub.com/blog/blog/2011/11/01/hi-poly-vs-low-poly/

Bump (normal) mapping

- Special kind of texture that modifies surface normal
  - Surface normal is a vector that is perpendicular to a surface
- The surface is still flat but shading appears as on an uneven surface
- Easily done in fragment shaders

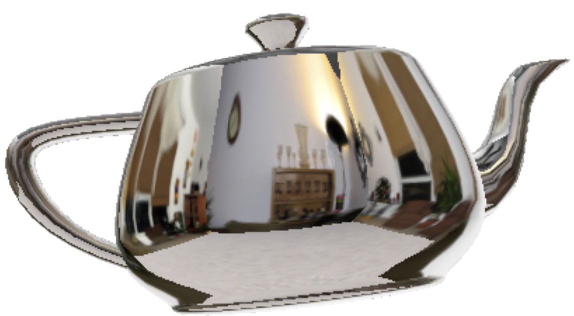
Displacement mapping

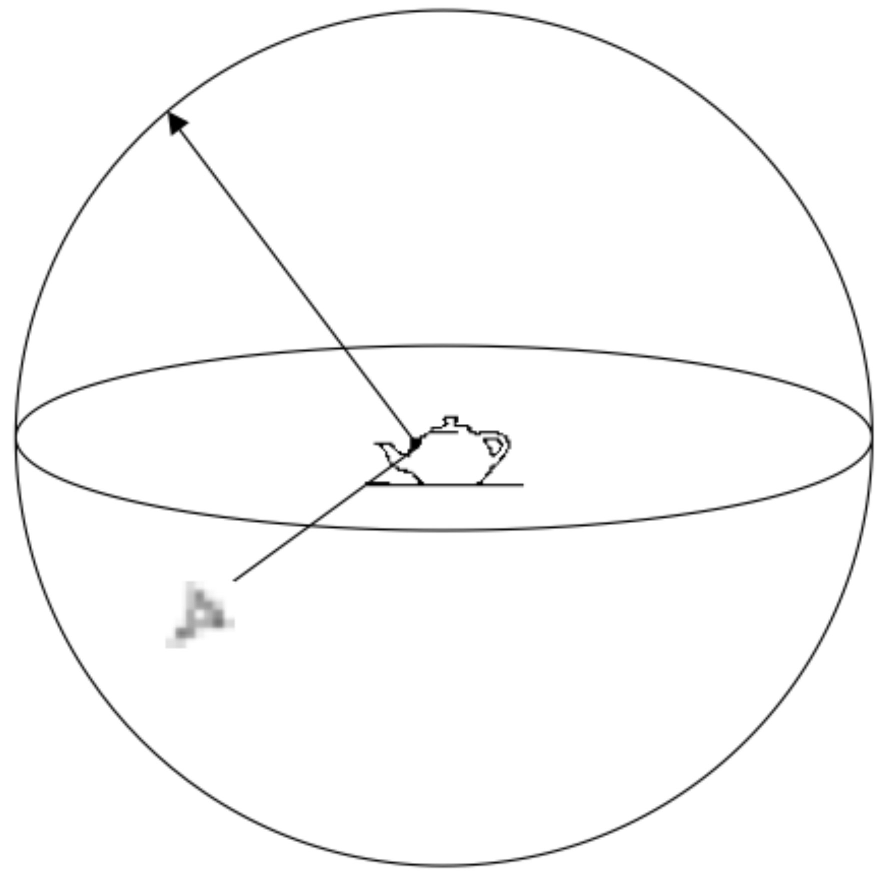
- Texture that modifies surface
- Better results than bump mapping since the surface is not flat
- Requires geometry shaders



Environment mapping

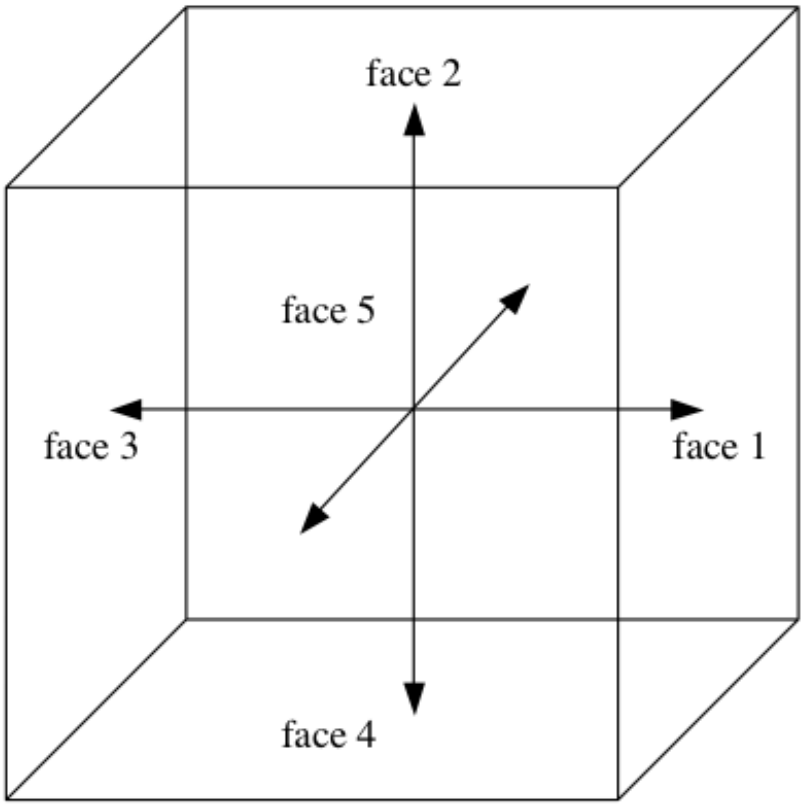
- To show environment reflected by an object



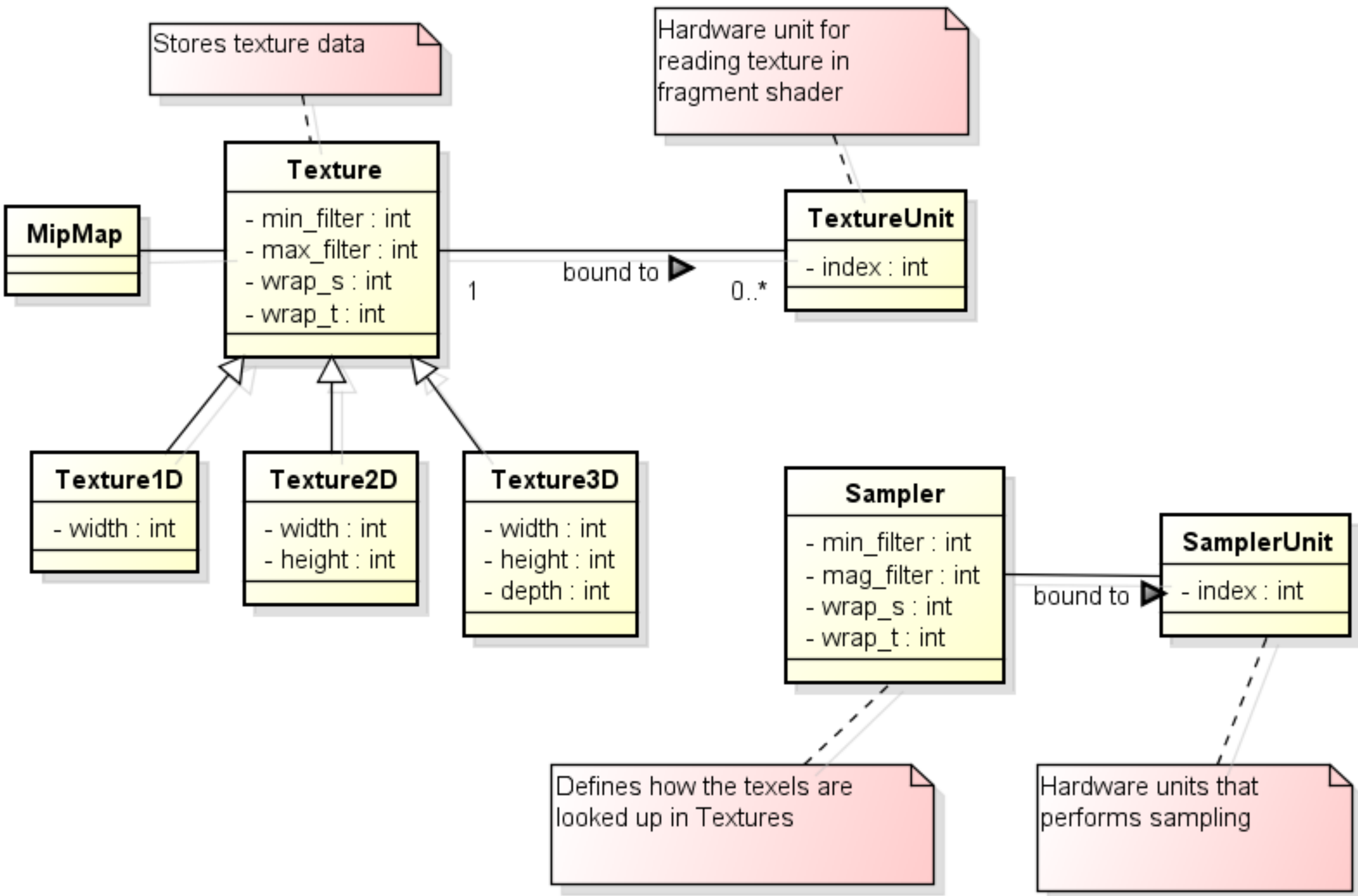


Environment mapping

- Environment cube
- Each face captures environment in that direction



Texture objects in OpenGL



Setting up a texture

```
// Create a new texture object in memory and bind it
int texId = glGenTextures();
glActiveTexture(textureUnit);
glBindTexture(GL_TEXTURE_2D, texId);

// All RGB bytes are aligned to each other and each component is 1 byte
glPixelStorei(GL_UNPACK_ALIGNMENT, 1);

// Upload the texture data and generate mipmaps
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, tWidth, tHeight, 0,
        GL_RGBA, GL_UNSIGNED_BYTE, buf);
glGenerateMipmap(GL_TEXTURE_2D);
```

Texture parameters

```
//Setup filtering, i.e. how OpenGL will interpolate the pixels when scaling up or down
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_NEAREST);
```

How to interpolate in 2D

How to interpolate between mipmap levels

```
//Setup wrap mode, i.e. how OpenGL will handle pixels outside of the expected range
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
```

Fragment shader

```
#version 330
uniform sampler2D texture_diffuse;
in vec2 frag_TextureCoord;

out vec4 out_Color;

void main(void) {
	out_Color = texture(texture_diffuse, frag_TextureCoord);
}
```

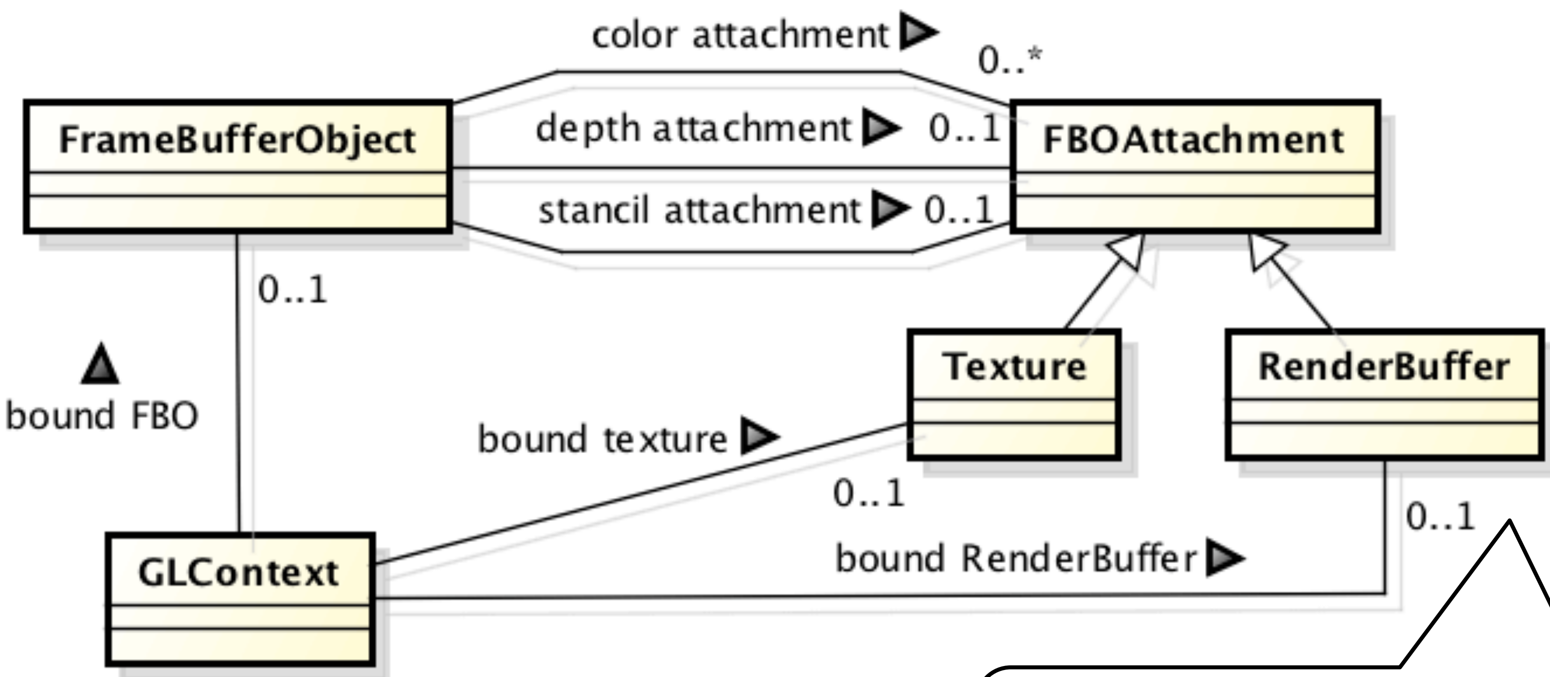
Rendering

```
// Bind the texture
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, texId);

glBindVertexArray(vao);
glDrawElements(GL_TRIANGLES, indicesCount, GL_UNSIGNED_INT, 0);
glBindVertexArray(0);

glBindTexture(GL_TEXTURE_2D, 0);
```

Frame Buffer Objects

Frame Buffer Objects (FBOs)

- Instead of rendering to the screen buffer (usually GL_BACK), an image can be rendered to an off-screen buffer: a Texture or a RenderBuffer



Faster to render to than a texture but cannot be sampled, pixels can be only copied.

Frame Buffer Object applications

- Post-processing, tone-mapping, blooming, etc.

- Reflections (in water), animated textures (e.g. TV screen)

- When the result of rendering is not shown (e.g. saved to disk)

FBO: Code example 1/3

- Create FBO, attach a Texture (colour) and a RenderBuffer (depth)

```
int color_tex = glGenTextures();
glBindTexture(GL_TEXTURE_2D, color_tex);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, 256, 256, 0, GL_BGRA,
GL_UNSIGNED_BYTE, NULL);

int myFBO = glGenFramebuffers();
glBindFramebuffer(GL_FRAMEBUFFER, myFBO);
//Attach 2D texture to this FBO
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
GL_TEXTURE_2D, color_tex, 0);
```

FBO: Code example 2/3

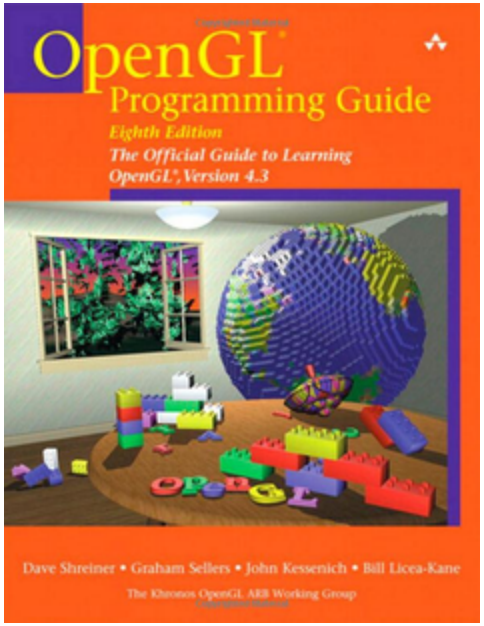
```
int depth_rb = glGenRenderbuffers();
glBindRenderbuffer(GL_RENDERBUFFER, depth_rb);
glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT24,
256, 256);
//Attach depth buffer to FBO
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
GL_RENDERBUFFER, depth_rb);
```

FBO: Code example 3/3

- Render

```
glBindFramebuffer(GL_FRAMEBUFFER, myFBO);
glClearColor(0.0, 0.0, 0.0, 0.0);
glClearDepth(1.0f);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

// Render

glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

References



- The OpenGL Programming Guide, 8th Edition, The Official Guide to Learning OpenGL by Dave Shreiner et al (2013) ISBN-10: 0321773039

- OpenGL quick reference guide https://www.opengl.org/documentation/glsl/

- Google search: „man gl.....”

Introduction to Computer Graphics

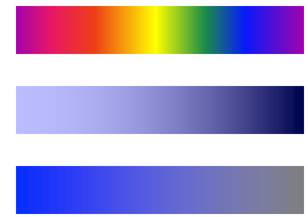
- Background
- Rendering
- Graphics pipeline
- Graphics hardware and modern OpenGL
- **Technology**
  - Colour spaces
  - Brief overview of display and printer technologies

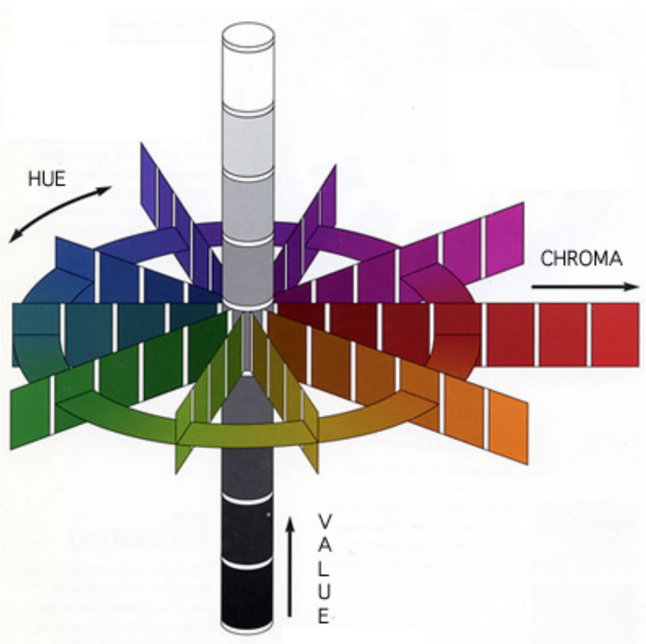
Representing colour

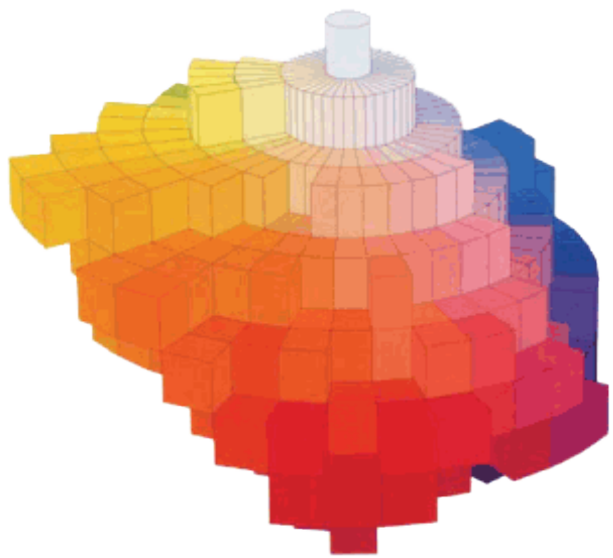
- we need a mechanism which allows us to represent colour in the computer by some set of numbers
  - preferably a small set of numbers which can be quantised to a fairly small number of bits each
- we will discuss:
  - Munsell's *artists'* scheme
    - which classifies colours on a perceptual basis
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

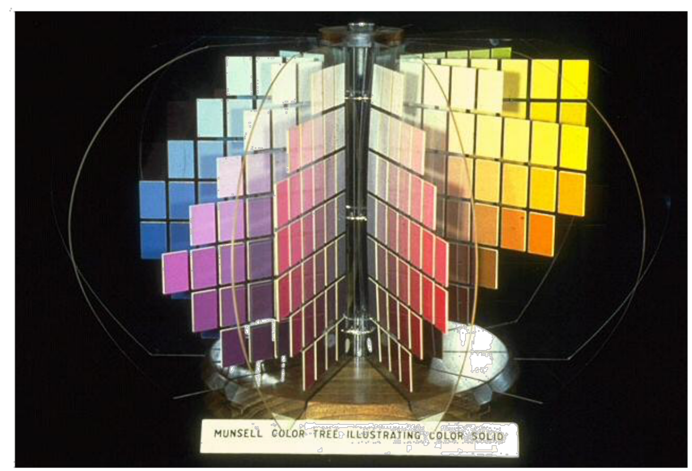
Munsell's colour classification system

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



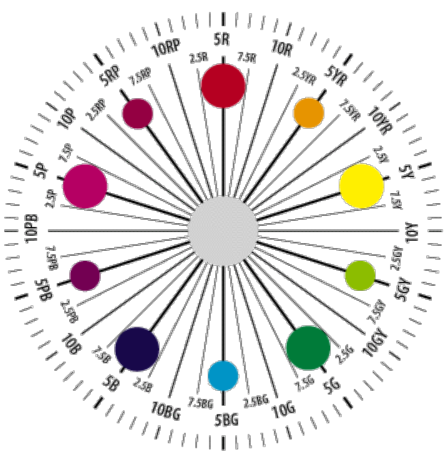


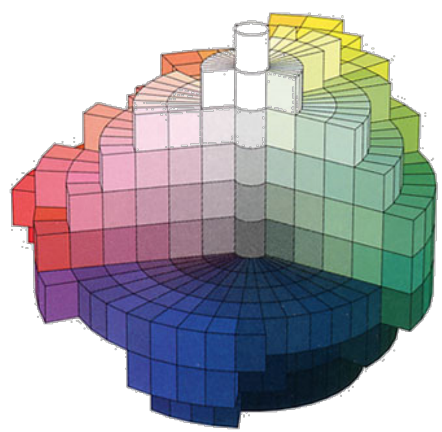


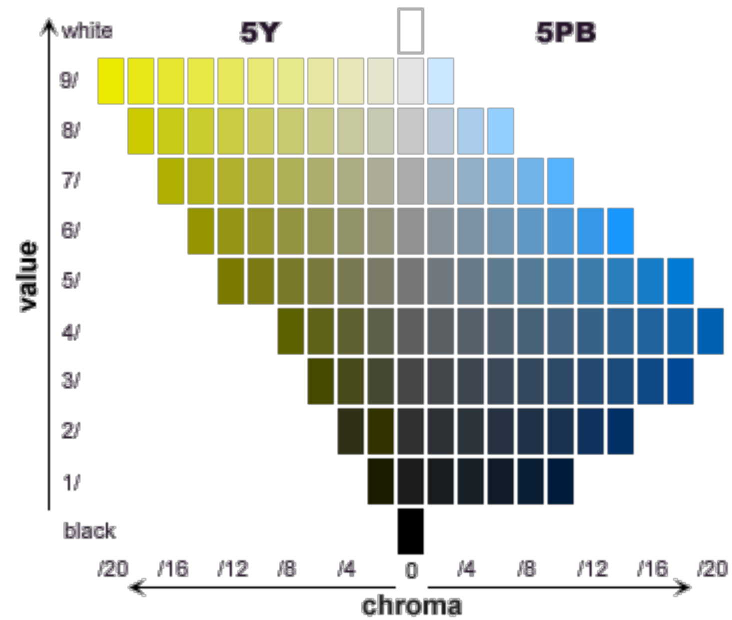


Munsell's colour classification system

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





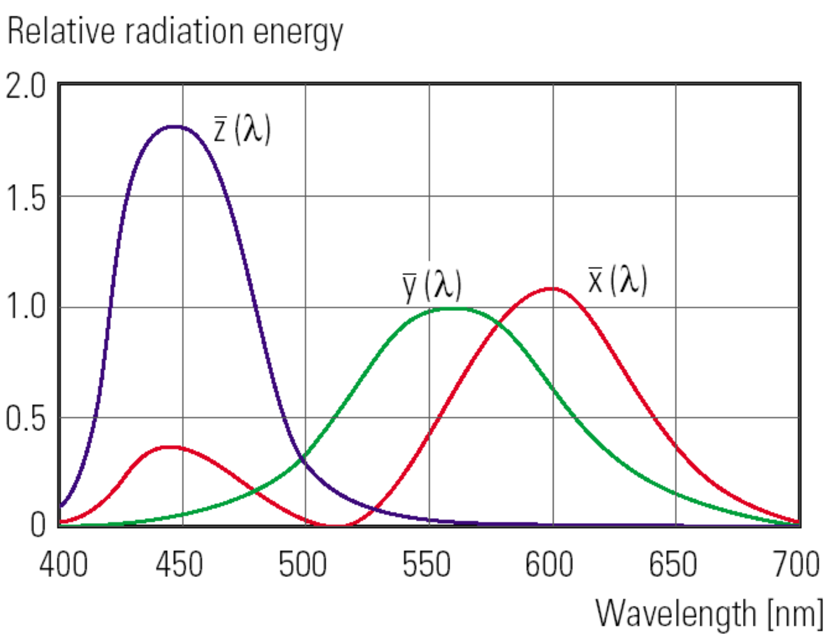


invented by Albert H. Munsell, an American artist, in 1905 in an attempt to systematically classify colours

FvDFH Sec 13.2.2

XYZ colour space

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



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

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

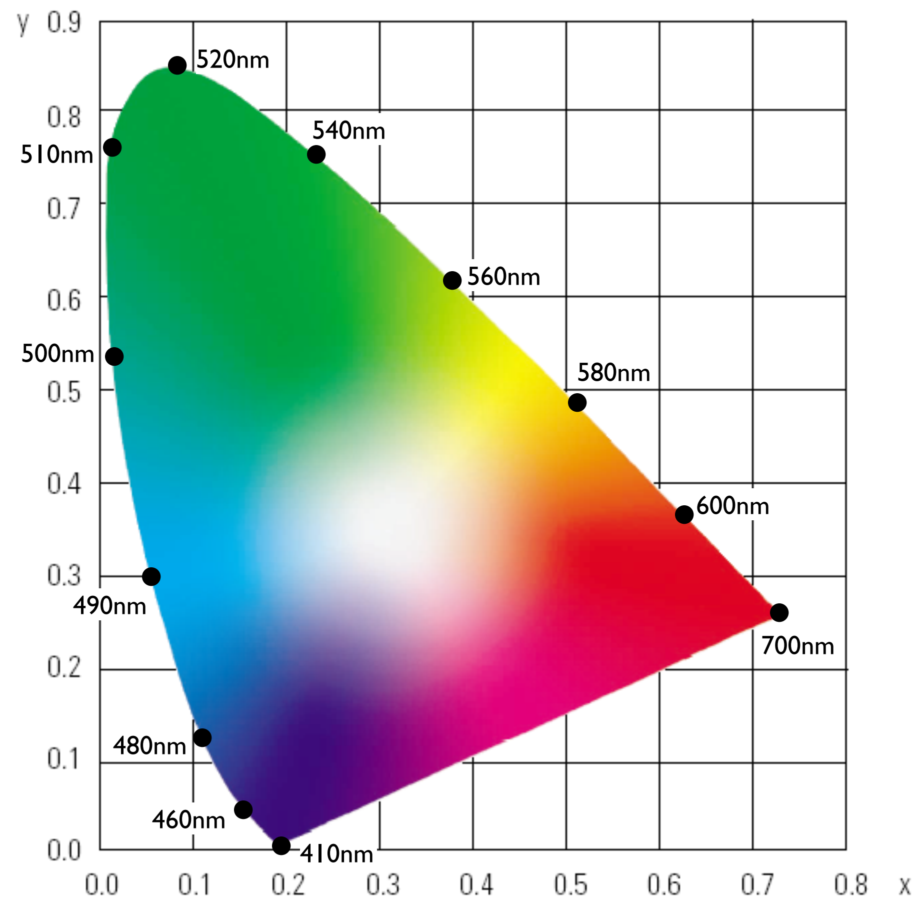
XYZ colour space was defined in 1931 by the *Commission Internationale de l'Éclairage* (CIE)

CIE chromaticity diagram

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

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

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



Colour spaces

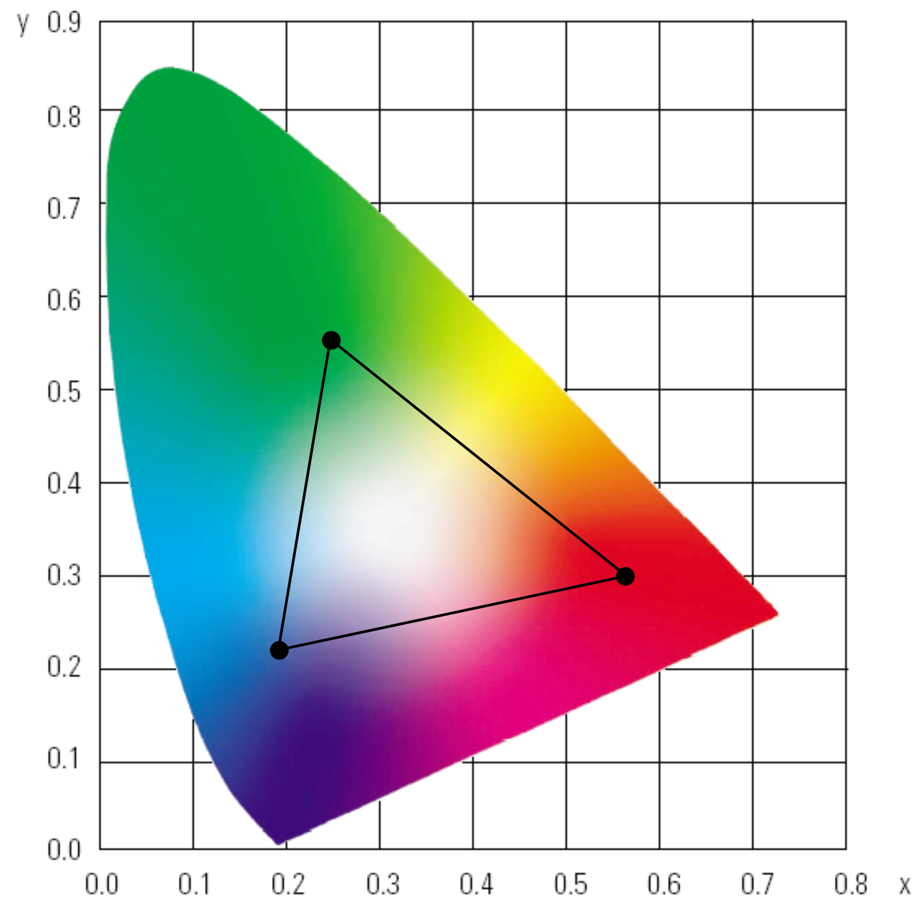
- CIE *XYZ*, *Yxy*
- Uniform
  - equal steps in any direction make equal perceptual differences
  - CIE $L^*u^*v^*$, CIE $L^*a^*b^*$
- Pragmatic
  - used because they relate directly to the way that the hardware works
  - *RGB*, *CMY*, *CMYK*
- Munsell-like
  - used in user-interfaces
  - considered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

RGB space

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

RGB in *XYZ* space

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



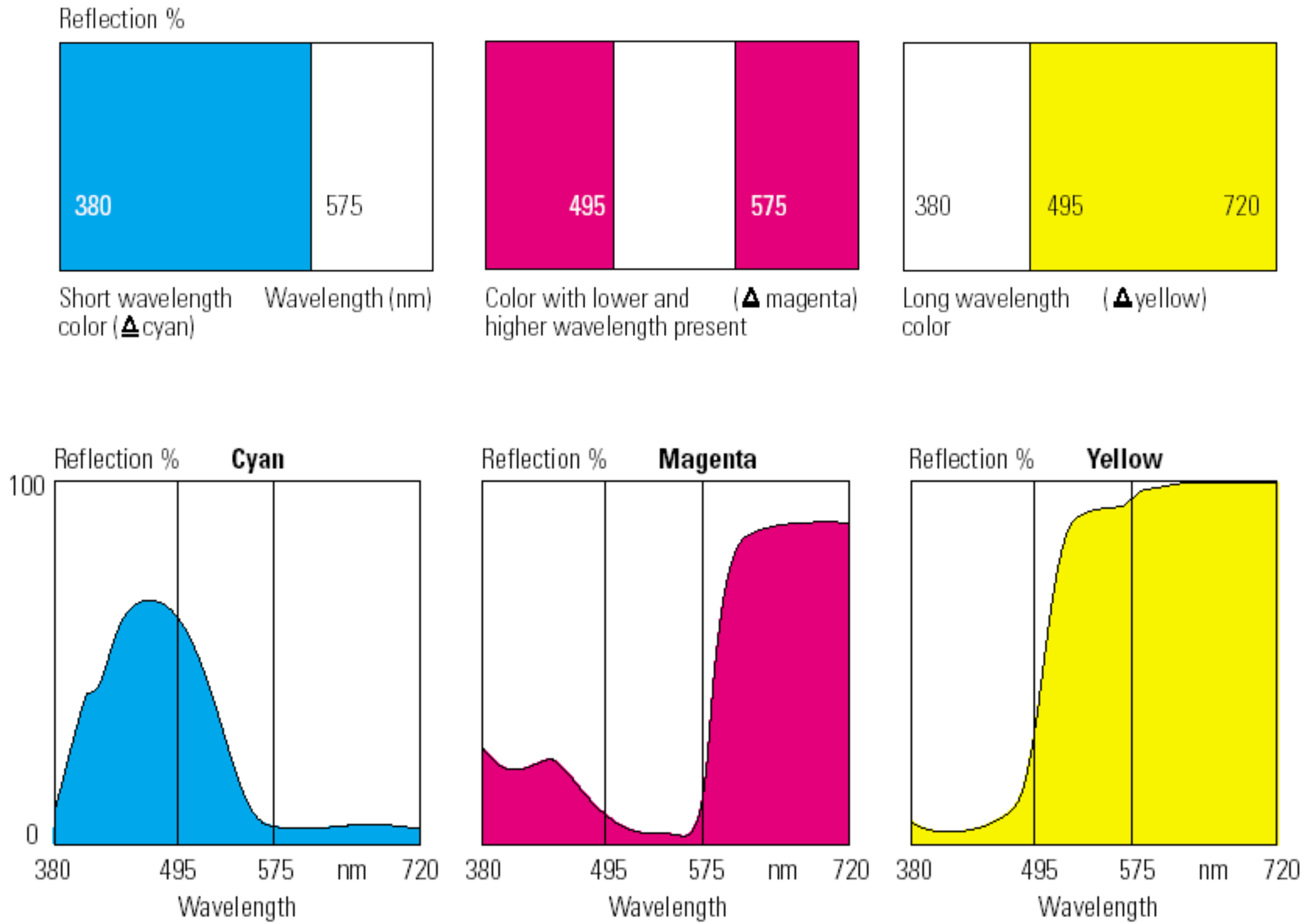
CMY space

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

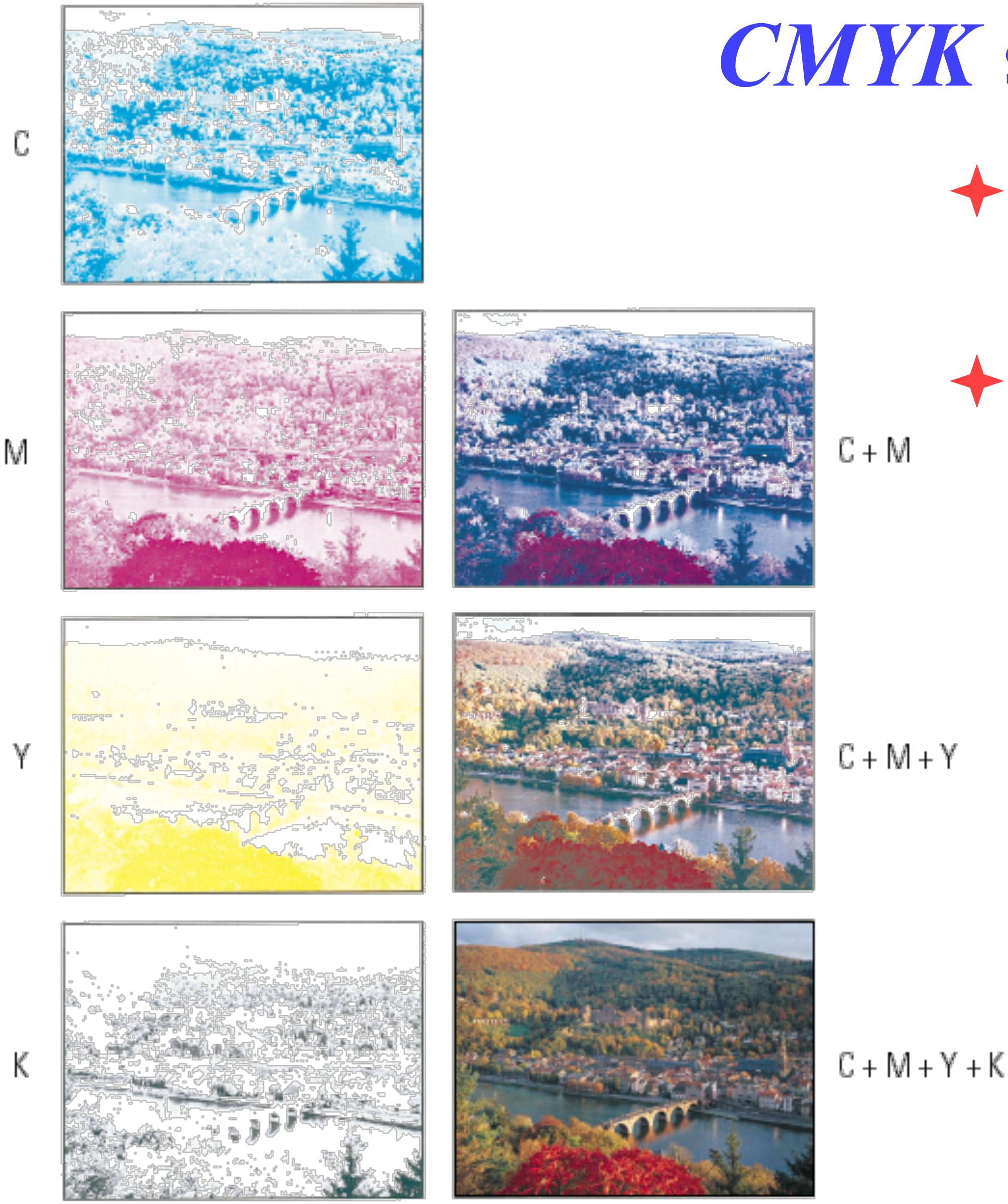
Ideal and actual printing ink reflectivities

ideal

actual

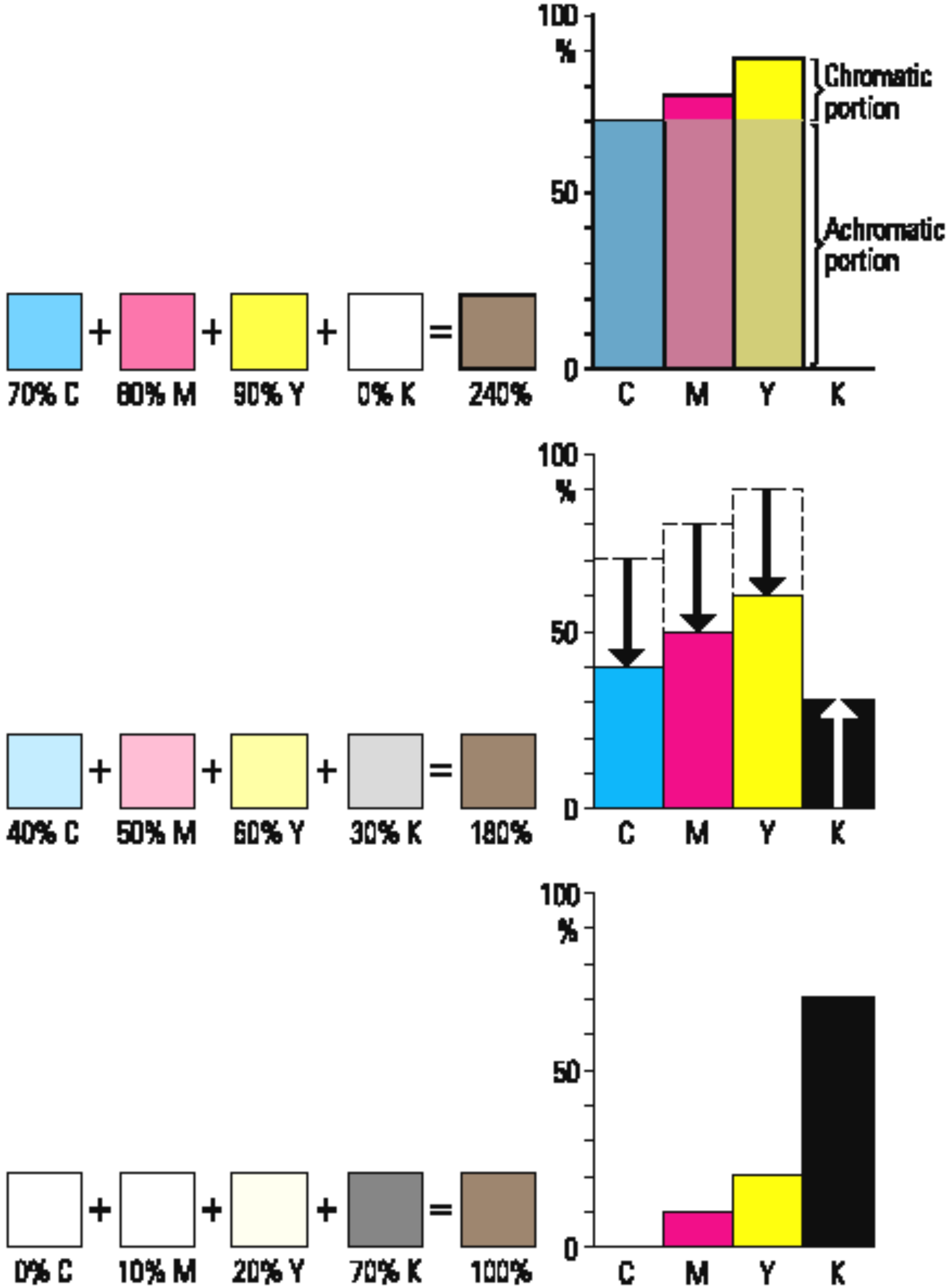


CMYK space



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

Using K



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

Image display

- a handful of technologies cover over 99% of all display devices
  - active displays

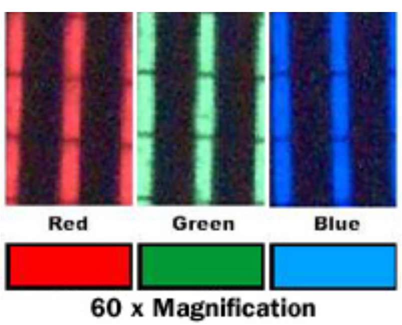
cathode ray tube	standard for late 20^{th} century
liquid crystal display	most common today
plasma displays	briefly popular but power-hungry
digital mirror displays	increasing use in video projectors

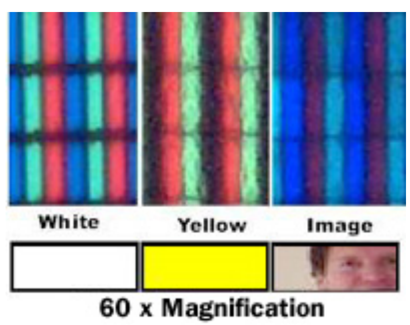
  - printers (passive displays)

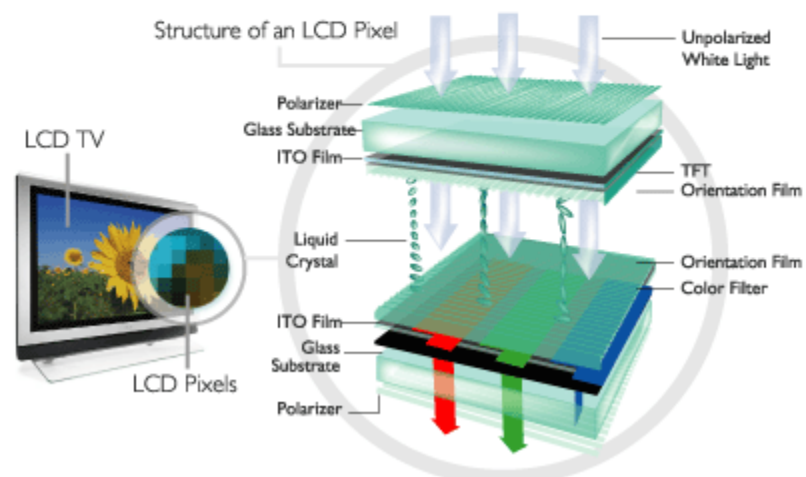
laser printers	the traditional office printer
ink jet printers	the traditional home printer
commercial printers	for high volume

Liquid crystal displays I

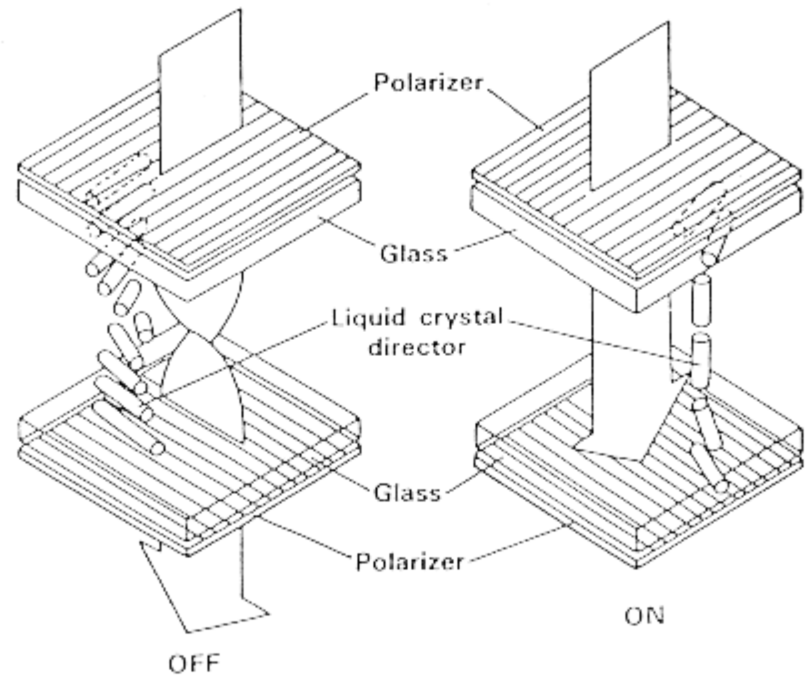
- liquid crystals can twist the polarisation of light
- basic control is by the voltage that is applied across the liquid crystal: either on or off, transparent or opaque
- greyscale can be achieved with some types of liquid crystal by varying the voltage
- colour is achieved with colour filters







Liquid crystal displays II



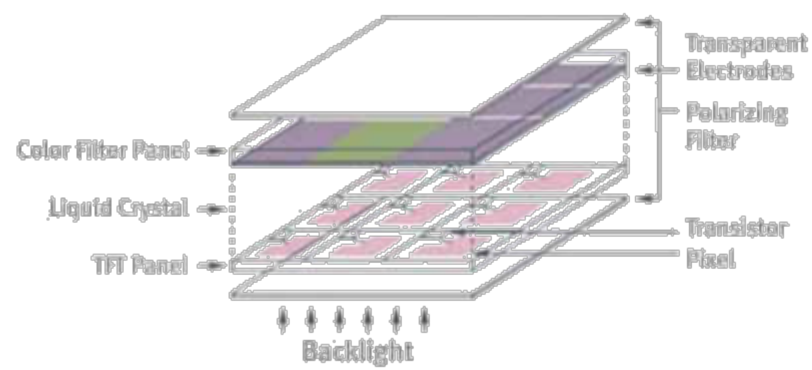
there are two polarizers at right angles to one another on either side of the liquid crystal: under normal circumstances these would block all light

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

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

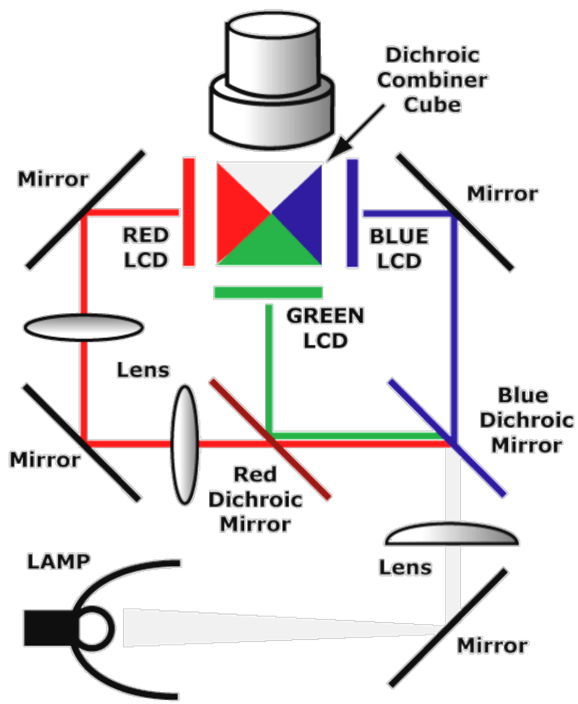
liquid crystal molecules are polar: they have a positive and a negative end

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



Liquid crystal displays III

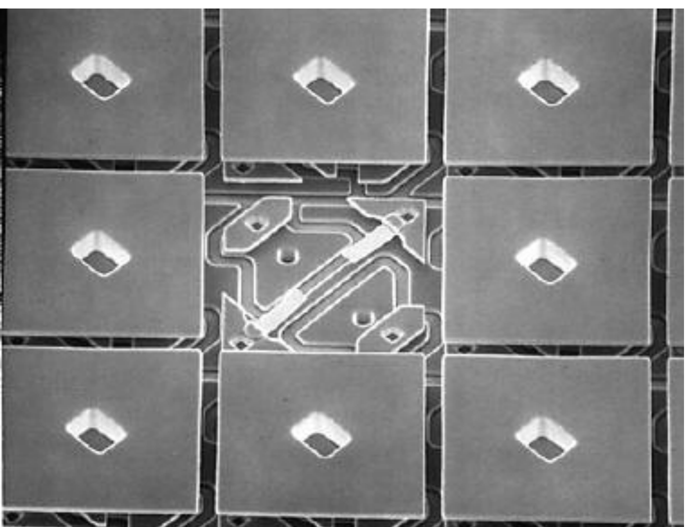
- low power consumption compared to CRTs although the back light uses a lot of power
- image quality historically not as good as cathode ray tubes, but improved dramatically over the last ten years
- uses
  - laptops
  - video projectors
  - rapidly replacing CRTs as desk top displays
  - increasing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

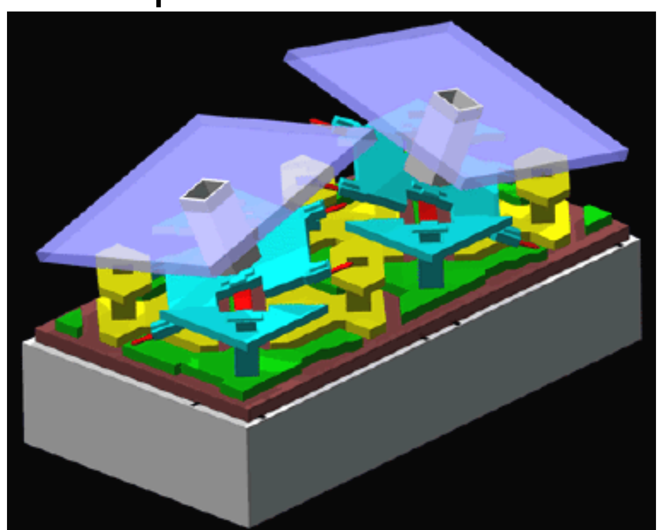


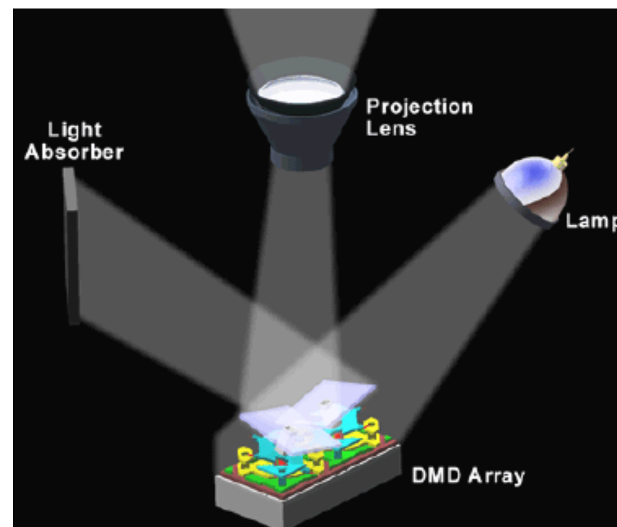
a three LCD video projector, with colour made by devoting one LCD panel to each of red, green and blue, and by splitting the light using dichroic mirrors which pass some wavelengths and reflect others

Digital micromirror devices I

- developed by Texas Instruments
  - often referred to as Digital Light Processing (DLP) technology
- invented in 1987, following ten year's work on deformable mirror devices
- manufactured like a silicon chip!
  - a standard 5 volt, 0.8 micron, CMOS process
  - micromirrors are coated with a highly reflected aluminium alloy
  - each mirror is $16\times16\mu m^2$

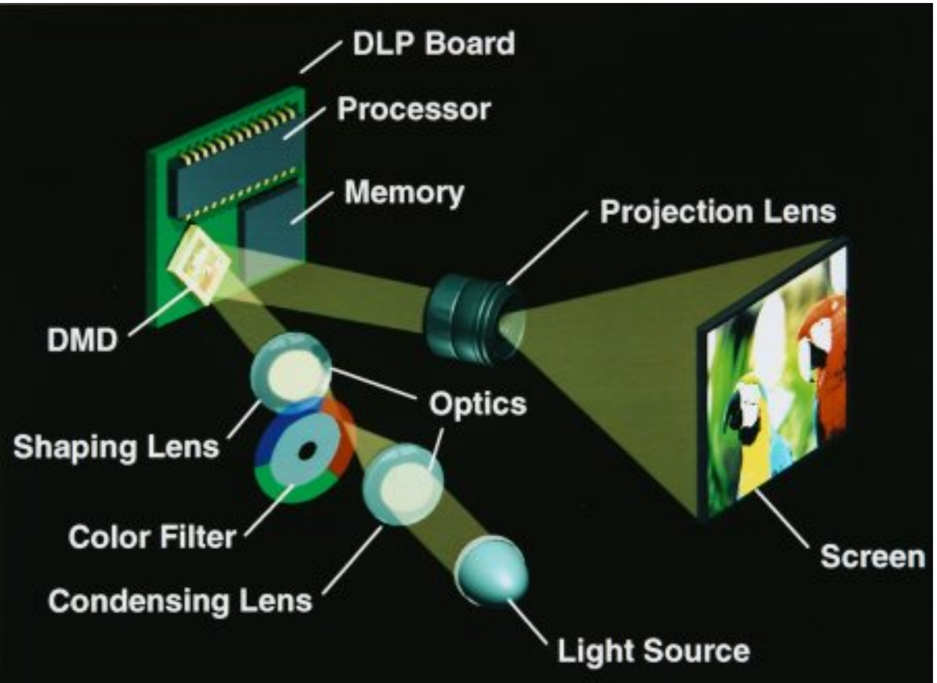


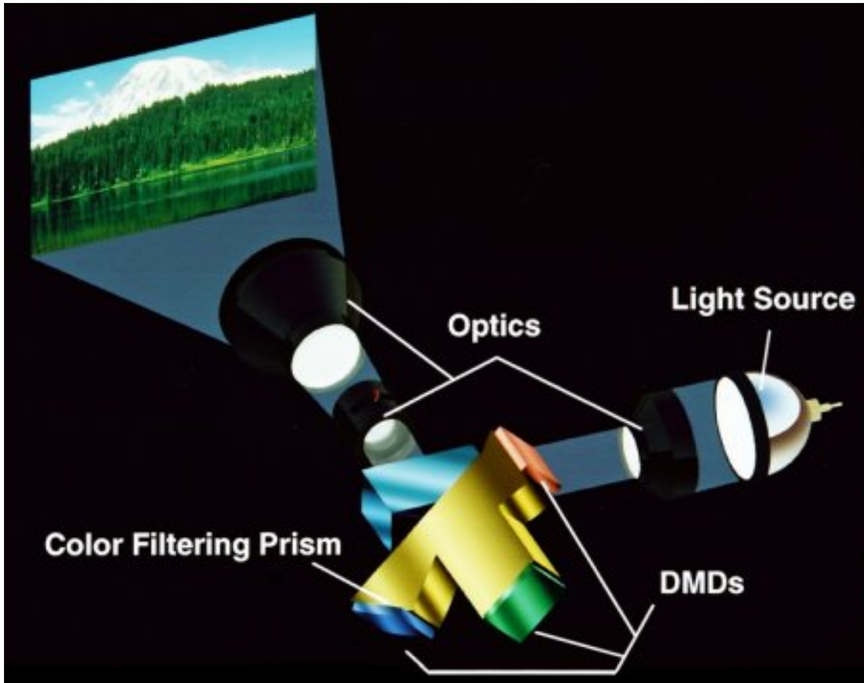




Digital micromirror devices II

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



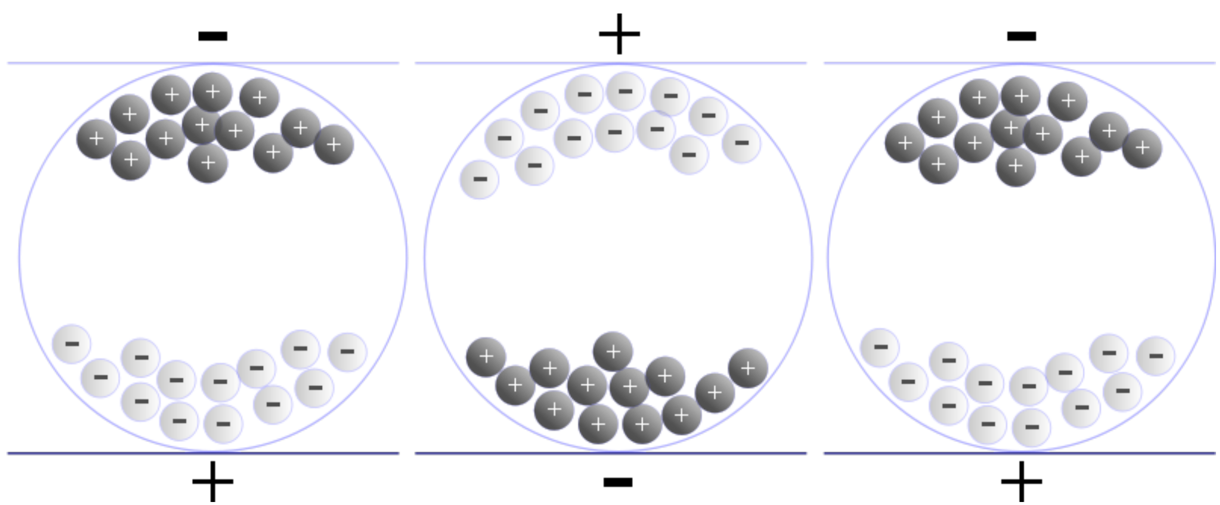


Electrophoretic displays I

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

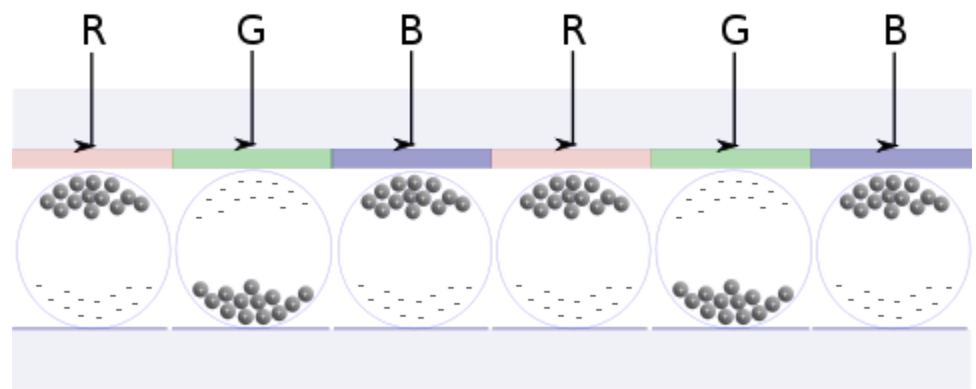
Electrophoretic displays II

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

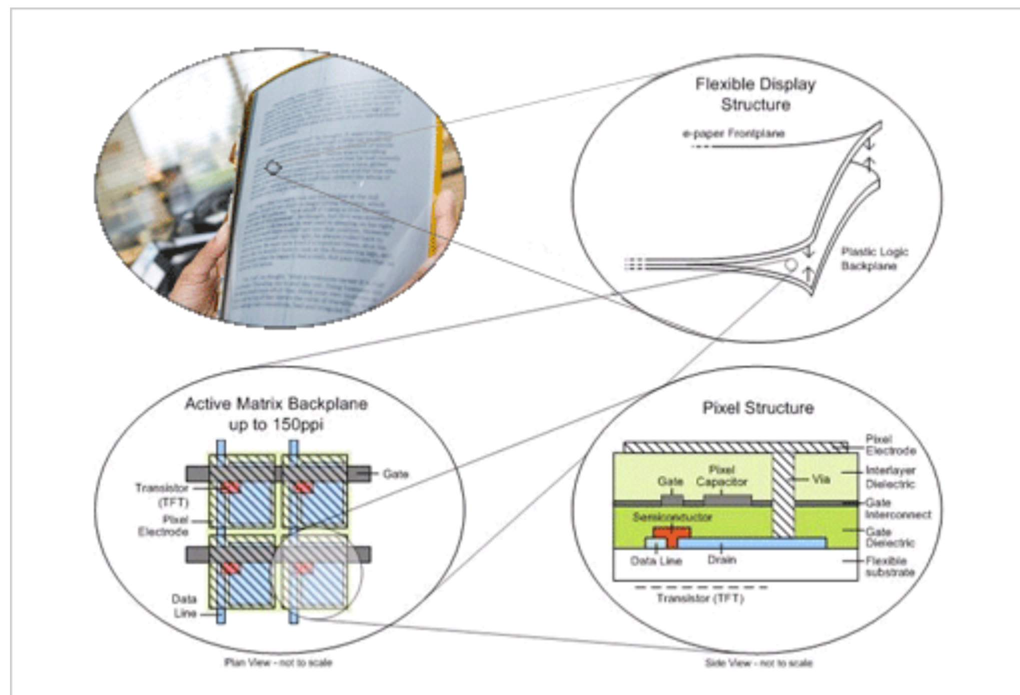


Electrophoretic displays III

- colour filters over individual pixels



- flexible substrate using plastic semiconductors (Plastic Logic)



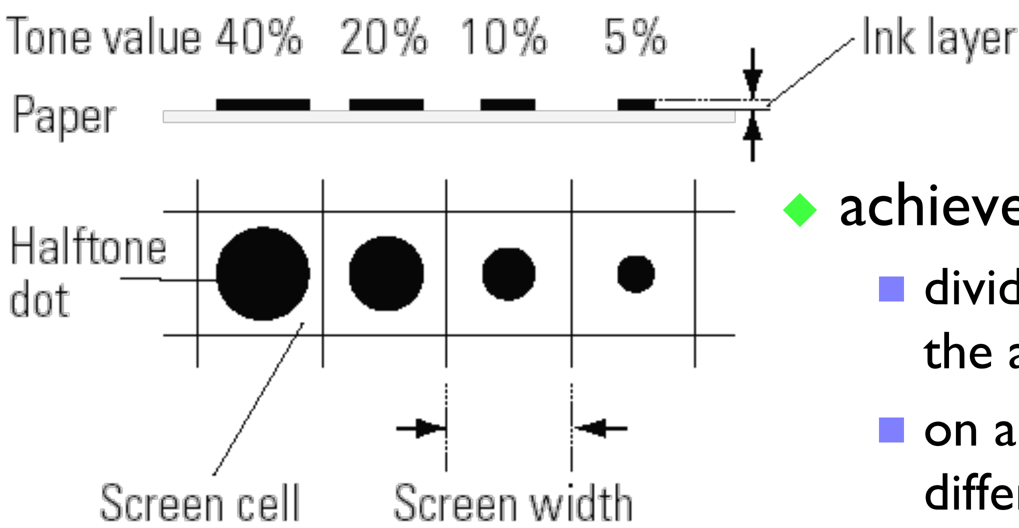
Printers

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

Printer resolution

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

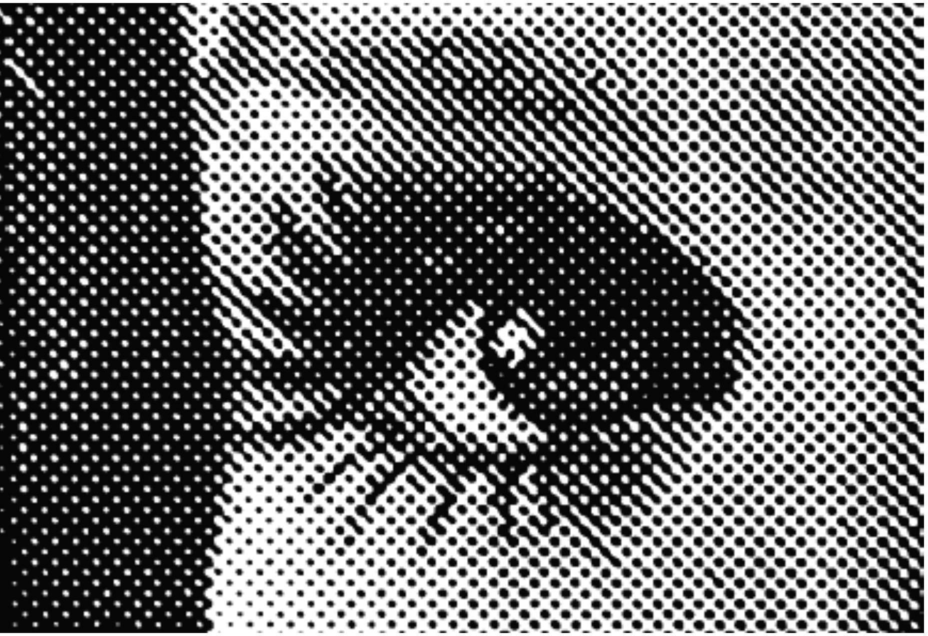
What about greyscale?

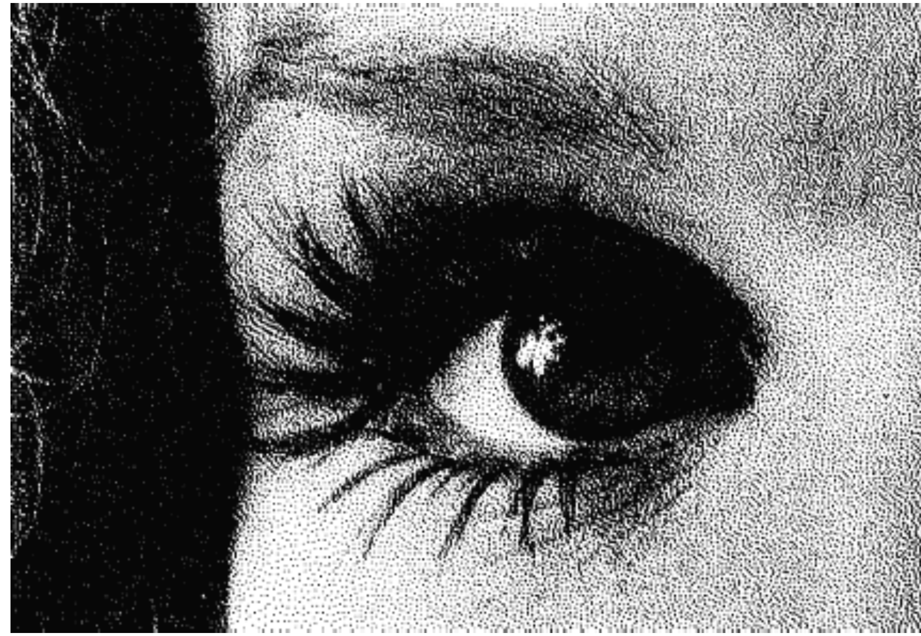


- achieved by halftoning
  - divide image into cells, in each cell draw a spot of the appropriate size for the intensity of that cell
  - on a printer each cell is $m \times m$ pixels, allowing m^2+1 different intensity levels
  - e.g. 300dpi with 4×4 cells ⇒ 75 cells per inch, 17 intensity levels
 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

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

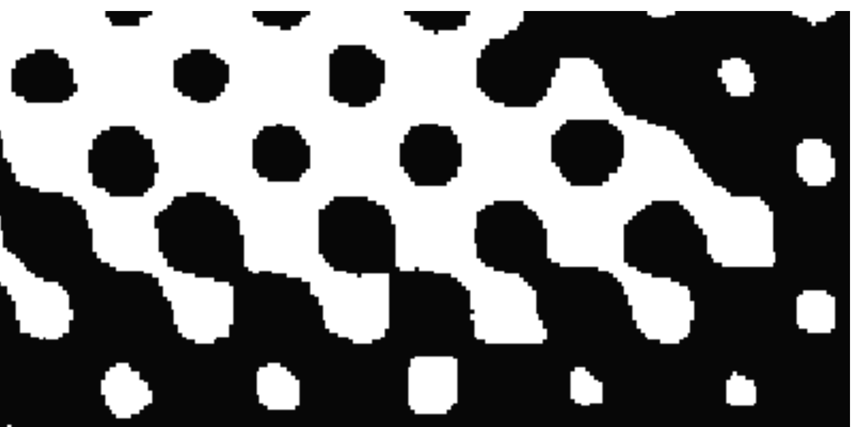
Halftoning & dithering examples

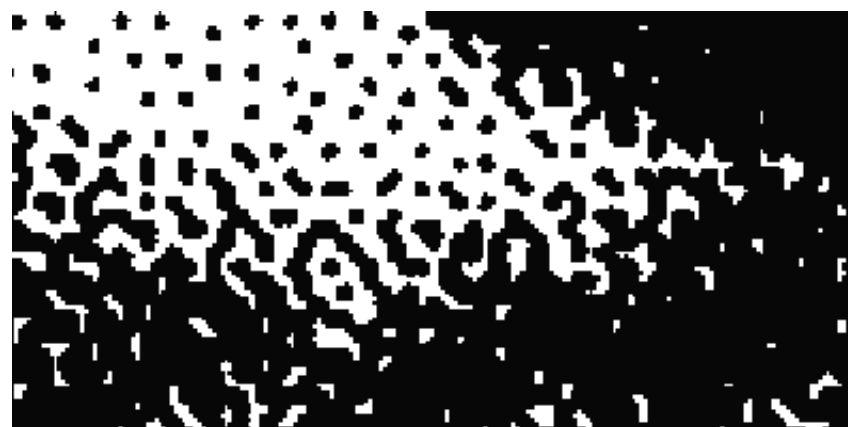




Halftoning

Dithering





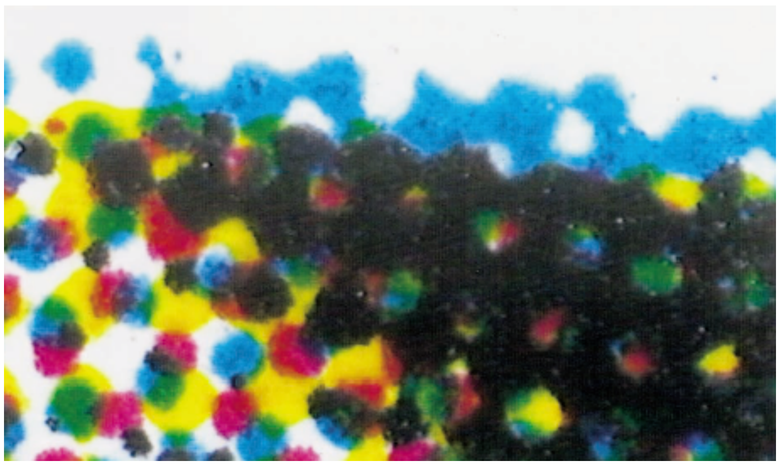
What about colour?

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

see slide 221 CMYK space

How do you produce halftoned colour?

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

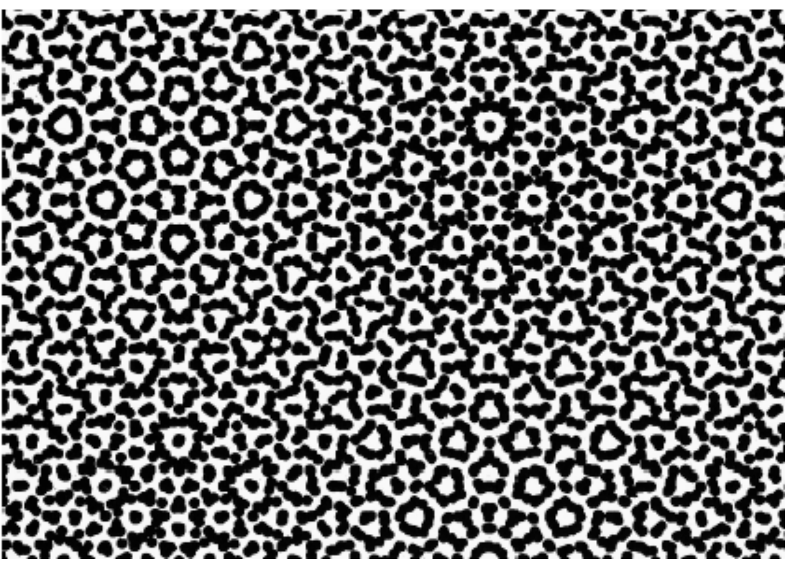


150 lpi × 16 dots per cell
= 2400 dpi phototypesetter
(16×16 dots per cell = 256
intensity levels)

Standard rulings (in lines per inch)

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

Four colour halftone screens



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

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

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

- At bottom is the moiré pattern
  - this is the best possible (minimal) moiré pattern
  - produced by this optimal set of angles
  - all four colours printed in black to highlight the effect

Range of printable colours

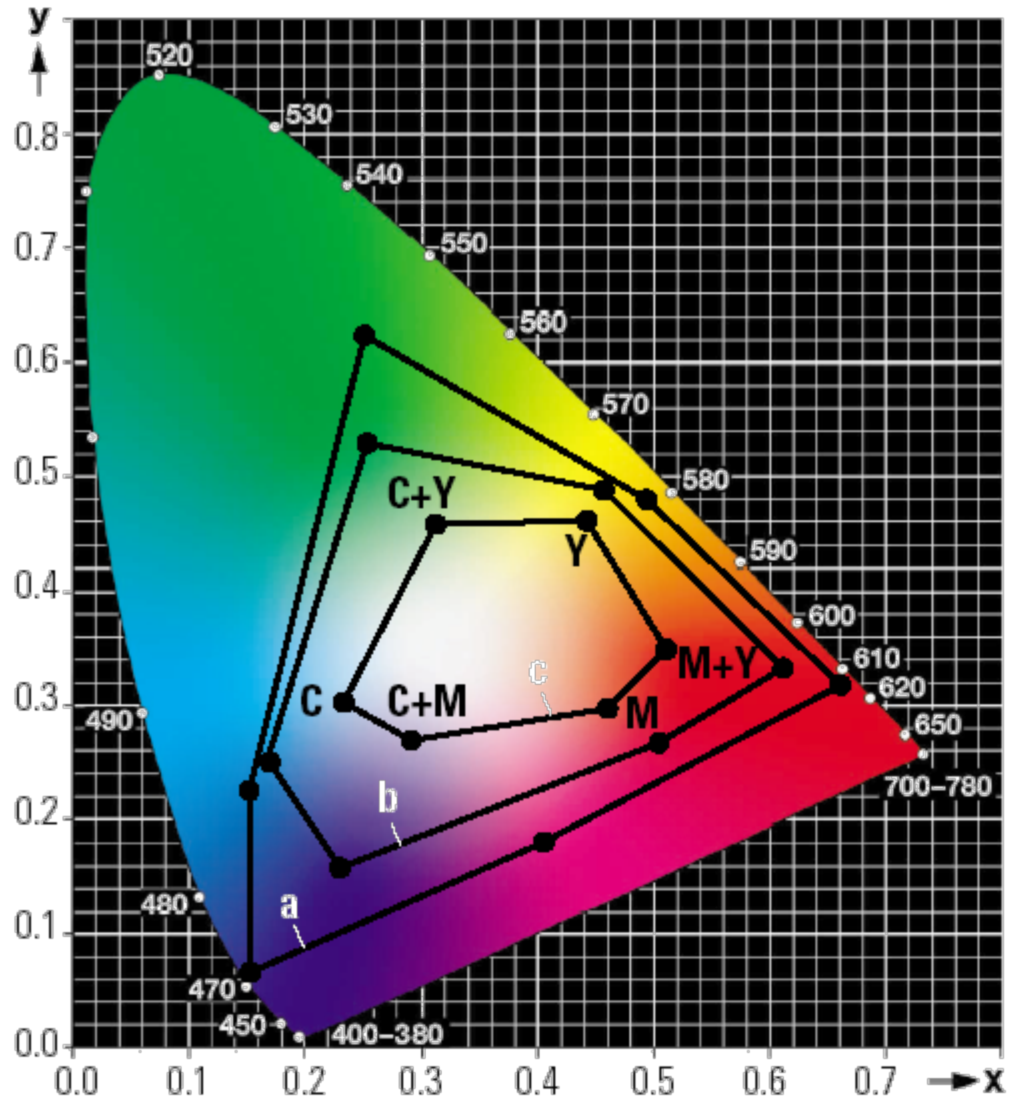
a: colour photography (diapositive)

b: high-quality offset printing

c: newspaper printing

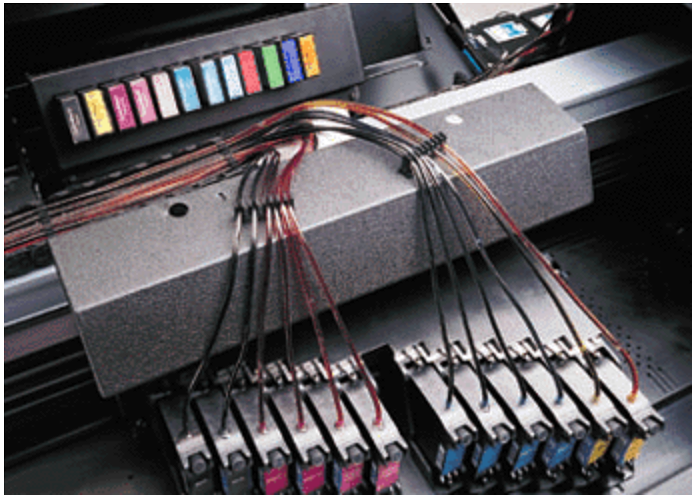
why the hexagonal shape?

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



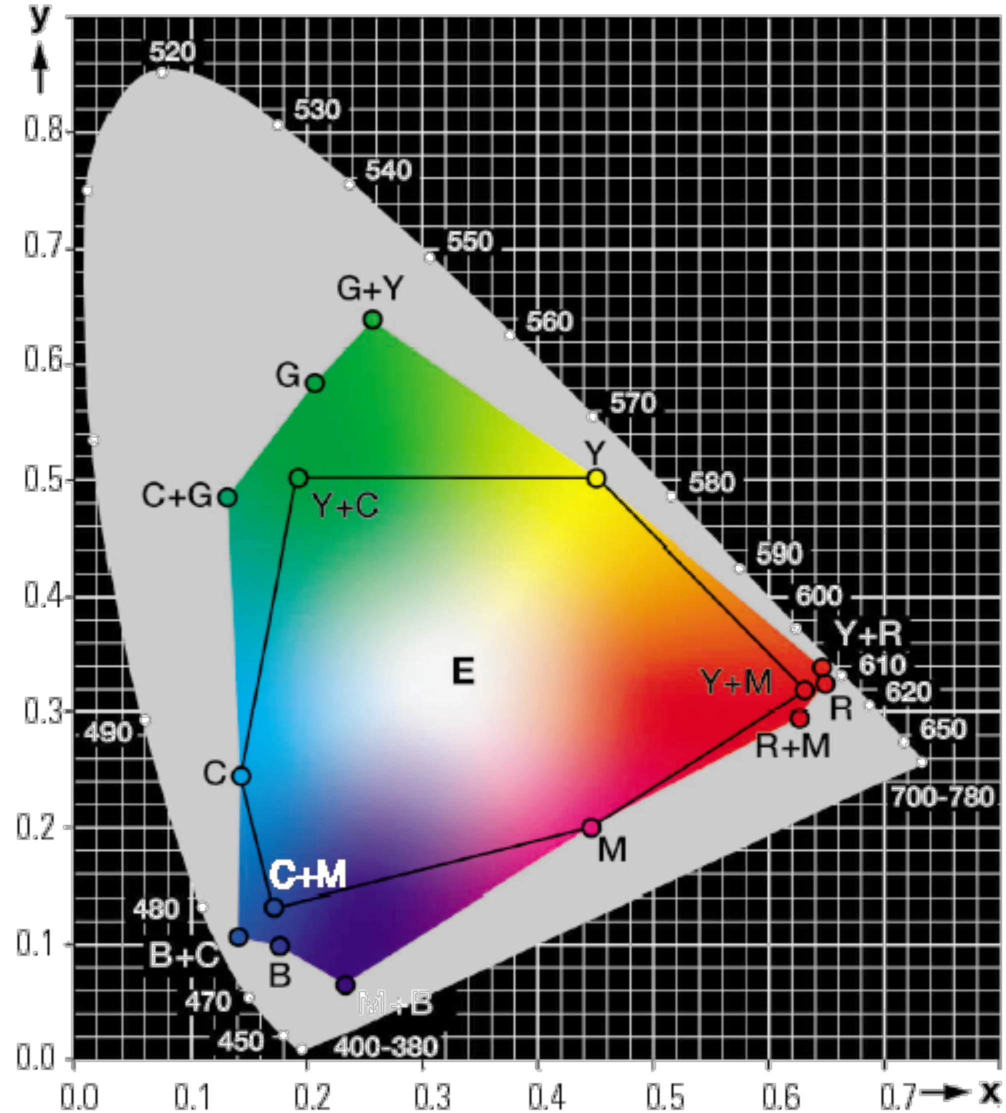
Beyond four colour printing

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

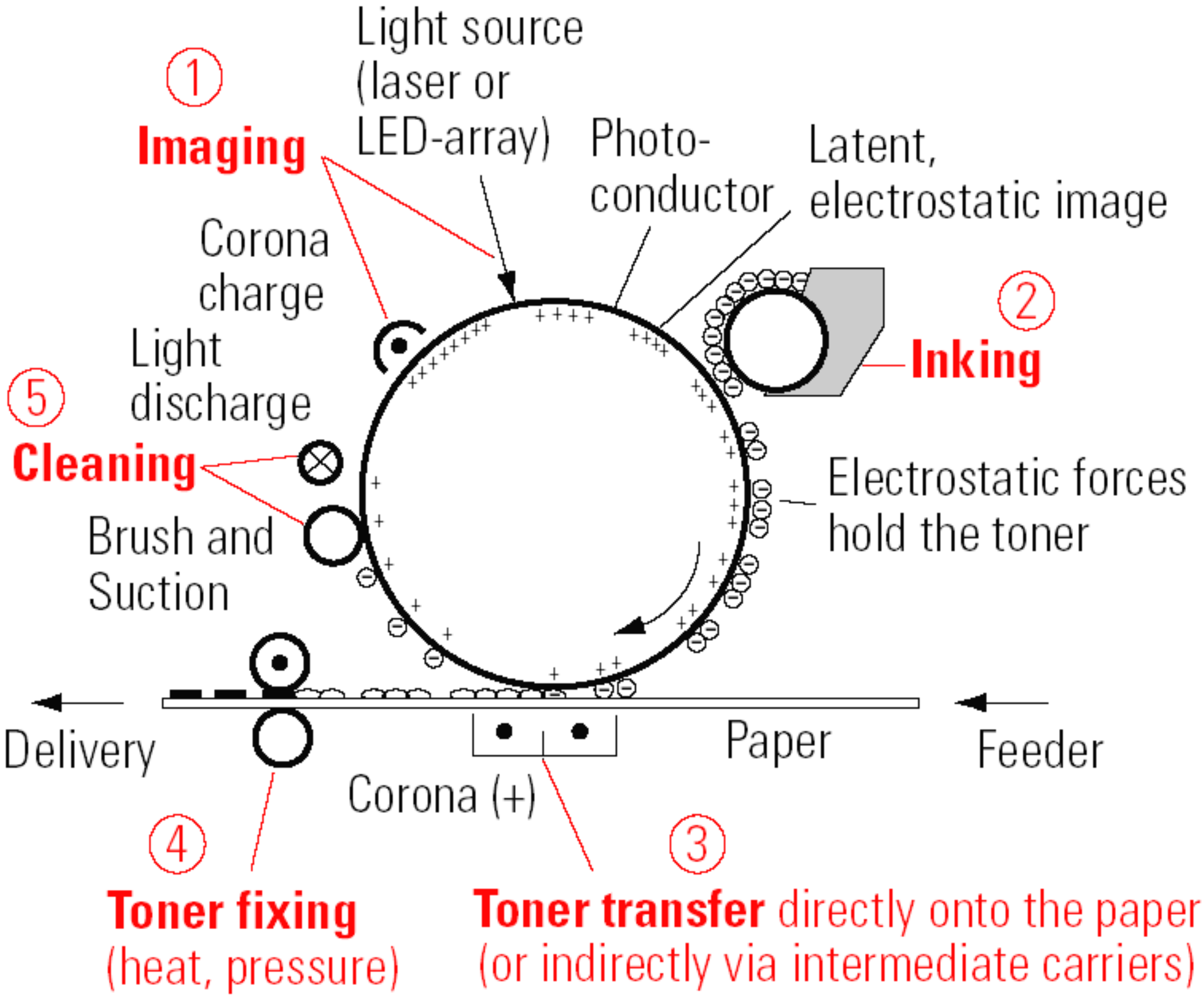


The extra range of colour

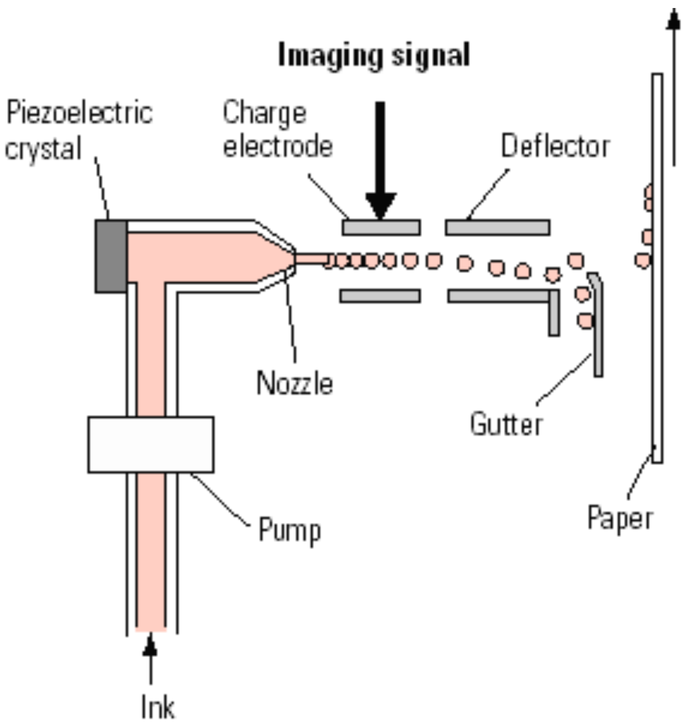
- this gamut is for so-called HiFi colour printing
  - uses cyan, magenta, yellow, plus red, green and blue inks



Laser printer

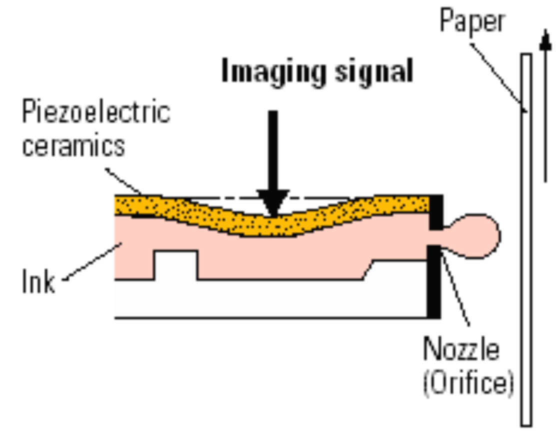


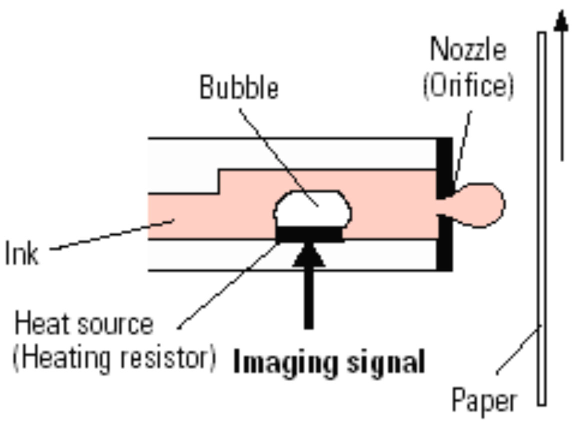
Ink jet printers



continuous ink jet
(left)

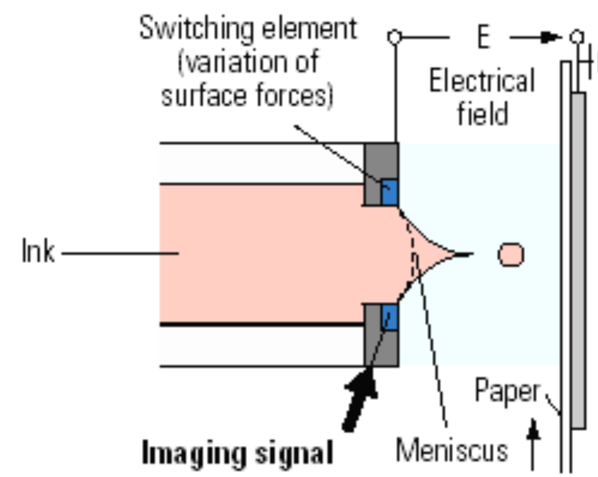
piezo ink jet
(right)



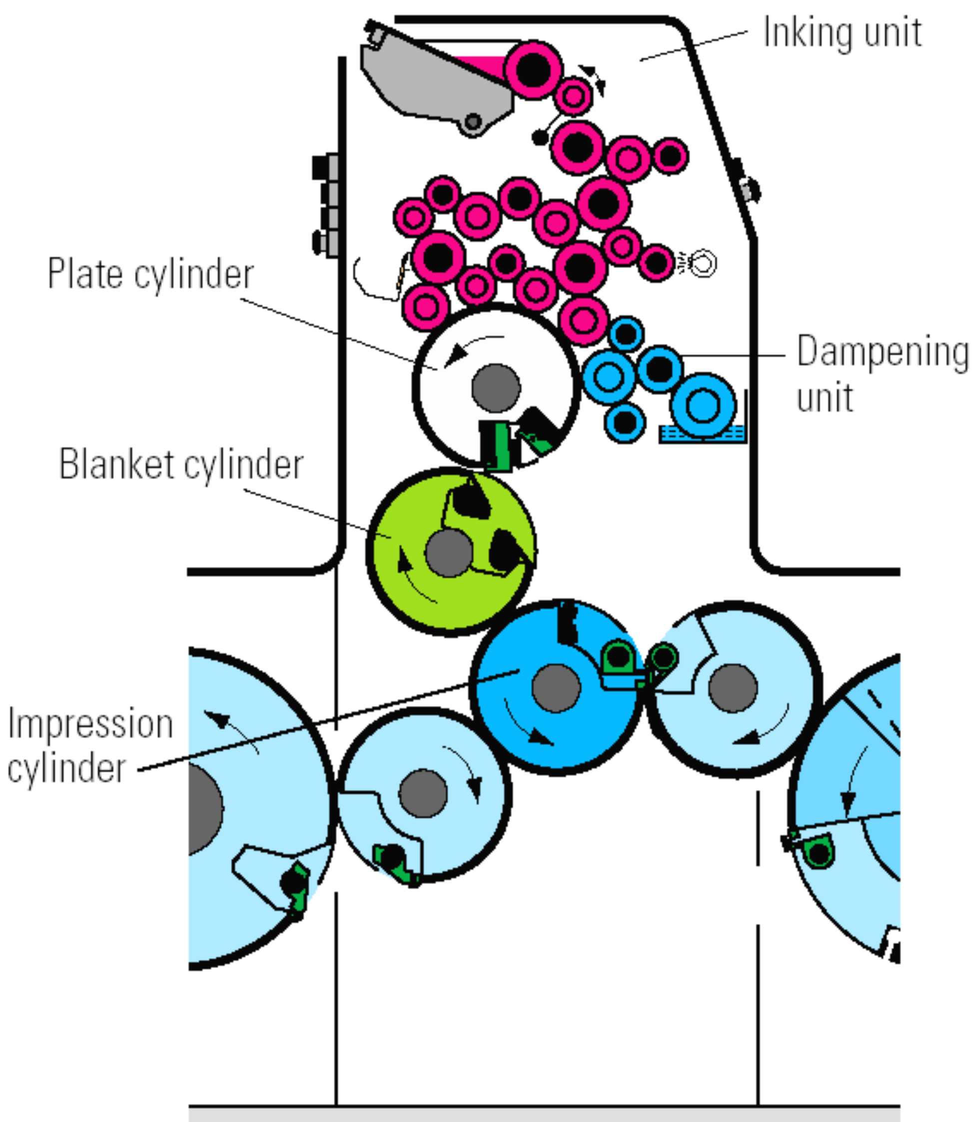


thermal ink jet
or bubble jet
(left)

electrostatic ink jet
(right)



Commercial offset printing



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

Course review

- Background
- Rendering
- Graphics pipeline
- Graphics hardware and modern OpenGL
- Technology

What next?

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

And then?

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