UNIVERSITY OF CAMBRIDGE

Introduction to Graphics

Computer Science Tripos Part 1A/1B
Michaelmas Term 2022/2023

Department of
Computer Science
and Technology
The Computer Laboratory

William Gates Building
15 JJ Thomson Avenue
Cambridge
CB3 0FD

www.cst.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 textbooks. They are intended to give a reasonable synopsis of the subjects discussed, but they give neither complete descriptions nor all the background material.

Selected slides contain a reference to the relevant section in the recommended textbook for this course: *Fundamentals of Computer Graphics* by Marschner & Shirley, CRC Press 2015 (4th or 5th edition). The references are in the format [FCG A.B/C.D], where A.B is the section number in the 4th edition and C.D is the section number in the 5th edition.

Introduction to Computer Graphics

Rafał Mantiuk

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

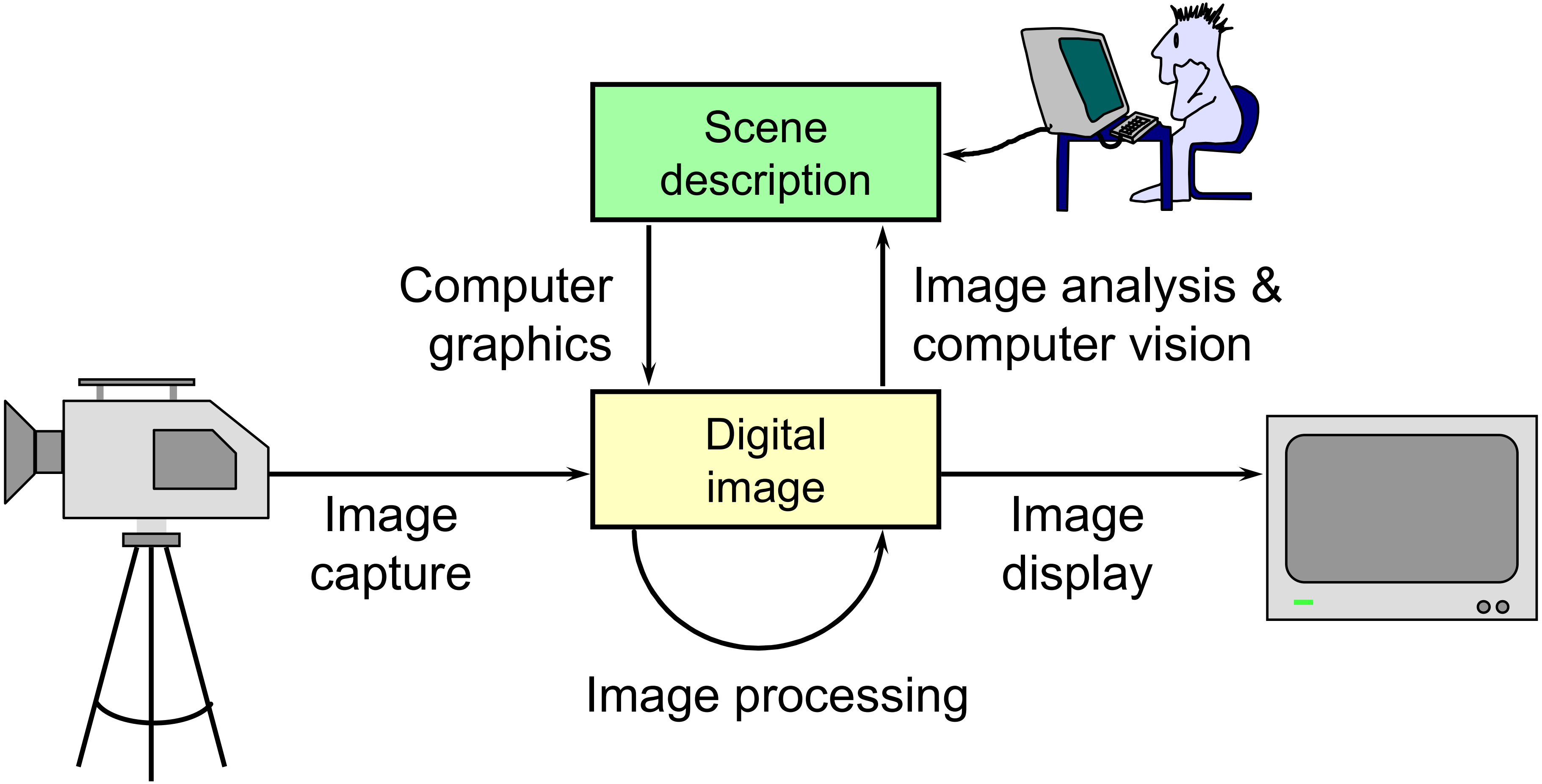
Eight lectures & two practical tasks

Part IA CST

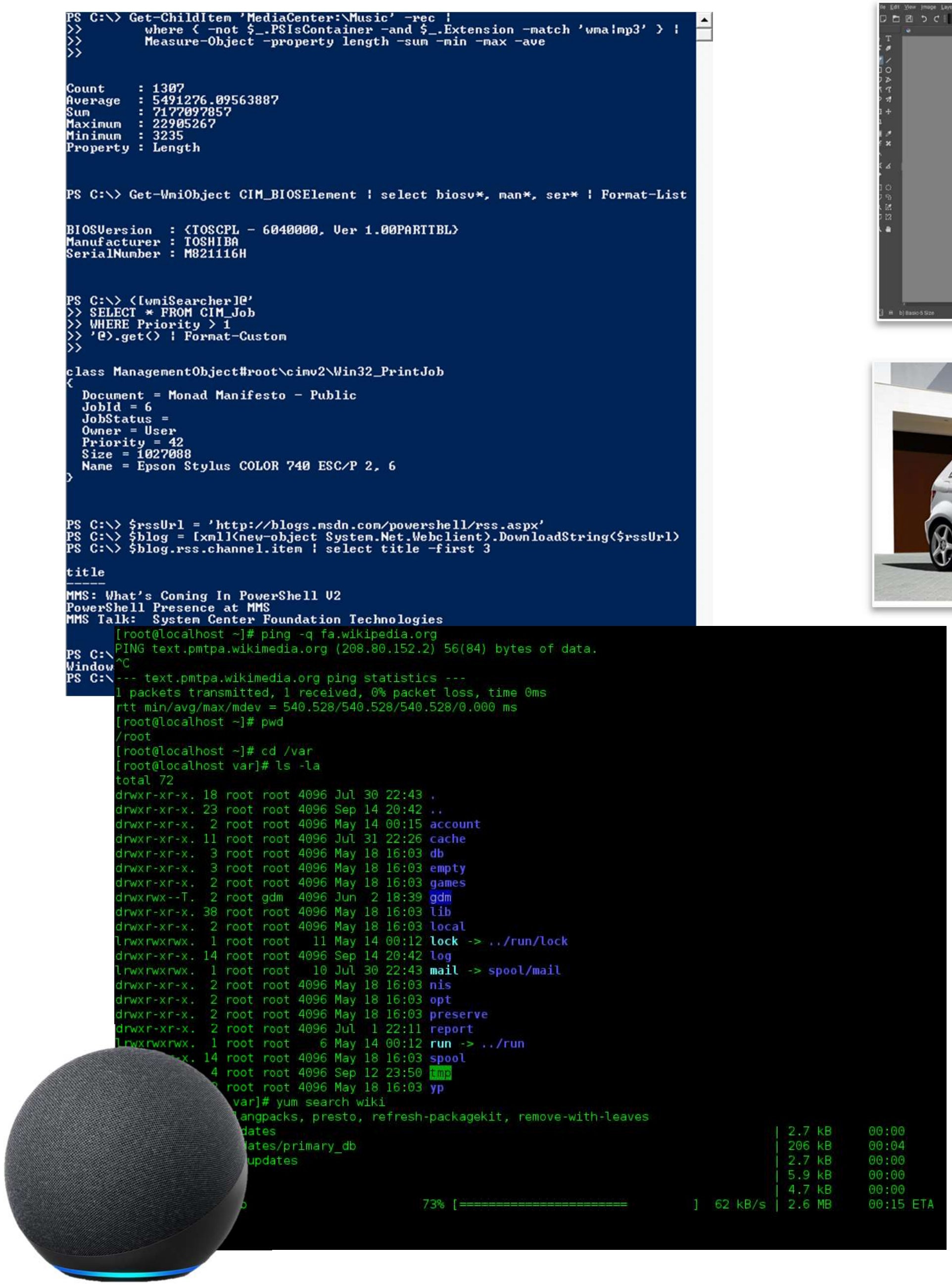
Two supervisions suggested

Two exam questions on Paper 3

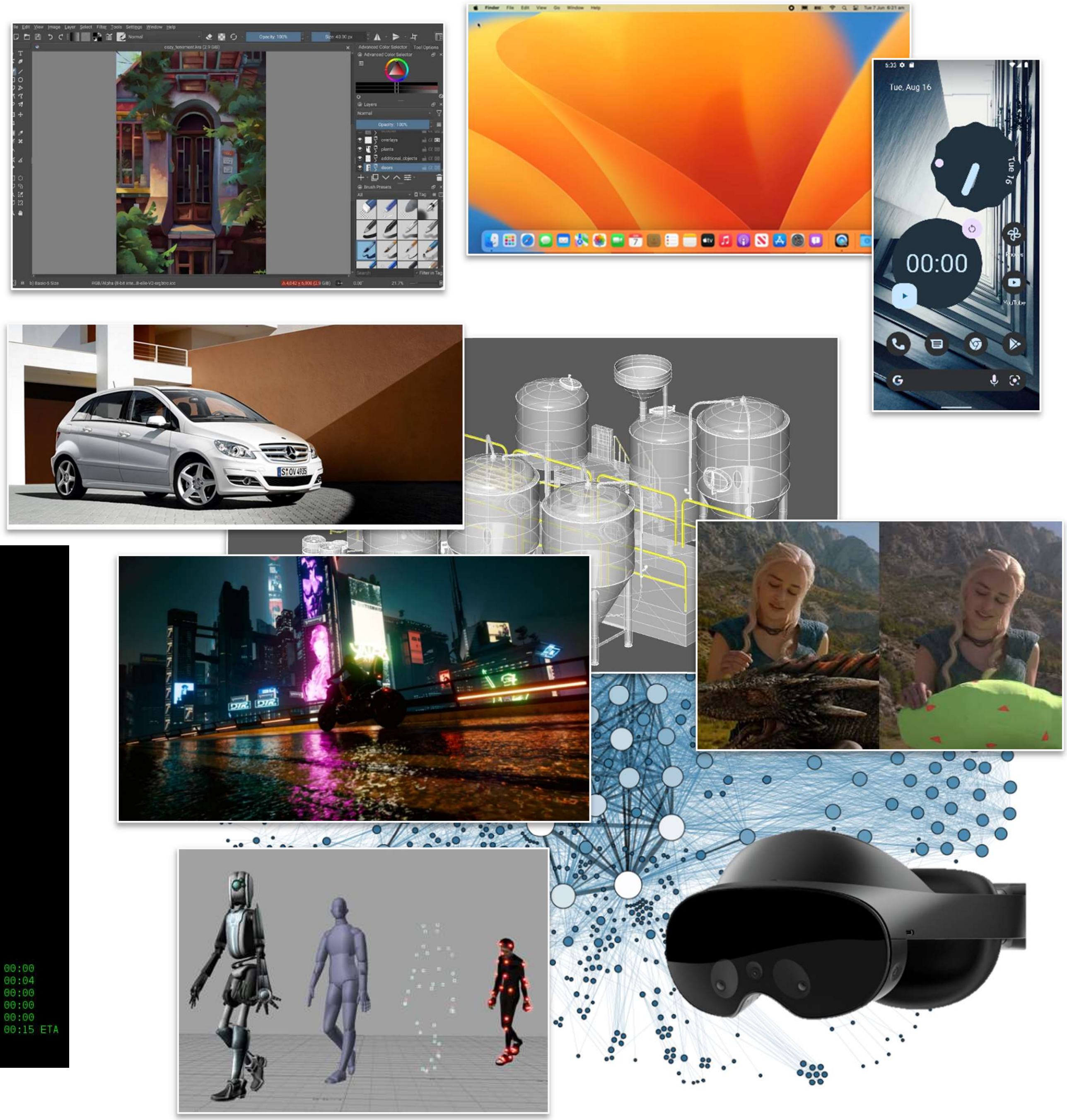
What are Computer Graphics & Image Processing?



Computing without graphics



Computing with graphics



Why bother with CG?

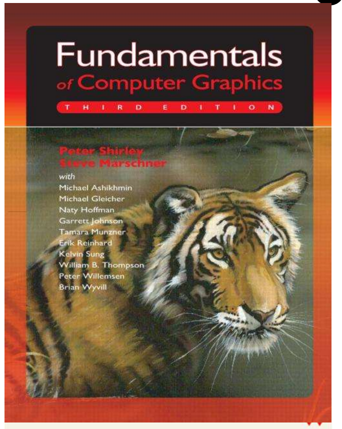
- *All* visual computer output depends on CG
  - printed output (laser/ink jet/phototypesetter)
  - monitor (CRT/LCD/OLED/DMD)
  - all visual computer output consists of real images generated by the computer from some internal digital image
- Much other visual imagery depends on CG
  - TV & movie special effects & post-production
  - most books, magazines, catalogues…
  - VR/AR

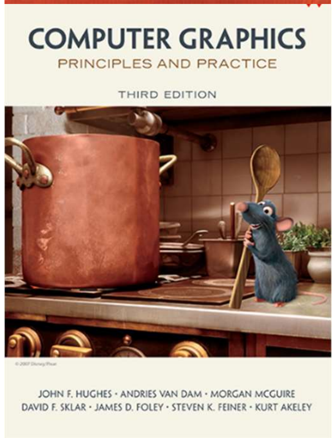
Course Structure

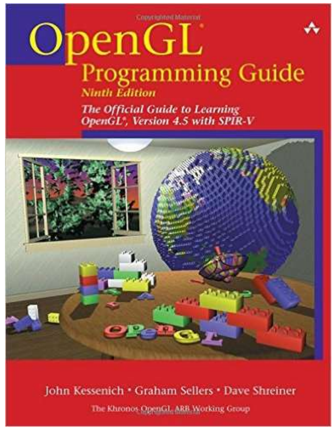
- **Background**
  - What is an image? Resolution and quantisation. Storage of images in memory. [1 lecture]
- **Rendering**
  - Perspective. Reflection of light from surfaces and shading. Geometric models. Ray tracing. [2 lectures]
- **Graphics pipeline**
  - Polygonal mesh models. Transformations using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

Course books

- ***Fundamentals of Computer Graphics***
  - Shirley & Marschner
    CRC Press 2015 (4th or 5th edition)
  - [FCG 8.1/9.1] – reference to section 3.1 in the 4th edition, 9.1 in the 5th 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?
  - Resolution and quantisation
  - Storage of images in memory
- **Rendering**
- **Graphics pipeline**
- **Rasterization**
- **Graphics hardware and modern OpenGL**
- **Human vision and colour & tone mapping**

What is a (digital) image?

- A digital photograph? ("JPEG")
- A snapshot of real-world lighting?

Image

From computing perspective (discrete)

2D array of pixels

- To represent images in memory
- To create image processing software

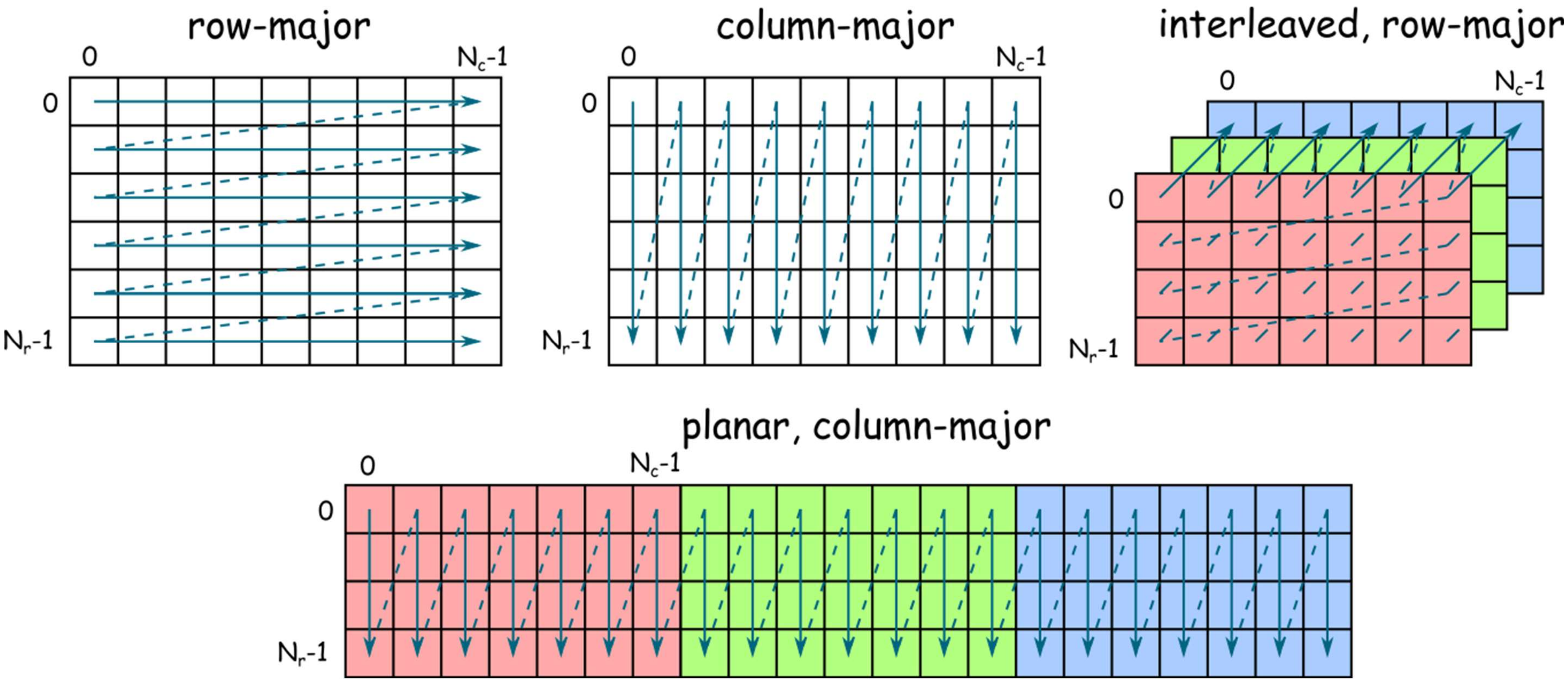
From mathematical perspective (continuous)

2D function

- To express image processing as a mathematical problem
- To develop (and understand) algorithms

Image

- 2D array of pixels
- In most cases, each pixel takes 3 bytes: one for each red, green and blue
- But how to store a 2D array in memory?



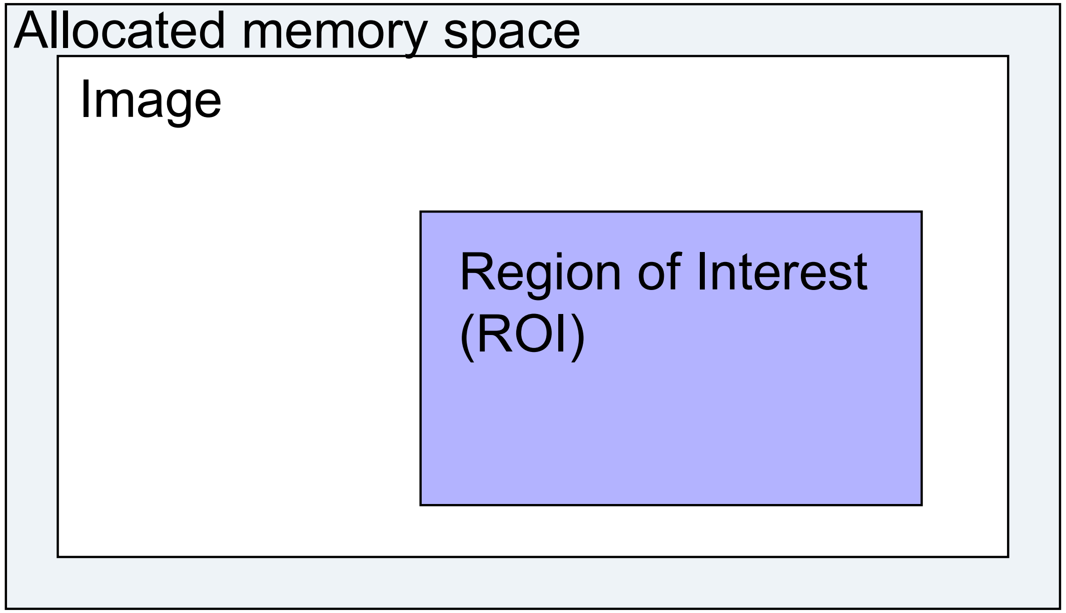
Stride

- Calculating the pixel component index in memory
  - For row-major order (grayscale)
    $$i(x, y) = x + y \cdot n_{cols}$$
  - For column-major order (grayscale)
    $$i(x, y) = x \cdot n_{rows} + y$$
  - For interleaved row-major (colour)
    $$i(x, y, c) = x \cdot 3 + y \cdot 3 \cdot n_{cols} + c$$
  - General case
    $$i(x, y, c) = x \cdot s_x + y \cdot s_y + c \cdot s_c$$

where s_x, s_y and s_c are the strides for the x, y and colour dimensions

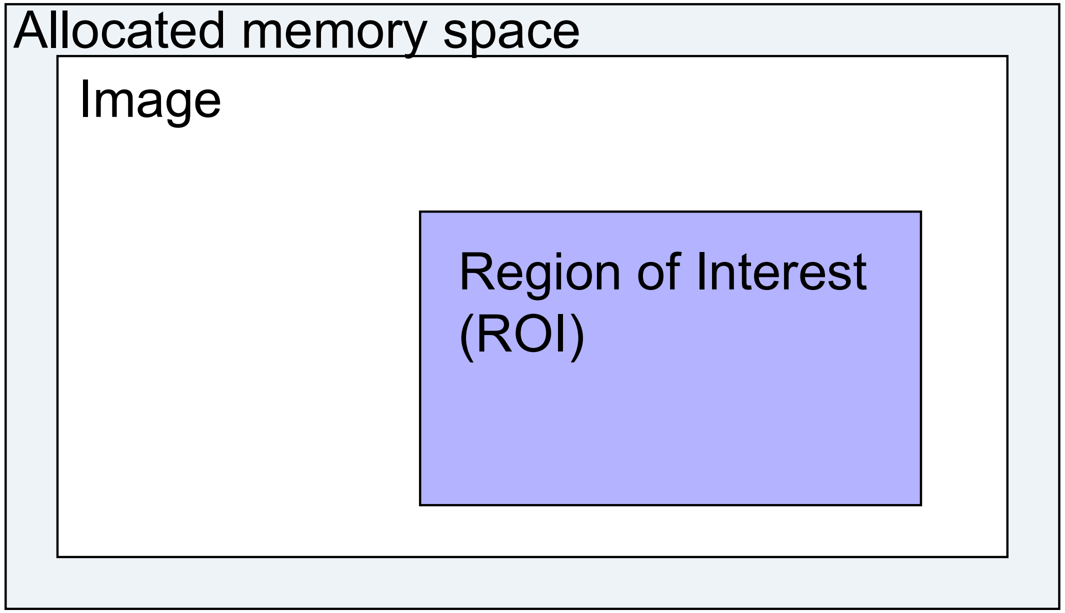
Padded images and stride

- Sometimes it is desirable to “pad” image with extra pixels
  - for example when using operators that need to access pixels outside the image border
- Or to define a region of interest (ROI)



- How to address pixels for such an image and the ROI?

Padded images and stride



$$i(x, y, c) = i_{first} + x \cdot s_x + y \cdot s_y + c \cdot s_c$$

- For row-major, interleaved, colour
  - $i_{first} =$
  - $s_x =$
  - $s_y =$
  - $s_c =$

Pixel (PIcture ELement)

- Each pixel (usually) consist of three values describing the color

  (red, green, blue)

- For example
  - (255, 255, 255) for white
  - (0, 0, 0) for black
  - (255, 0, 0) for red
- Why are the values in the 0-255 range?
- How many bytes are needed to store 5MPixel image? (uncompressed)

Pixel formats, bits per pixel, bit-depth

- Grayscale – single **color channel**, 8 bits (1 byte)
- Highcolor – 2^{16}=65,536 colors (2 bytes)

Sample Length:	5					6						5				
Channel Membership:	Red					Green						Blue				
Bit Number:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RGBAX	R. G. B. A. X															
Sample Length Notation:	5.6.5.0.0															

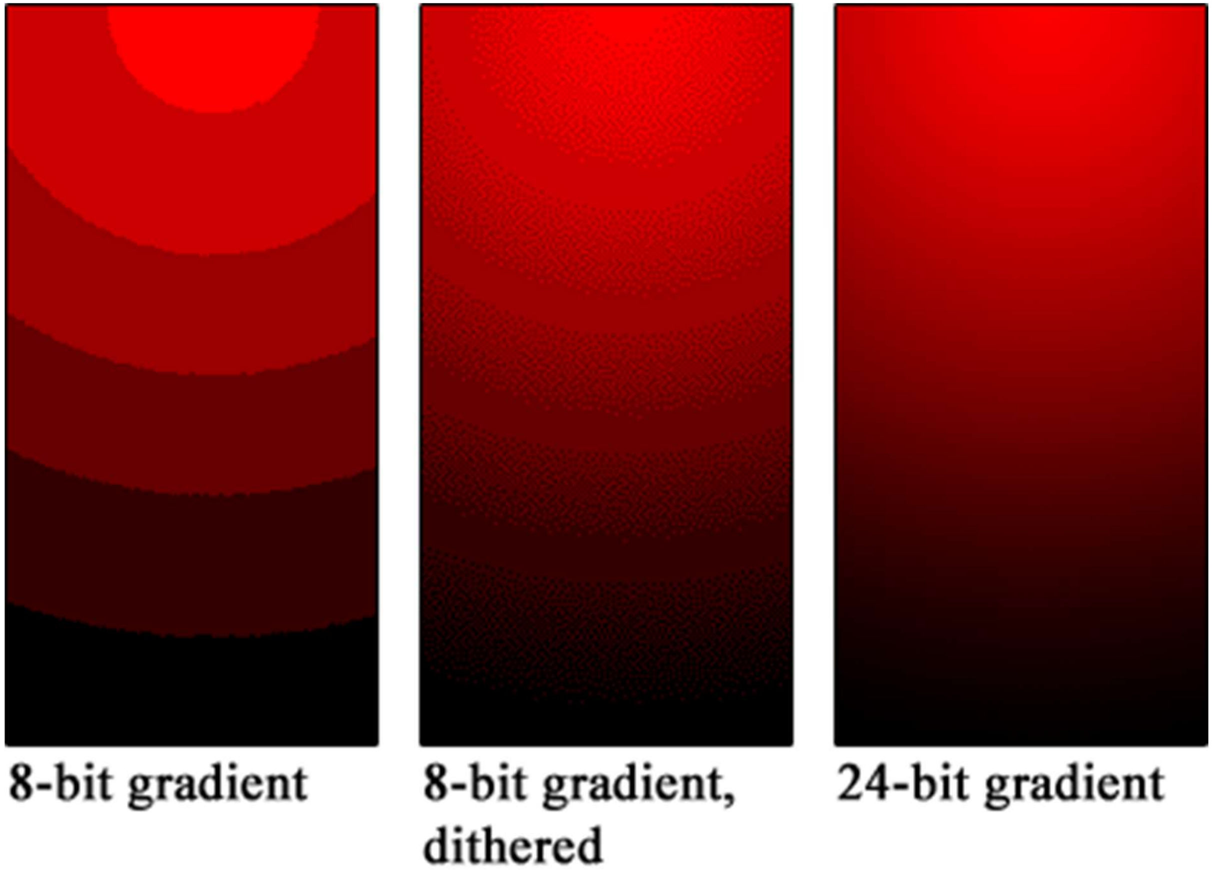
- Truecolor – 2^{24} = 16,8 million colors (3 bytes)
- Deepcolor – even more colors (>= 4 bytes)

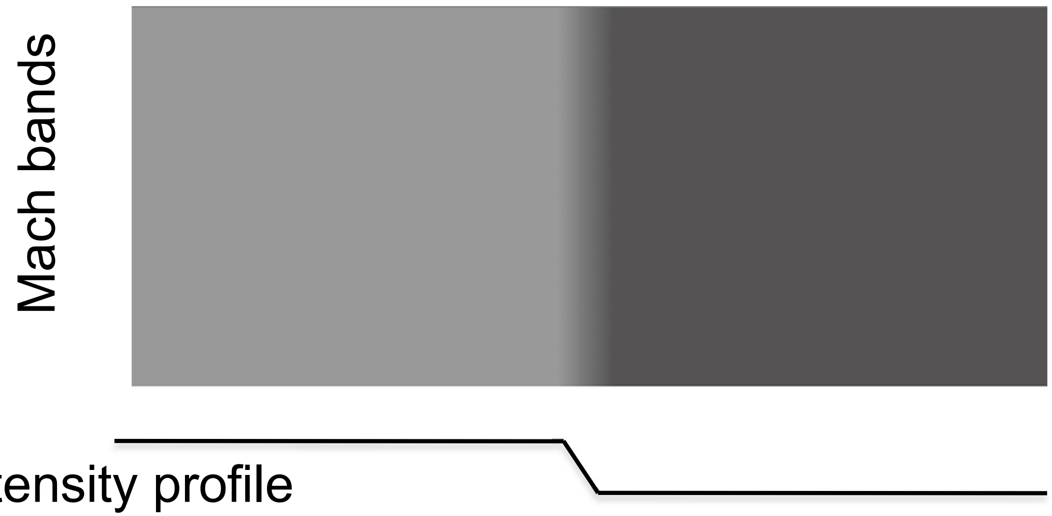
Sample Length:	2		10										10										10									
Channel Membership:	None		Red										Green										Blue									
Bit Number:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RGBAX	R . G . B . A . X																															
Sample Length Notation:	10.10.10.0.2																															

- But why?

Color banding

- If there are not enough bits to represent color
- Looks worse because of the **Mach band** or **Chevreul** illusion
- Dithering (added noise) can reduce banding
  - Printers but also some LCD displays





What is a (computer) image?

- A digital photograph? ("JPEG")
- A snapshot of real-world lighting?

Image

From computing perspective (discrete)

2D array of pixels

- To represent images in memory
- To create image processing software

From mathematical perspective (continuous)

2D function

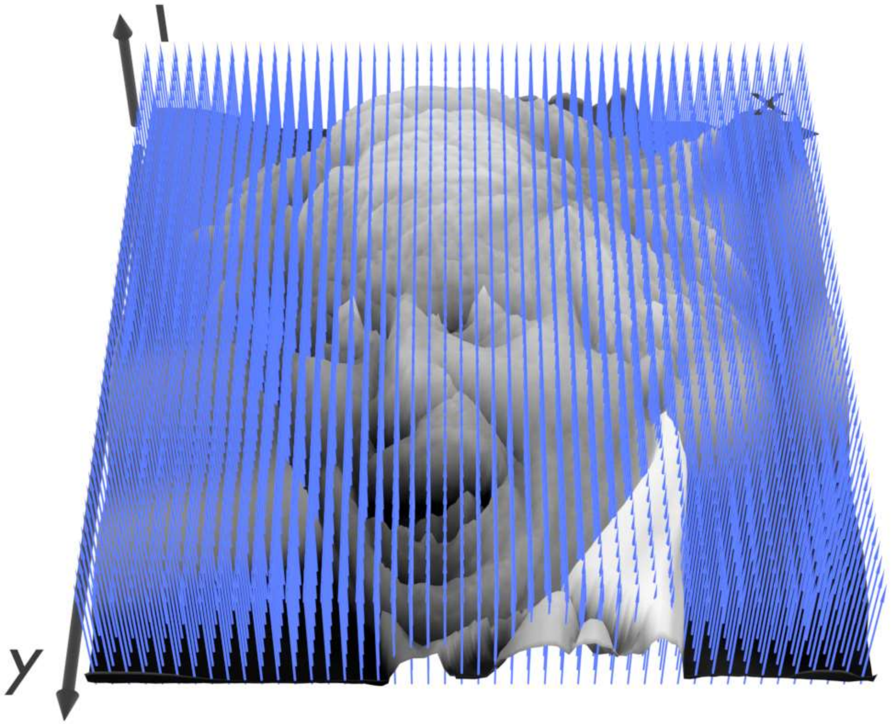
- To express image processing as a mathematical problem
- To develop (and understand) algorithms

Image – 2D function

- Image can be seen as a function I(x,y), that gives intensity value for any given coordinate (x,y)

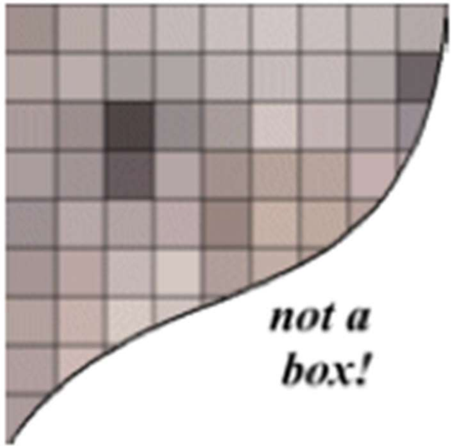
Sampling an image

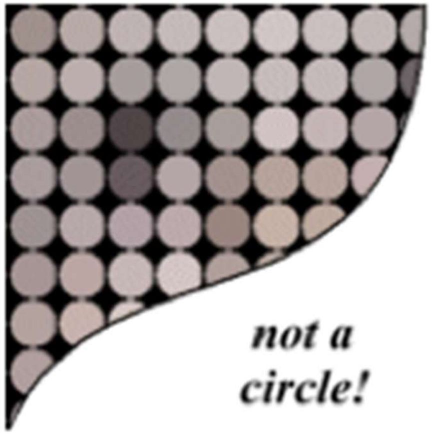
- The image can be sampled on a rectangular sampling grid to yield a set of samples. These samples are pixels.



What is a pixel? (math)

- A pixel is not
  - a box
  - a disk
  - a teeny light
- A pixel is a point
  - it has no dimension
  - it occupies no area
  - it cannot be seen
  - it has coordinates
- A pixel is a **sample**

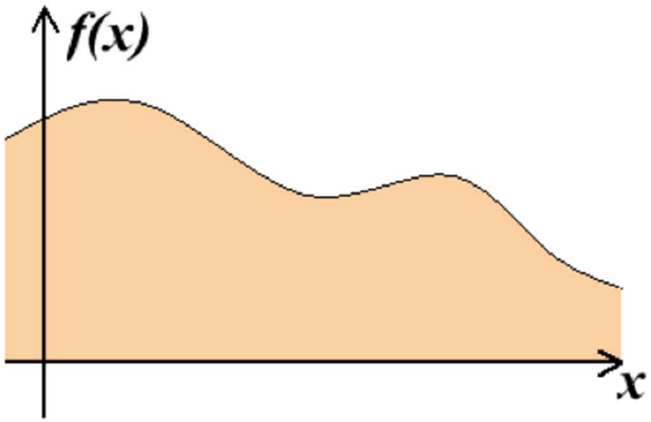


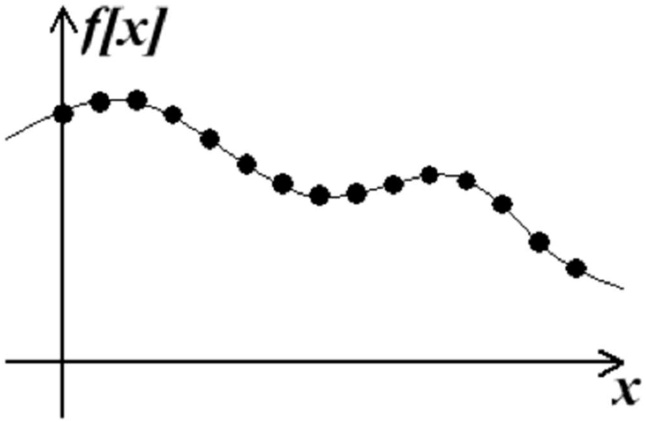


From: http://groups.csail.mit.edu/graphics/classes/6.837/F01/Lecture05/lecture05.pdf

Sampling and quantization

- Physical world is described in terms of continuous quantities
- But computers work only with discrete numbers
- Sampling – process of mapping continuous function to a discrete one
- Quantization – process of mapping continuous variable to a discrete one

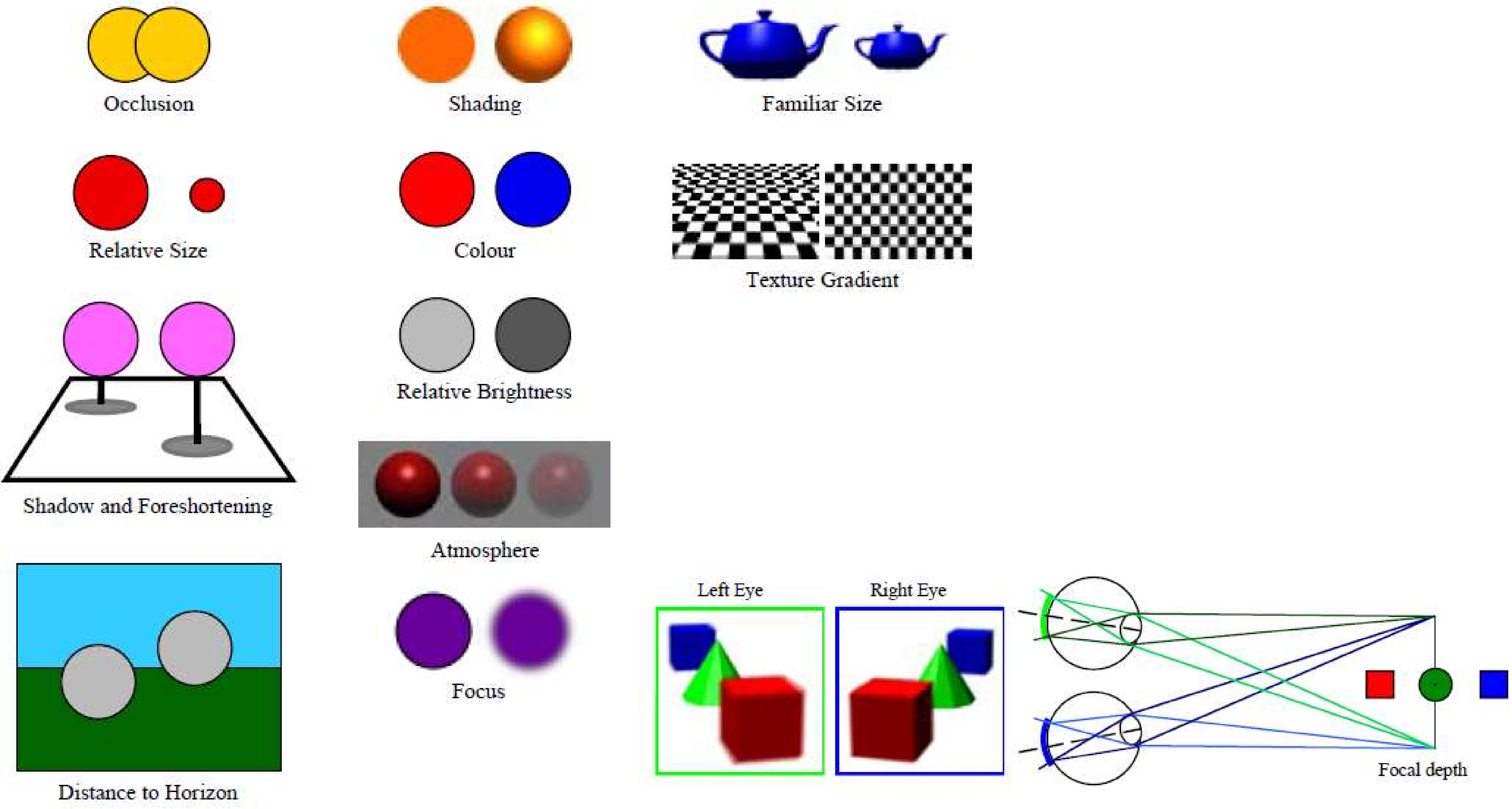




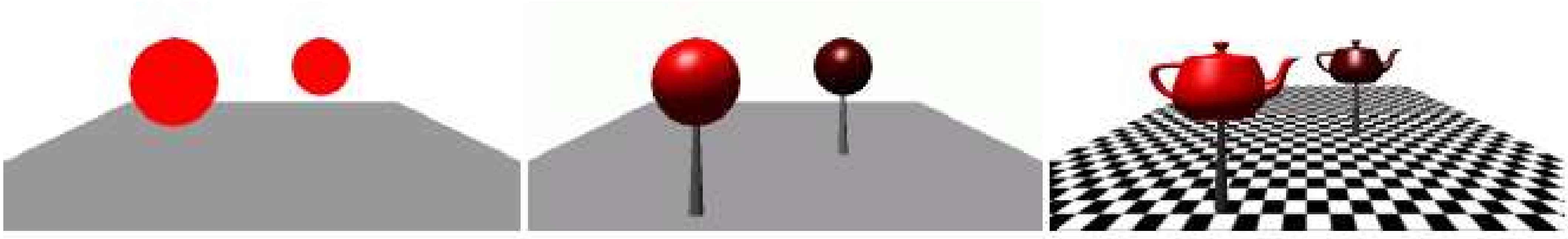
Computer Graphics & Image Processing

- Background
- **Rendering**
  - Perspective
  - Reflection of light from surfaces and shading
  - Geometric models
  - Ray tracing
- Graphics pipeline
- Graphics hardware and modern OpenGL
- Human vision and colour & tone mapping

Depth cues


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

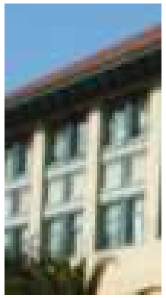
Rendering depth

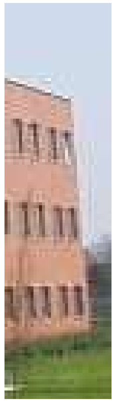


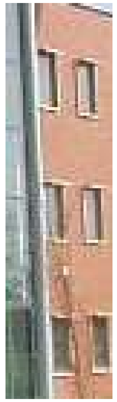
Perspective in photographs

Gates Building – the rounded version
(Stanford)

Gates Building – the rectilinear version
(Cambridge)



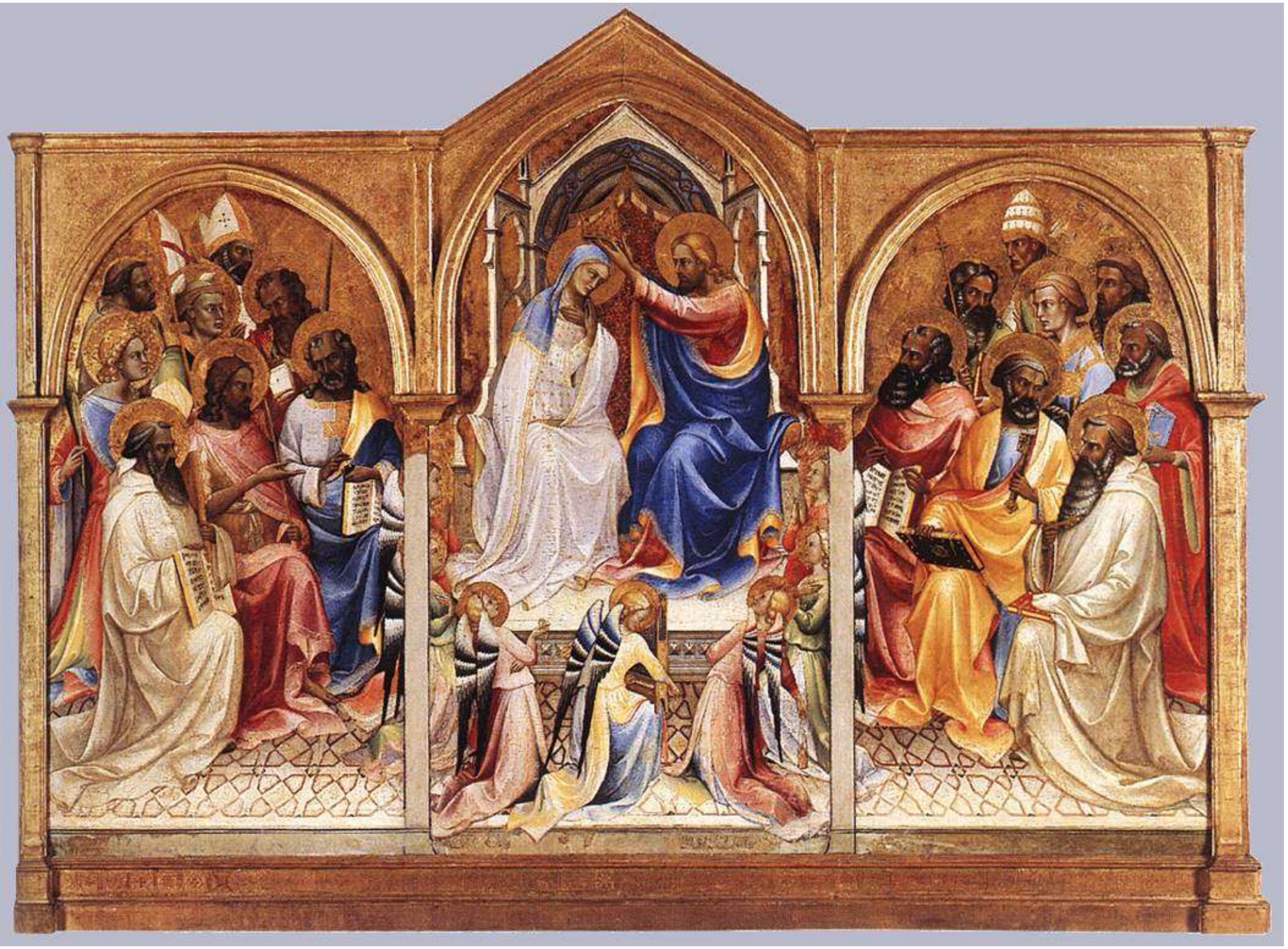




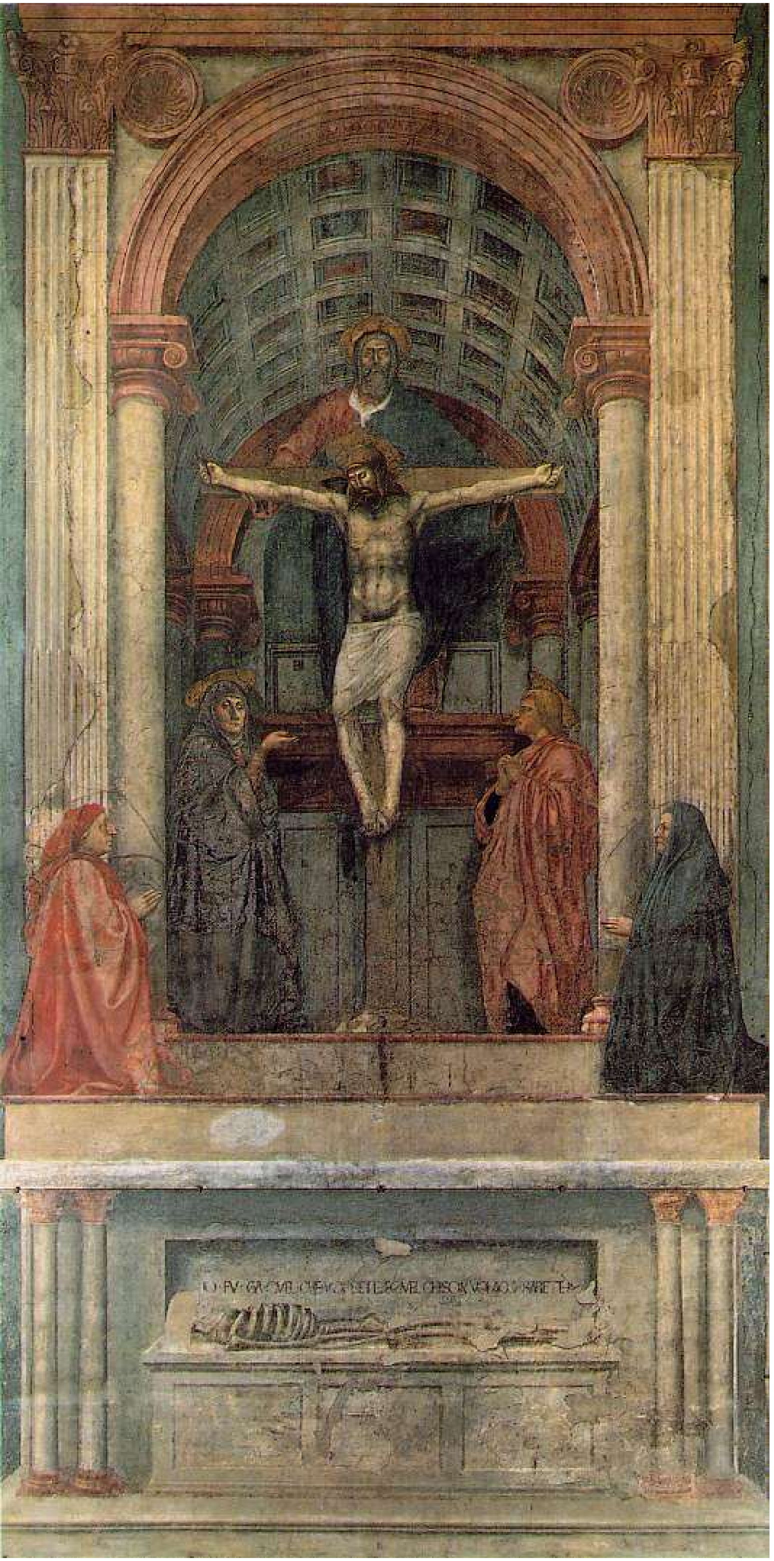
Early perspective

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

Wrong perspective



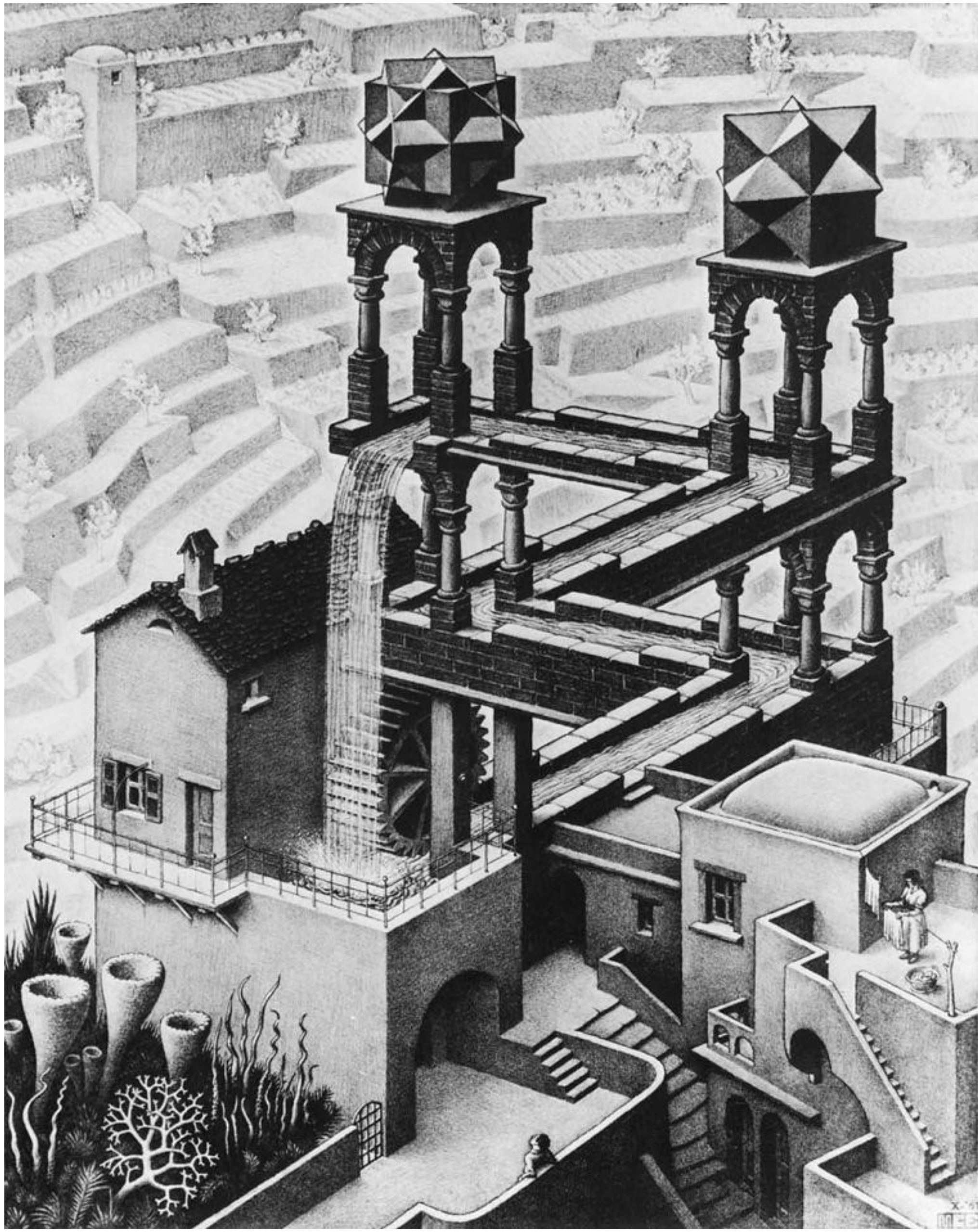
- Adoring saints
- Lorenzo Monaco 1407-09
- National Gallery London



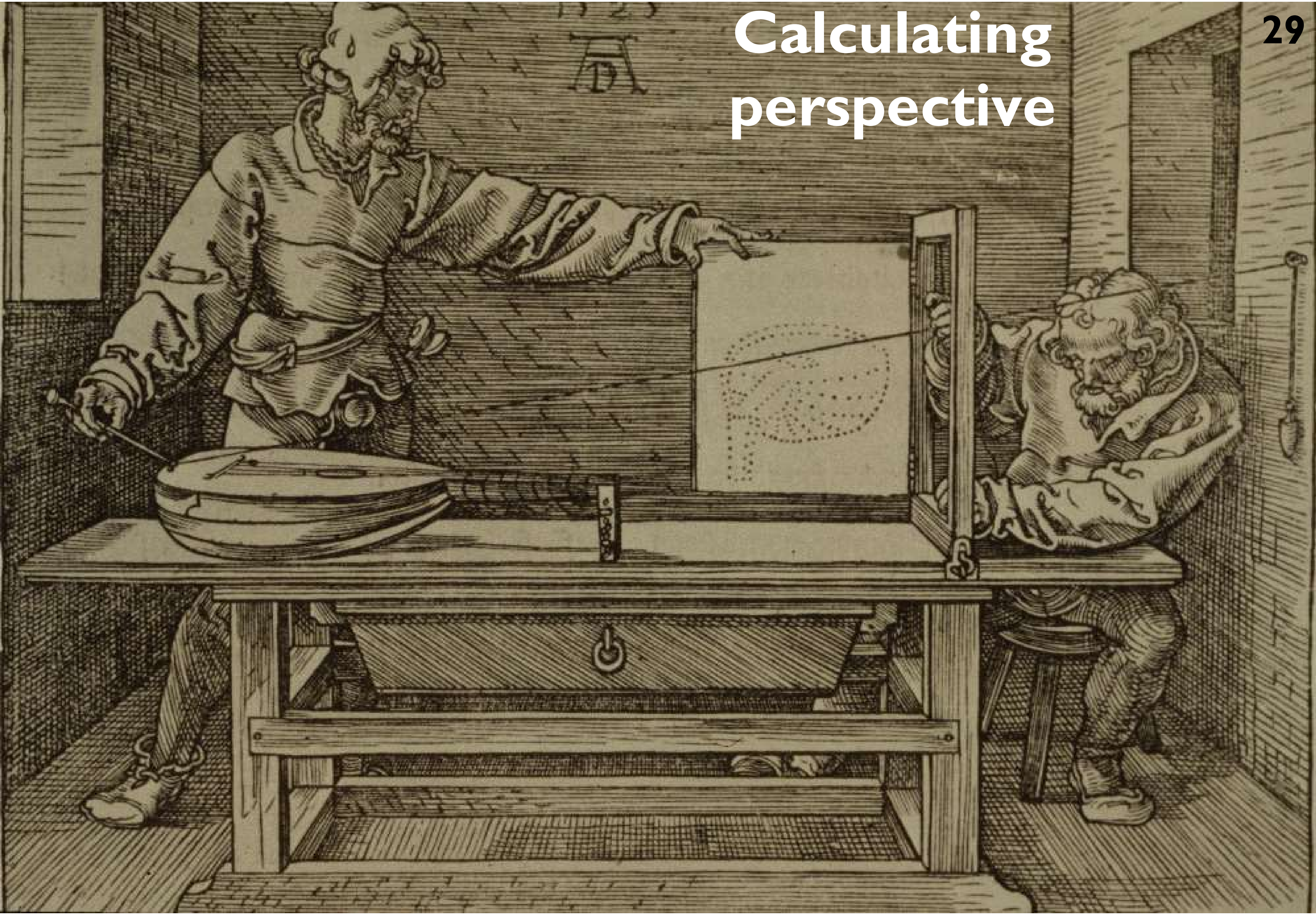
Renaissance perspective

- 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

False perspective

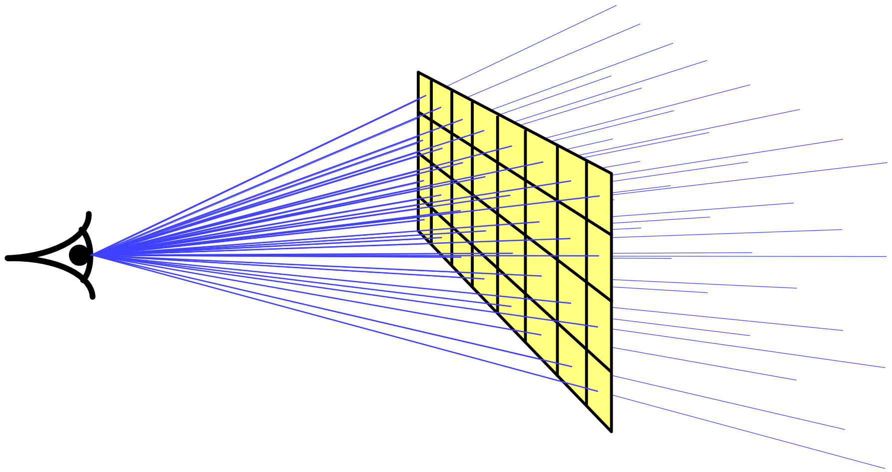


Calculating perspective

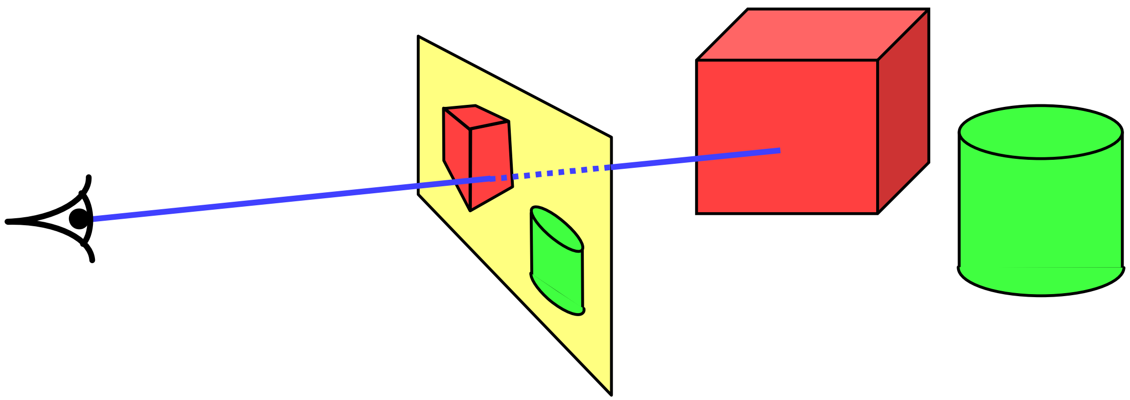


Ray tracing

- 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



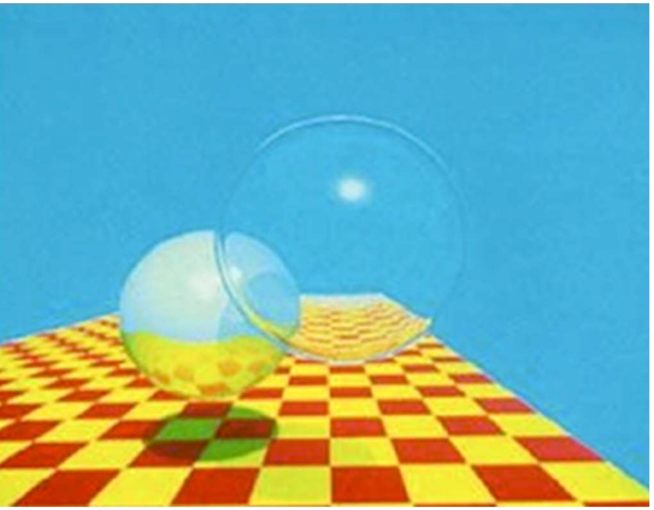
shoot a ray through each pixel



whatever the ray hits determines the colour of that pixel

[FCG 4/4]

Ray tracing: examples



ray tracing easily handles reflection, refraction, shadows and blur

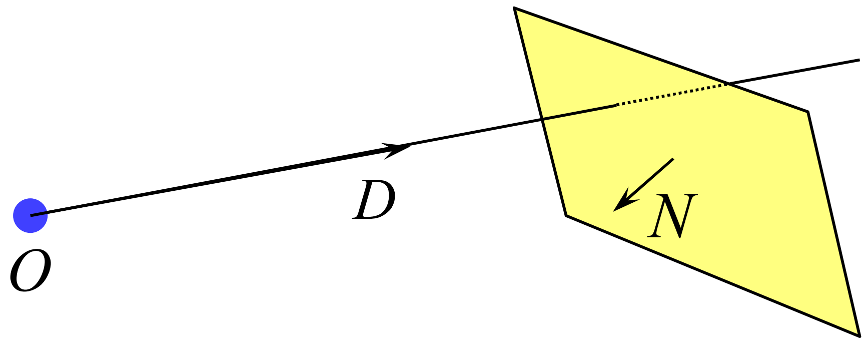
ray tracing is computationally expensive

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 1

- plane



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

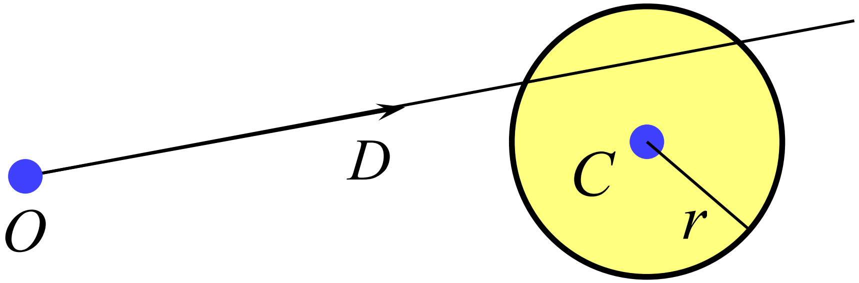
$$\text{plane: } P \cdot N + d = 0$$

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

- 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)

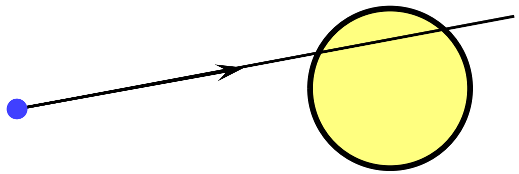
Intersection of a ray with an object 2

- sphere

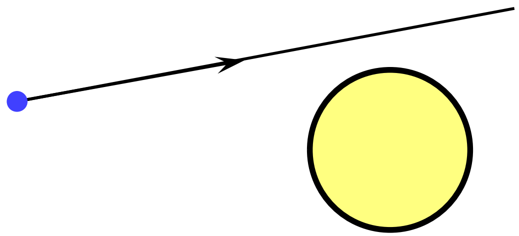


ray: $P = O + sD,\ s \geq 0$

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



d real



d imaginary

$$a = D \cdot D$$

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

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

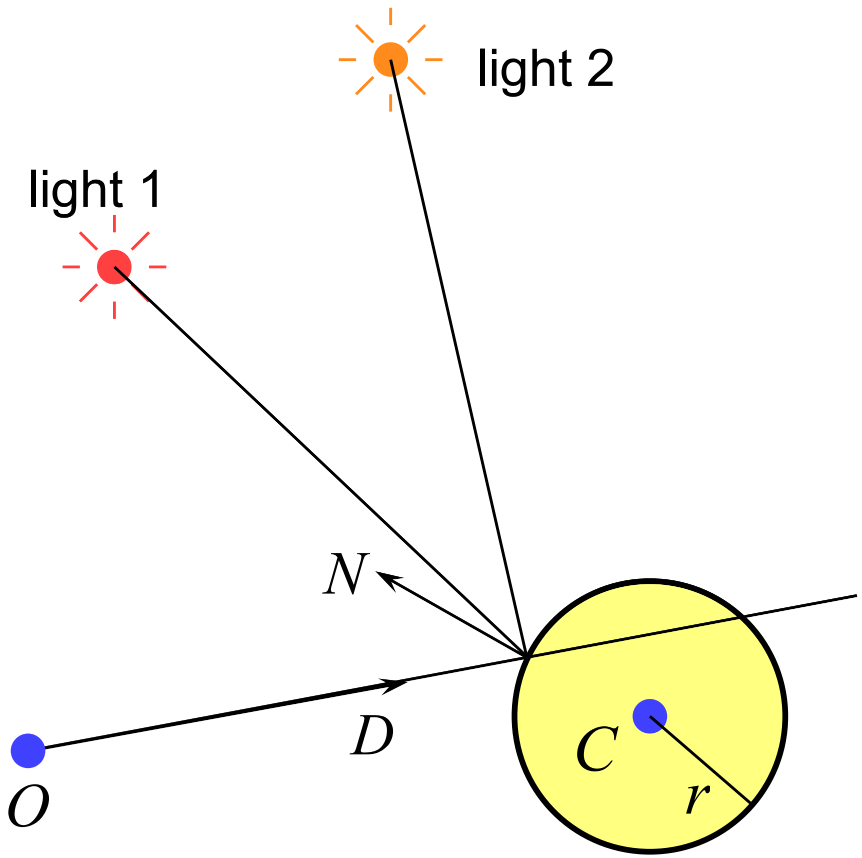
$$d = \sqrt{b^2 - 4ac}$$

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

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

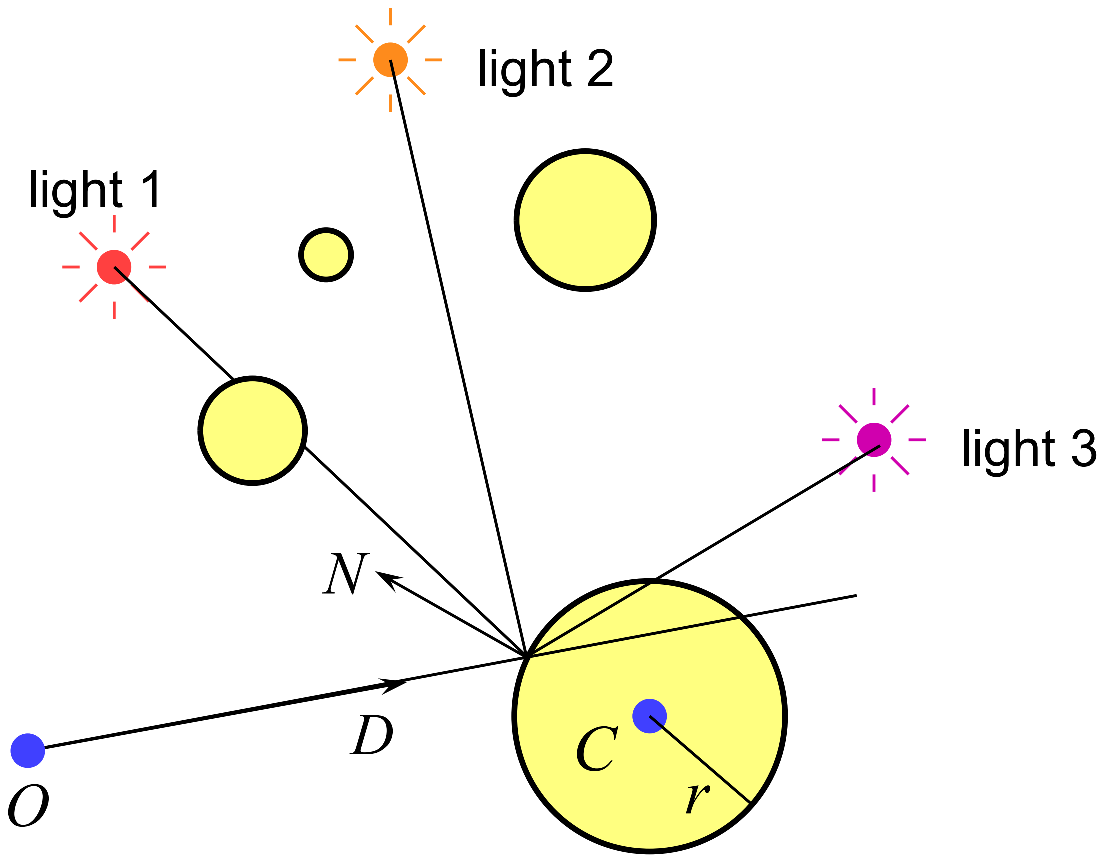
- cylinder, cone, torus
  - all similar to sphere
  - try them as an exercise

Ray tracing: shading



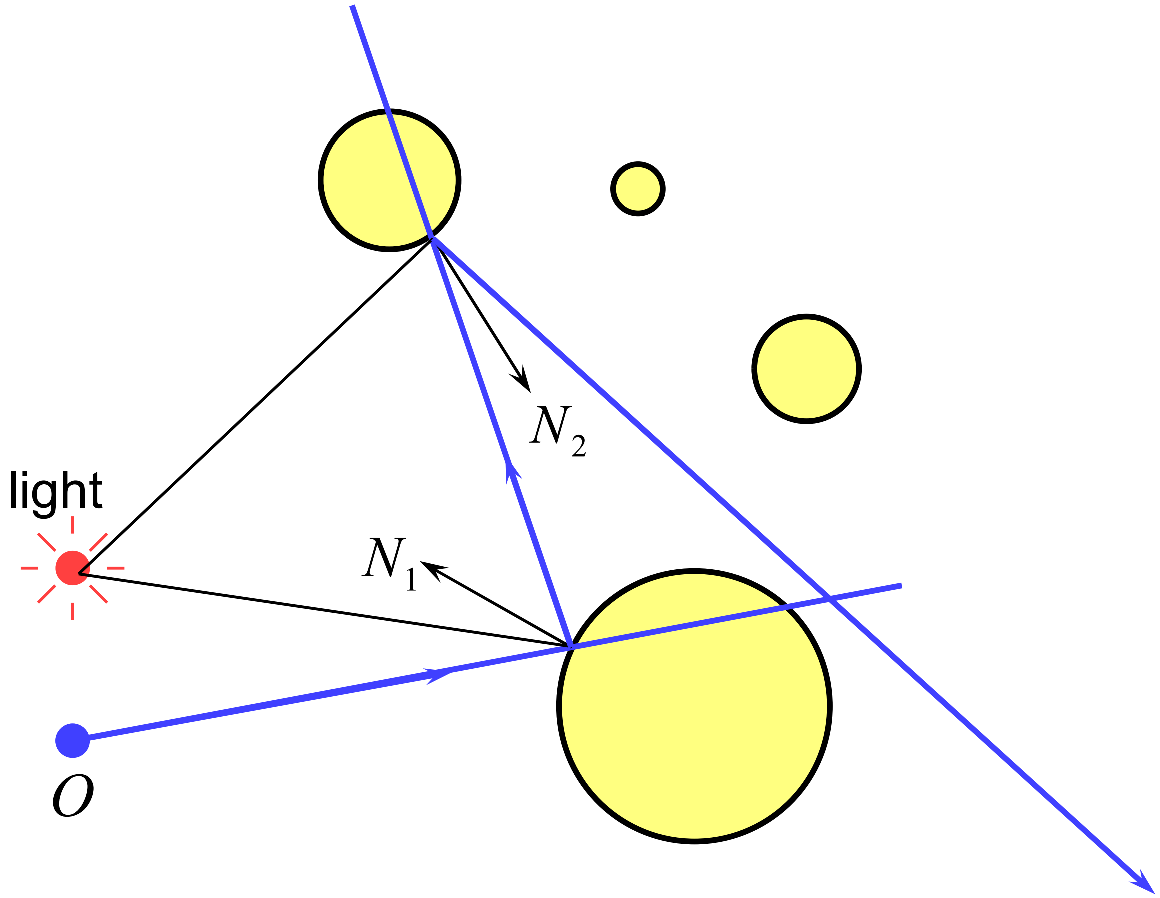
- once you have the intersection of a ray with the nearest object you can also:
  - calculate the normal to the object at that intersection point
  - shoot rays from that point to all of the light sources, and calculate the diffuse and specular reflections off the object at that point
    - this (plus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

Ray tracing: shadows



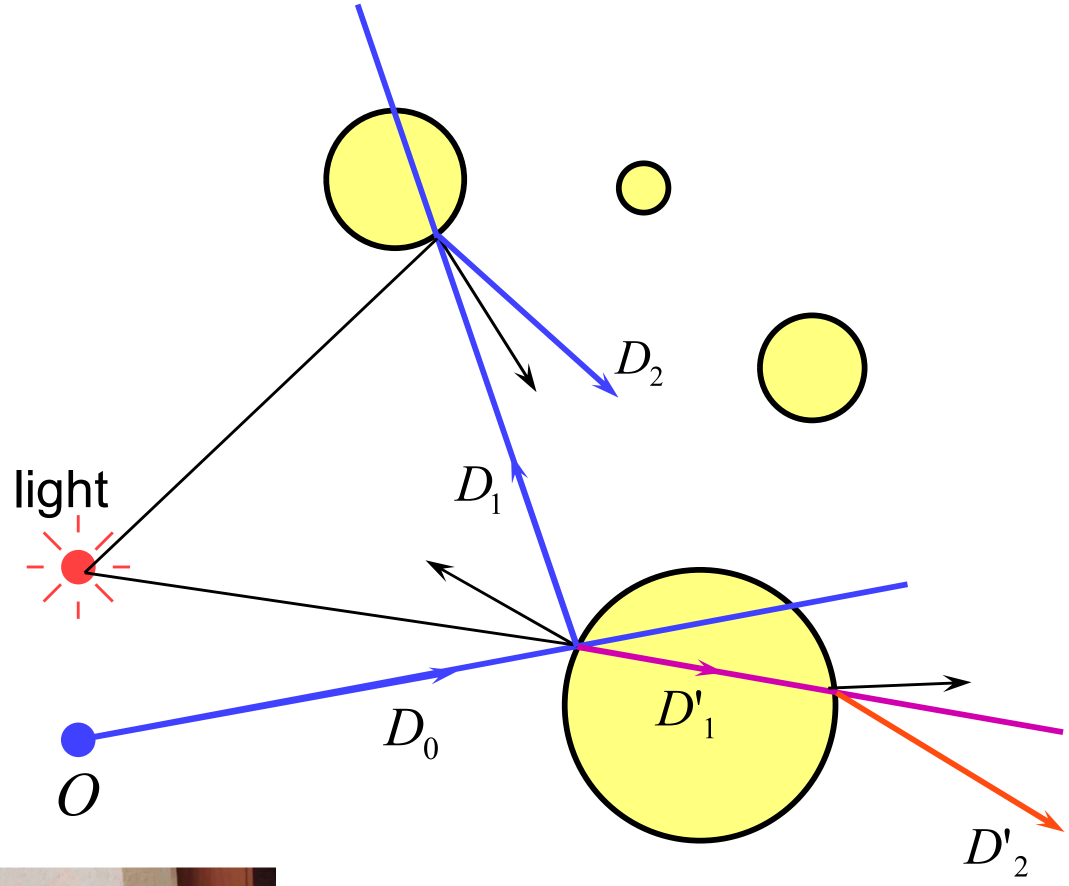
- because you are tracing rays from the intersection point to the light, you can check whether another object is between the intersection and the light and is hence casting a shadow
  - also need to watch for self-shadowing

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




Example of a refraction

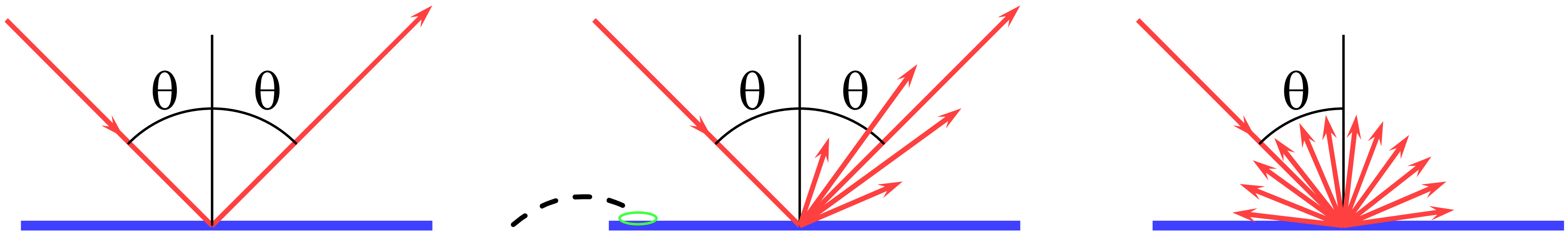
- objects can be totally or partially transparent
  - this allows objects behind the current one to be seen through it
- transparent objects can have refractive indices
  - bending the rays as they pass through the objects
- transparency + reflection means that a ray can split into two parts

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

[FCG 4.5-4.8/5]

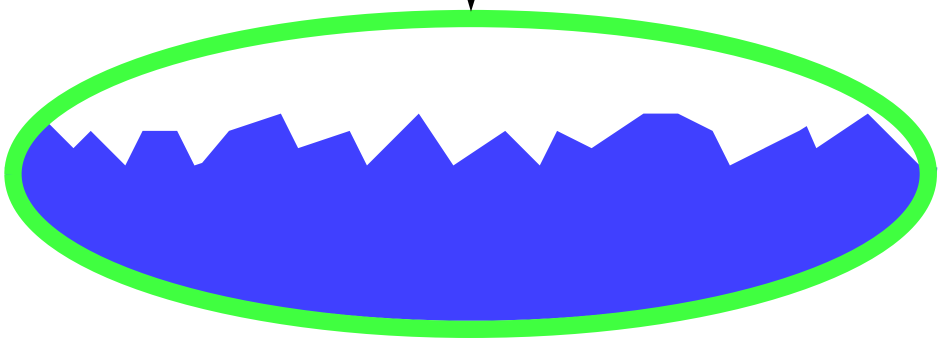
How do surfaces reflect light?



perfect specular reflection (mirror)

imperfect specular reflection

diffuse reflection (Lambertian reflection)



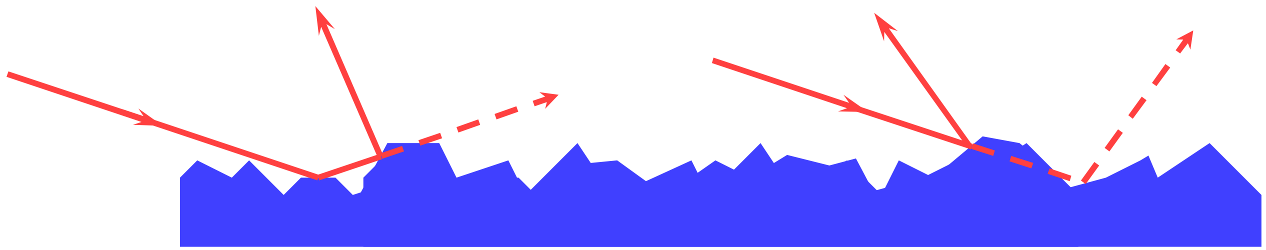
the surface of a specular reflector is facetted, each facet reflects perfectly but in a slightly different direction to the other facets

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

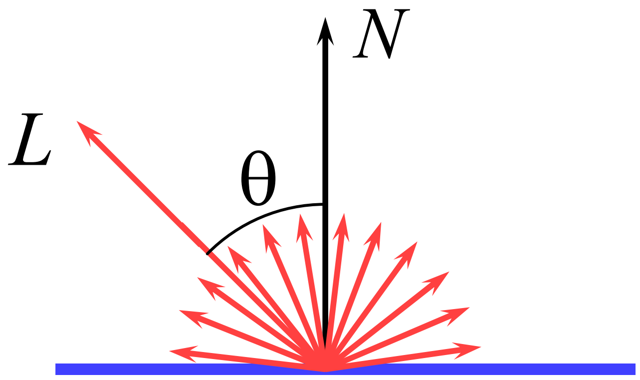


- plastics are good examples of surfaces with:
  - specular reflection in the light's colour
  - diffuse reflection in the plastic's colour

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

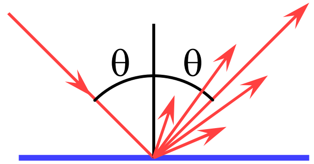
I is the intensity of the light reflected by the surface

use this equation to calculate the colour of a pixel

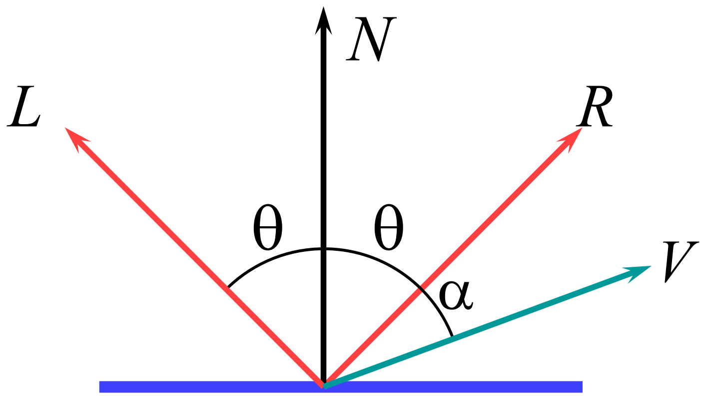
Diffuse shading: comments

- 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
  - Bidirectional Reflectance Distribution Function

Imperfect specular reflection



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



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

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

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

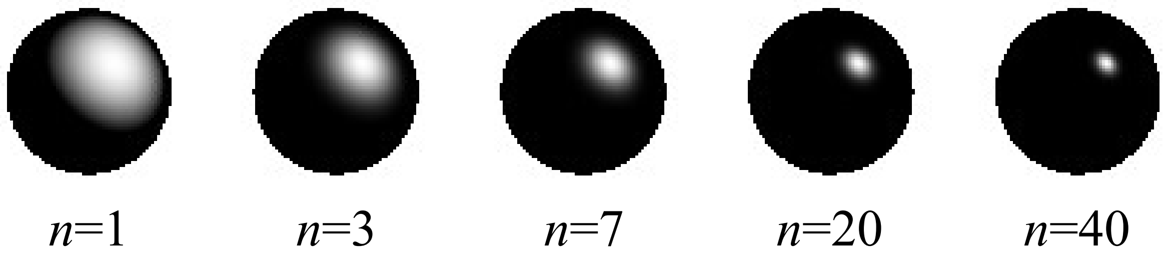
V is a normalised vector pointing at the viewer

I_l is the intensity of the light source

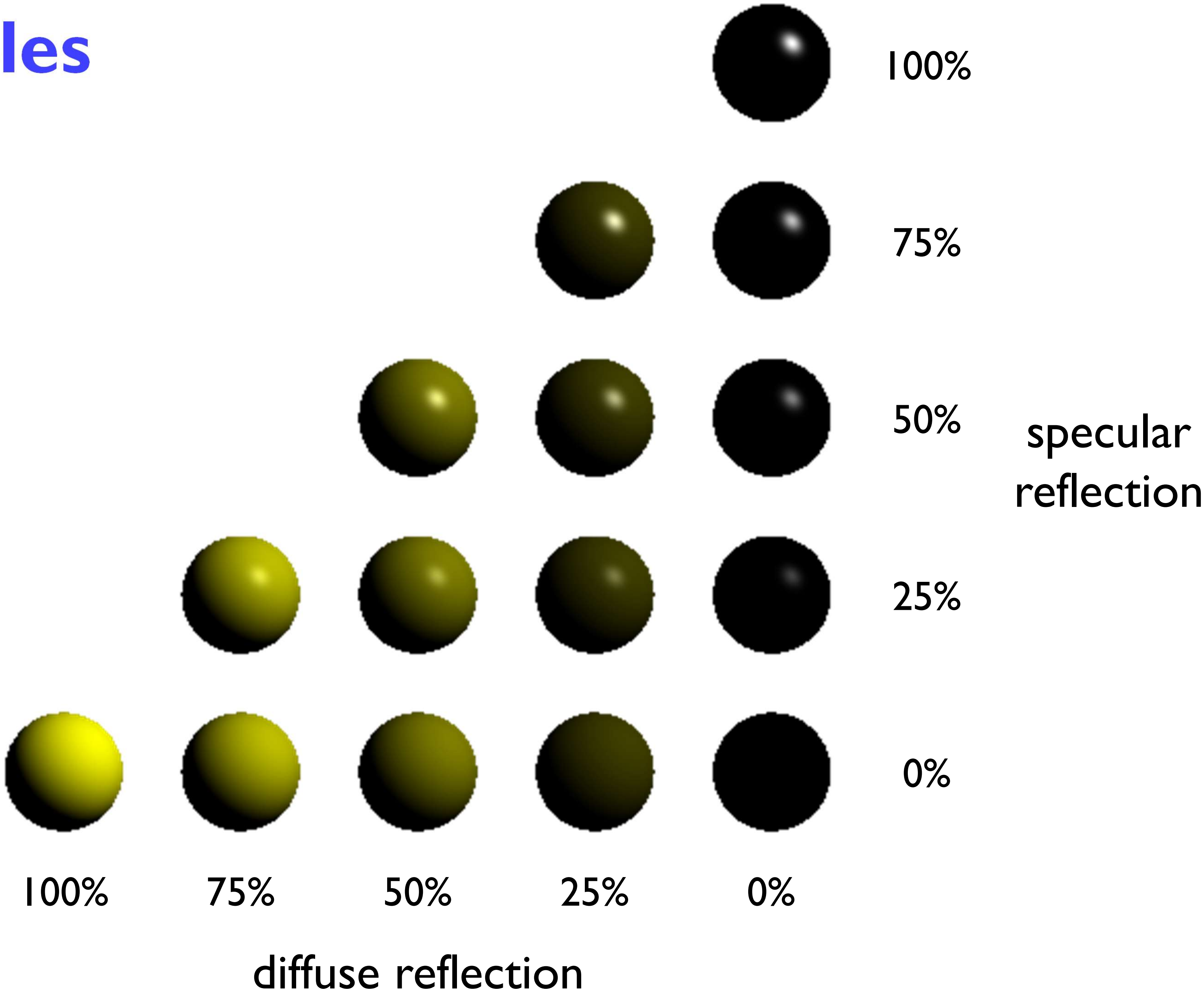
k_s is the proportion of light which is specularly reflected by the surface

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

I is the intensity of the specularly reflected light



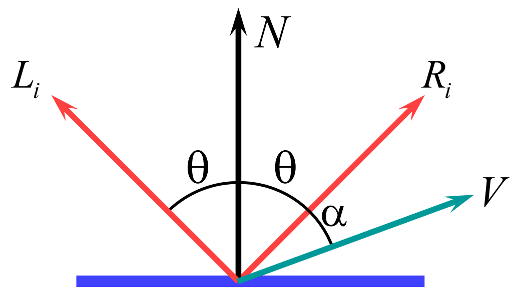
Examples


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

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



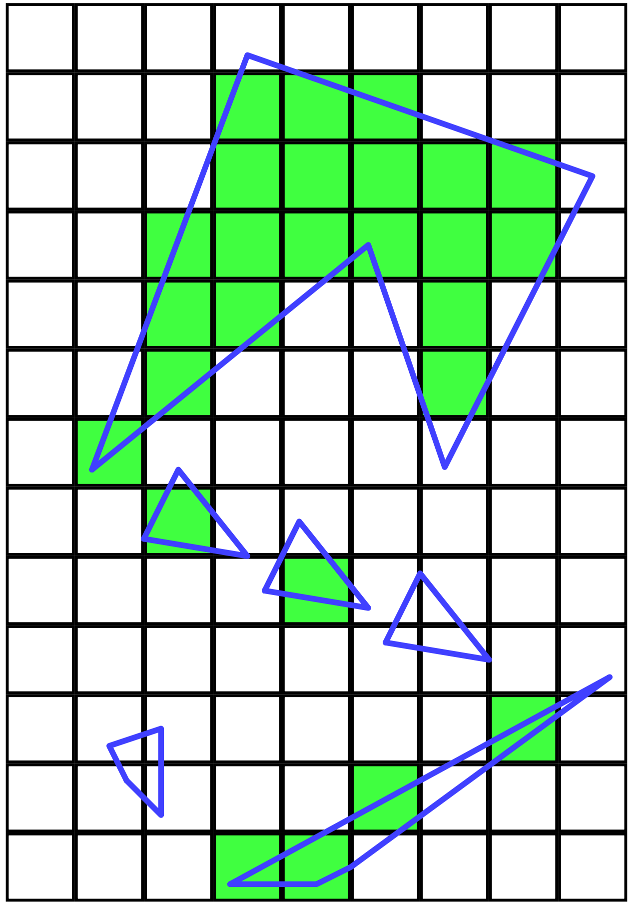
  - the more lights there are in the scene, the longer this calculation will take

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

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

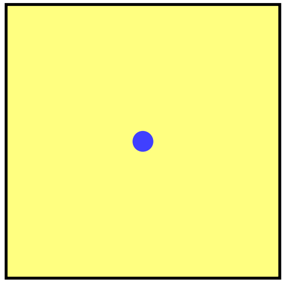


Anti-aliasing

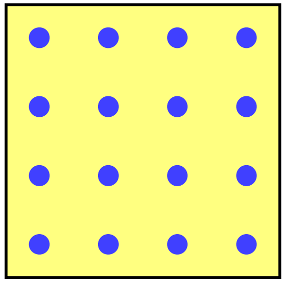
- these artefacts (and others) are jointly known as aliasing
- methods of ameliorating the effects of aliasing are known as *anti-aliasing*
  - in signal processing *aliasing* is a precisely defined technical term for a particular kind of artefact
  - in computer graphics its meaning has expanded to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

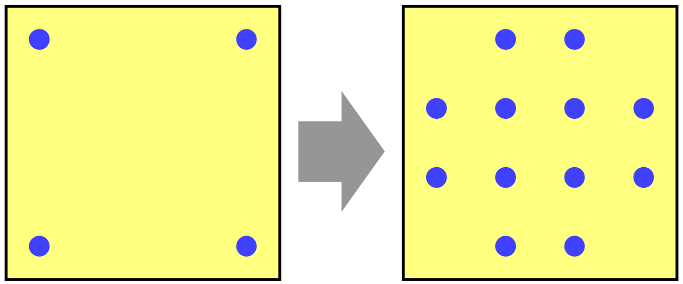
- single point
  - shoot a single ray through the pixel's centre



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

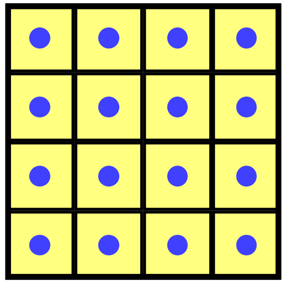


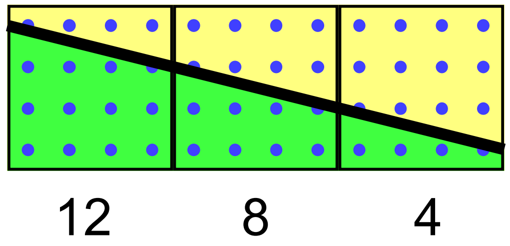
- adaptive super-sampling
  - shoot a few rays through the pixel, check the variance of the resulting values, if similar enough stop, otherwise shoot some more rays

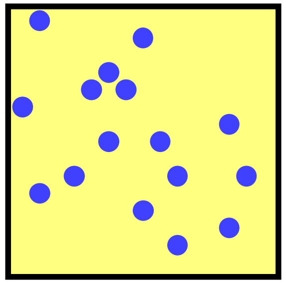


Types of super-sampling 1

- regular grid
  - divide the pixel into a number of sub-pixels and shoot a ray through the centre of each
  - problem: can still lead to noticable aliasing unless a very high resolution sub-pixel grid is used
- random
  - shoot N rays at random points in the pixel
  - replaces aliasing artefacts with noise artefacts
    - the eye is far less sensitive to noise than to aliasing

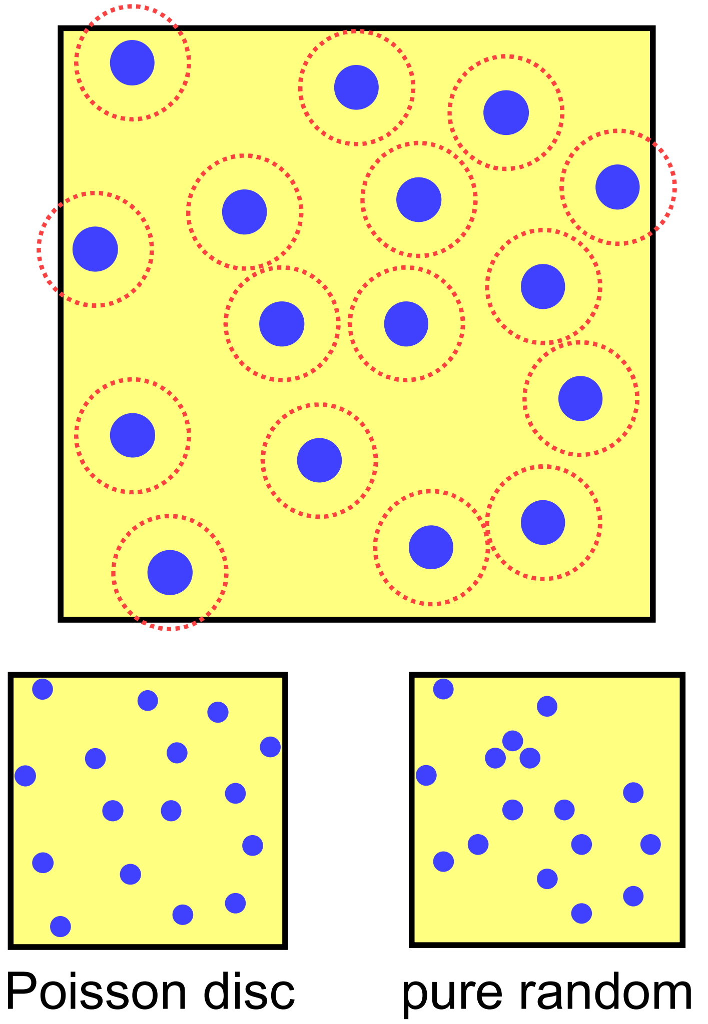






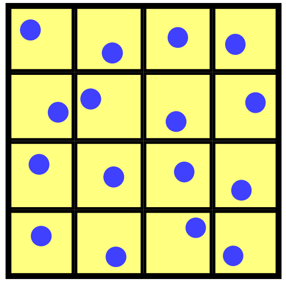
Types of super-sampling 2

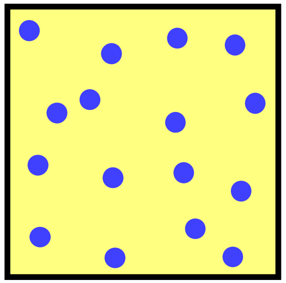
- Poisson disc
  - shoot N rays at random points in the pixel with the proviso that no two rays shall pass through the pixel closer than ε to one another
  - for N rays this produces a better looking image than pure random sampling
  - very hard to implement properly



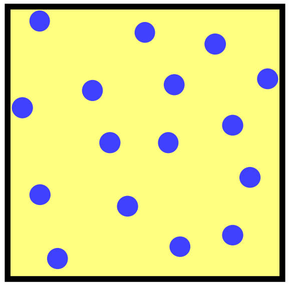
Types of super-sampling 3

- 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

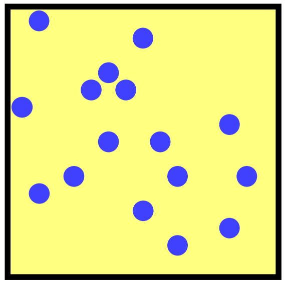




jittered



Poisson disc



pure random

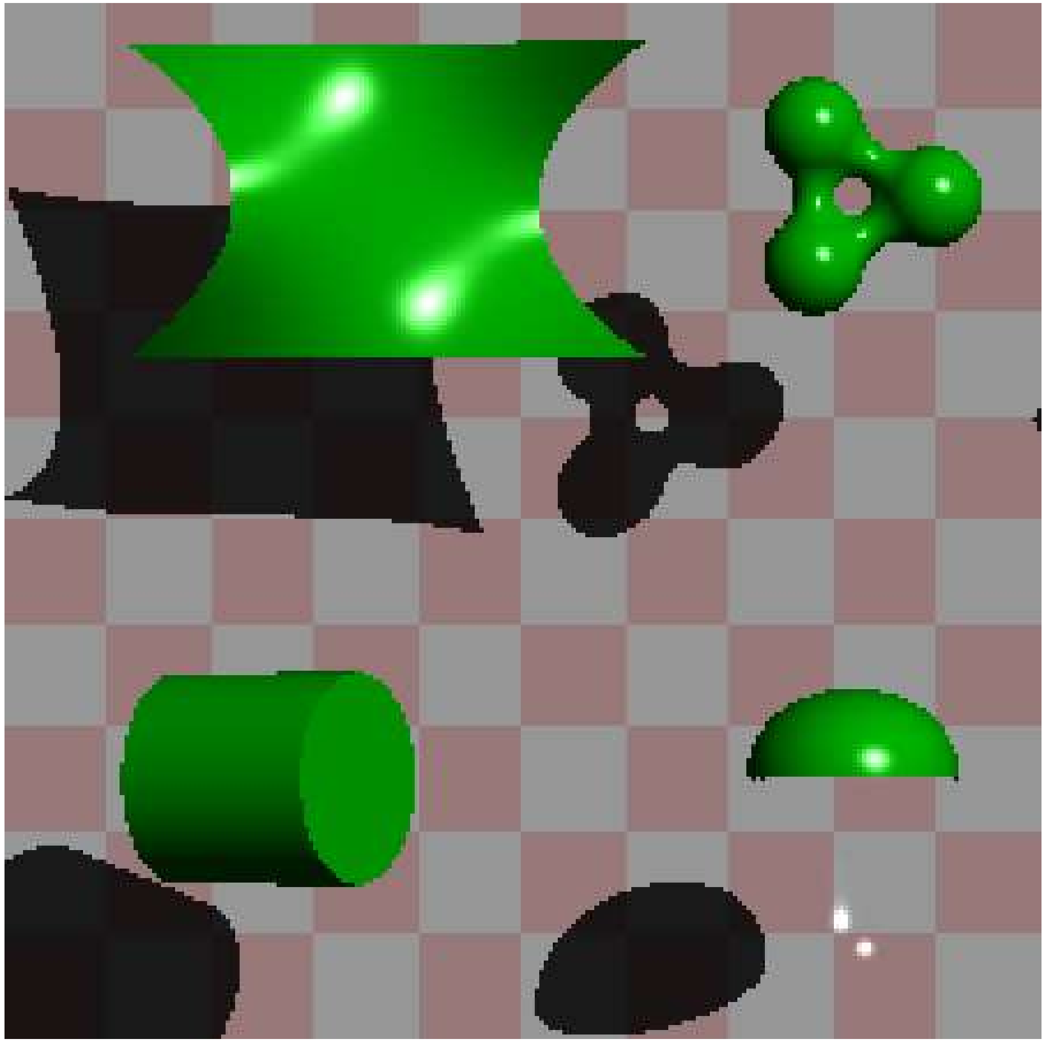
More reasons for wanting to take multiple samples per pixel

- 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

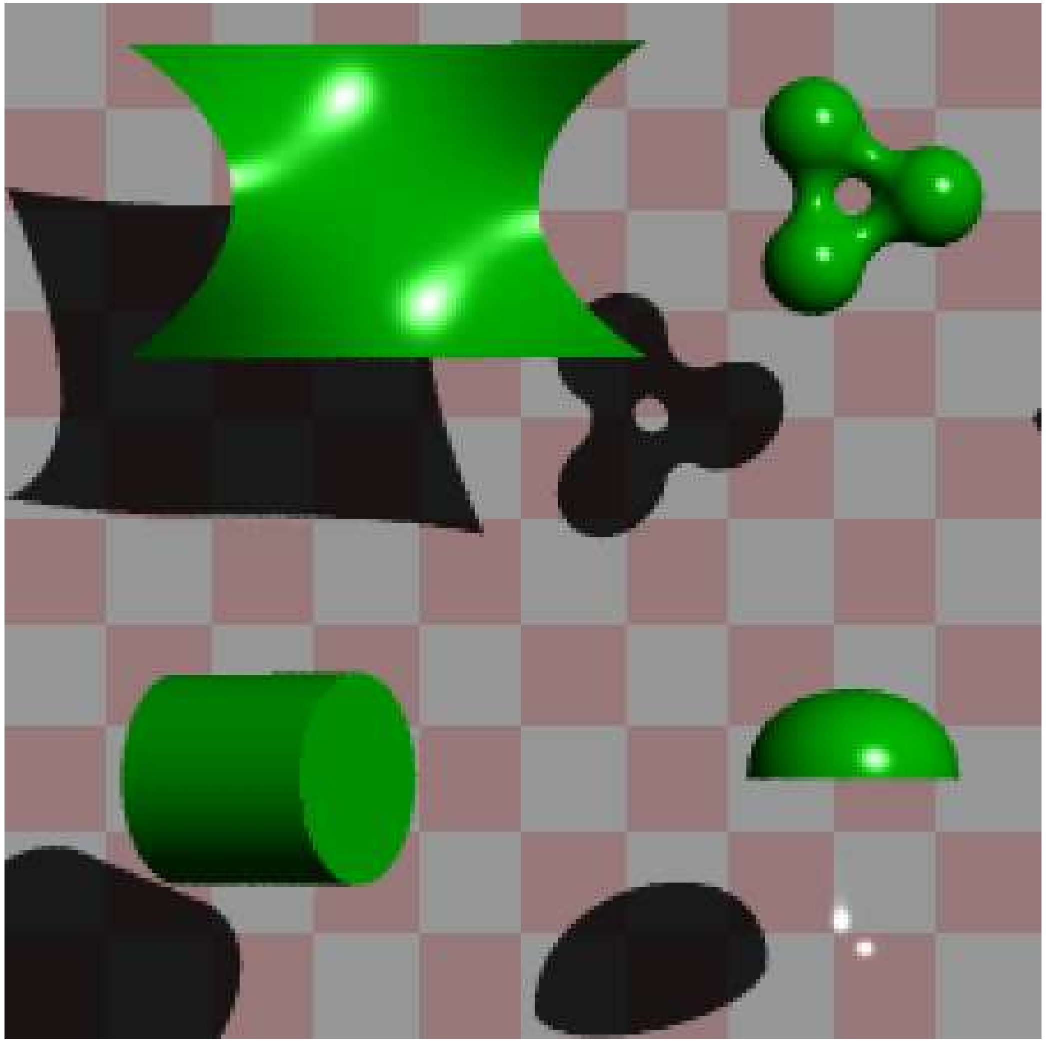
Examples of distributed ray tracing

- 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



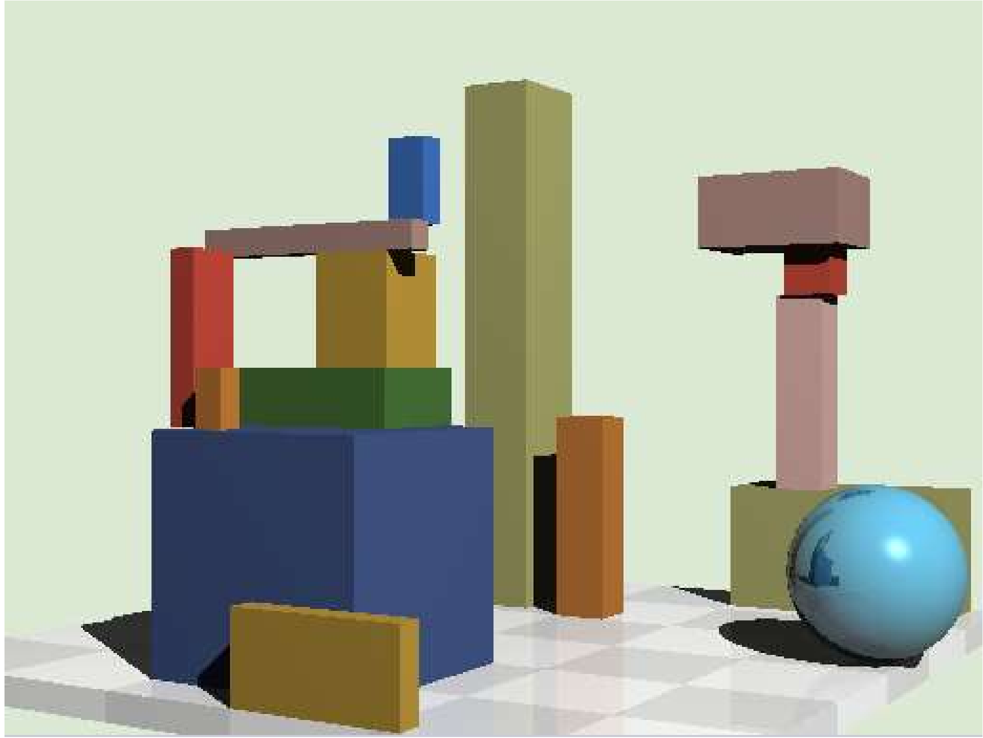
multiple samples per pixel

Area *vs* point light source

an area light source produces soft shadows

a point light source produces hard shadows

Finite aperture



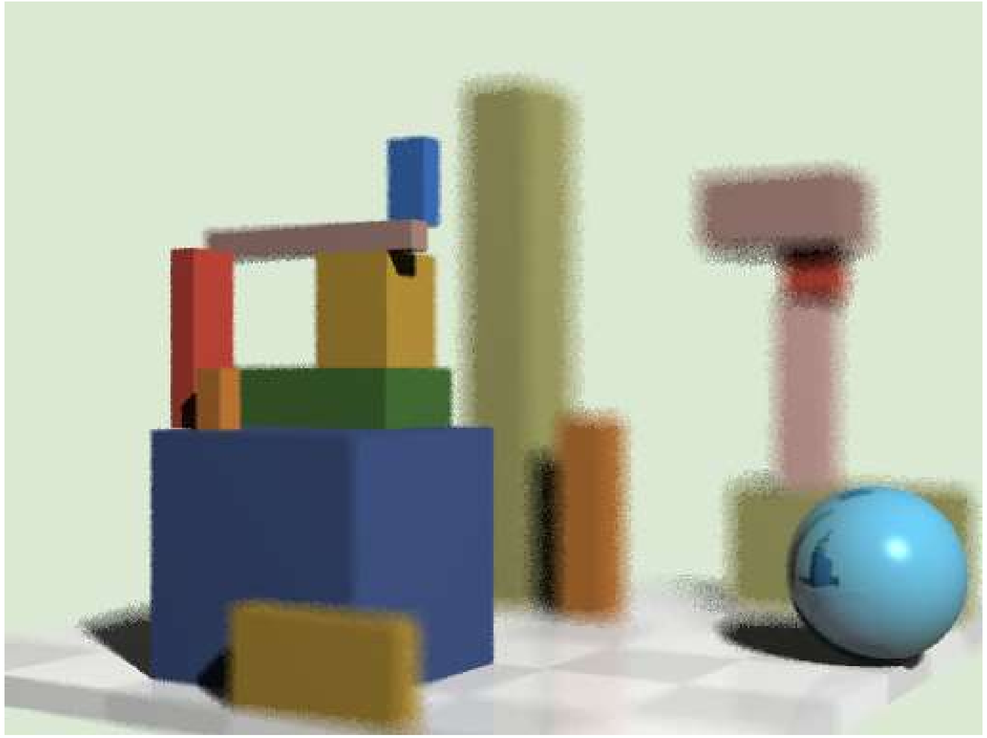
left, a pinhole camera

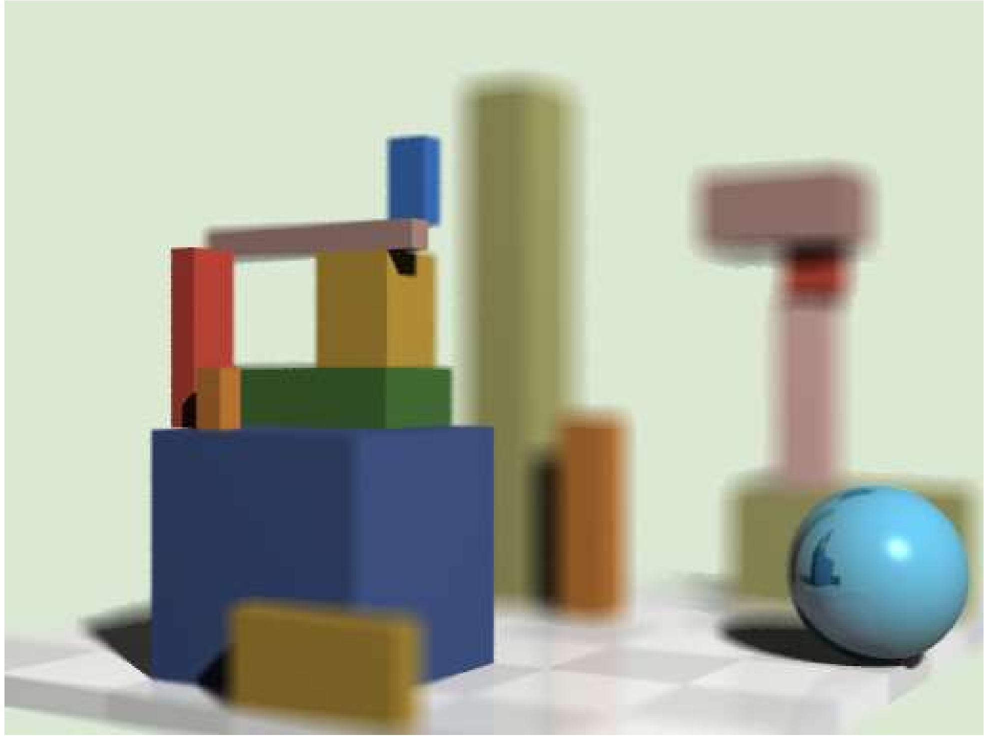
below, a finite aperture camera

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





Introduction to Computer Graphics

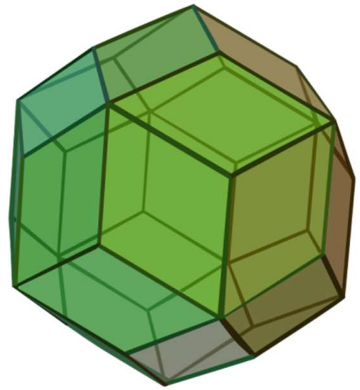
- Background
- Rendering
- **Graphics pipeline**
  - Polygonal mesh models
  - Transformations using matrices in 2D and 3D
  - Homogeneous coordinates
  - Projection: orthographic and perspective
- Rasterization
- Graphics hardware and modern OpenGL
- Human vision, colour and tone mapping

Unfortunately…

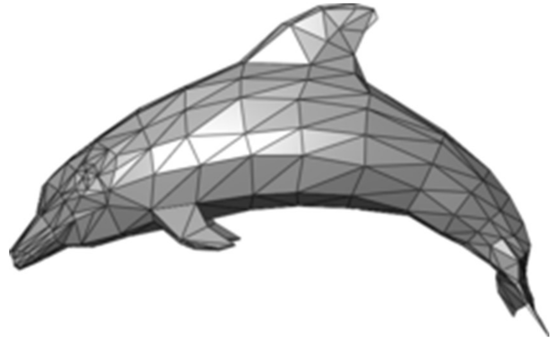
- Ray tracing is computationally expensive
  - used for super-high visual quality
- Video games and user interfaces need something faster
- Most real-time applications rely on **rasterization**
  - Model surfaces as polyhedra – meshes of polygons
  - Use composition to build scenes
  - Apply perspective transformation and project into plane of screen
  - Work out which surface was closest
  - Fill pixels with colour of nearest visible polygon
- Graphics cards have hardware to support this
- Ray tracing starts to appear in real-time rendering
  - The new generations of GPUs offer accelerated ray-tracing
  - But it still not as efficient as rasterization

Three-dimensional objects

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



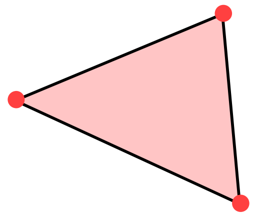
- Polygonal meshes
  - open or closed

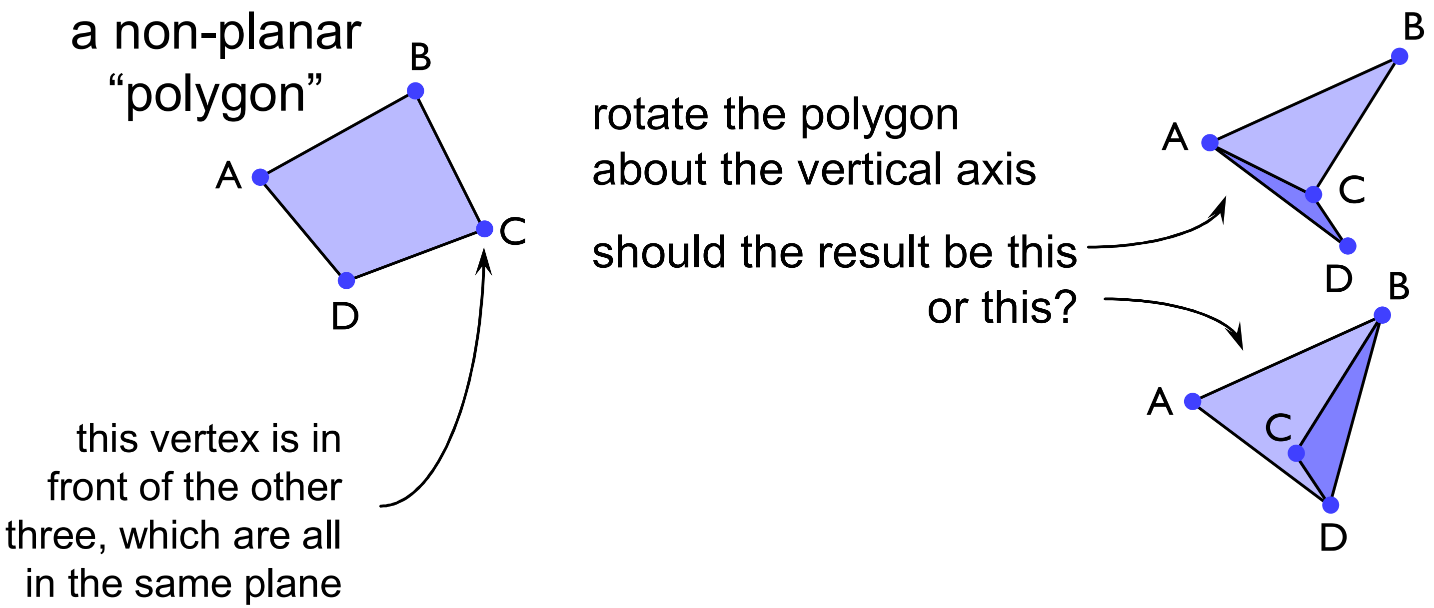


- Curved surfaces
  - must be converted to polygons to be drawn

Surfaces in 3D: polygons

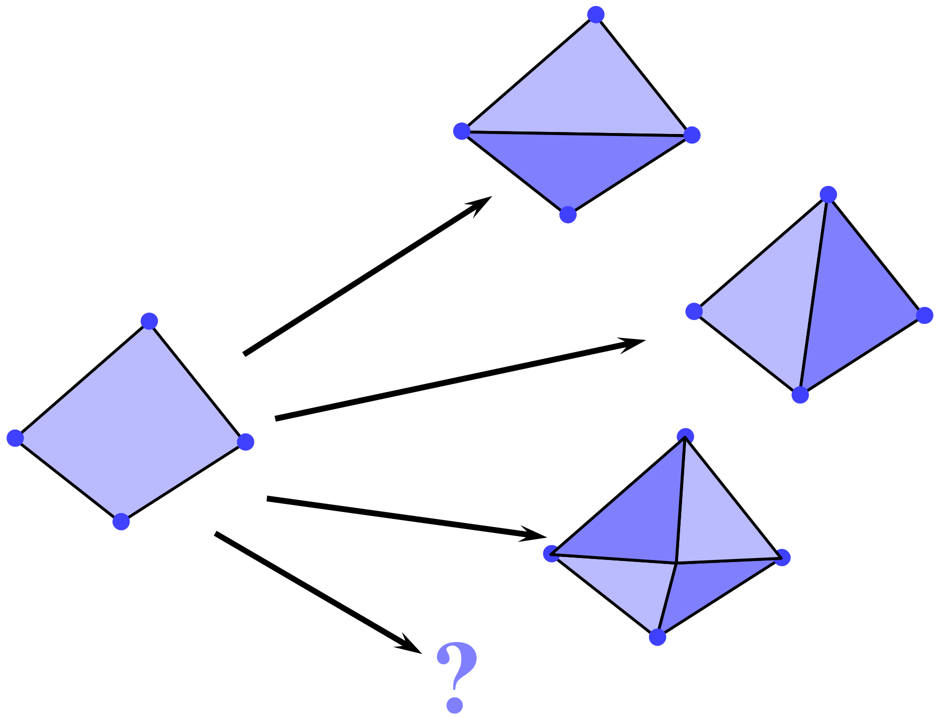
- Easier to consider planar polygons
  - 3 vertices (triangle) must be planar
  - > 3 vertices, not necessarily planar





Splitting polygons into triangles

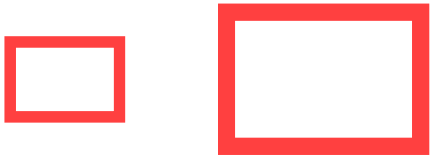
- Most Graphics Processing Units (GPUs) are optimised to draw triangles
- Split polygons with more than three vertices into triangles



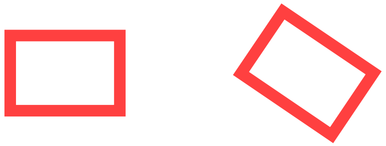
which is preferable?

2D transformations

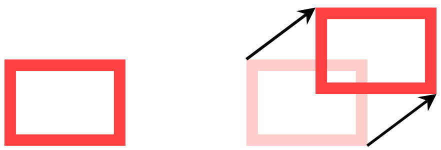
- scale



- 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

[FCG 6/7]

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
- $\boldsymbol{w=0}$ represents a point at infinity
- usually take the inverse transform to be:

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

- The symbol $\equiv$ means equivalent

[FCG 6.3/7.3]

Matrices in homogeneous co-ordinates

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

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

- **rotate**
  - **about origin, angle θ**

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

- **do nothing**
  - **identity**

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

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

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

Translation by matrix algebra

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

In homogeneous coordinates

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

In conventional coordinates

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

Concatenating transformations

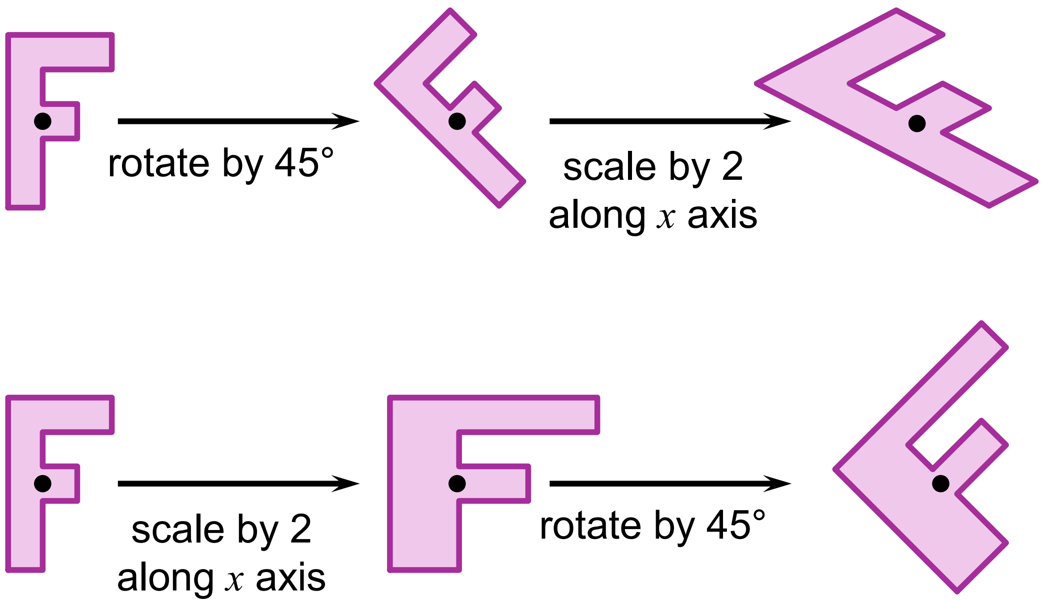
- often necessary to perform more than one transformation on the same object
- can concatenate transformations by multiplying their matrices
  e.g. a shear followed by a scaling:

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

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

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

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

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

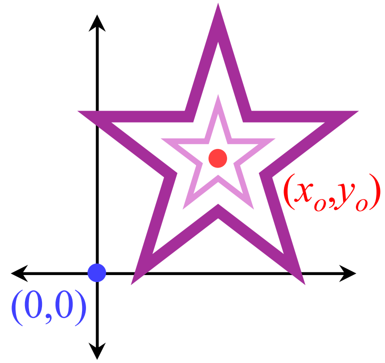
scale then rotate

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

rotate

Scaling about an arbitrary point

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



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

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

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

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

Exercise: show how to perform rotation about an arbitrary point

3D transformations

- 3D homogeneous co-ordinates

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

- 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

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

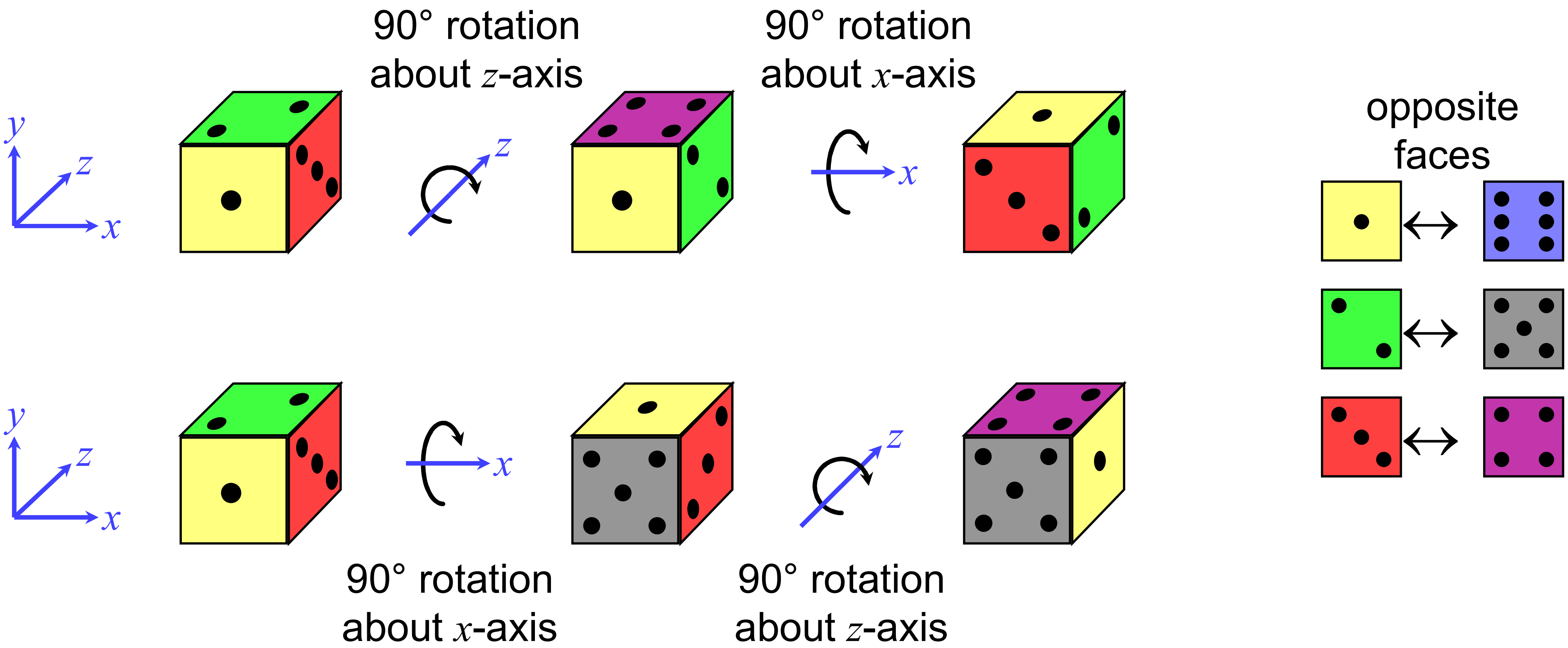
rotation about *z*-axis

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

rotation about *y*-axis

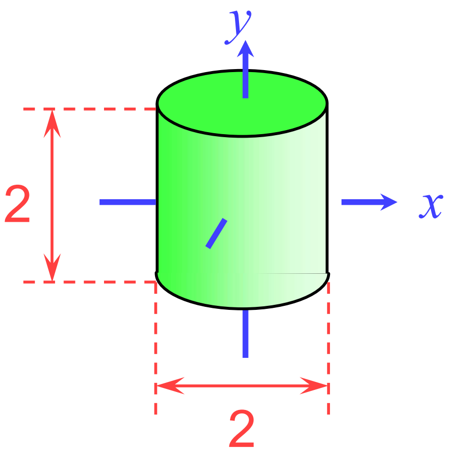
$$\begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

3D transformations are not commutative

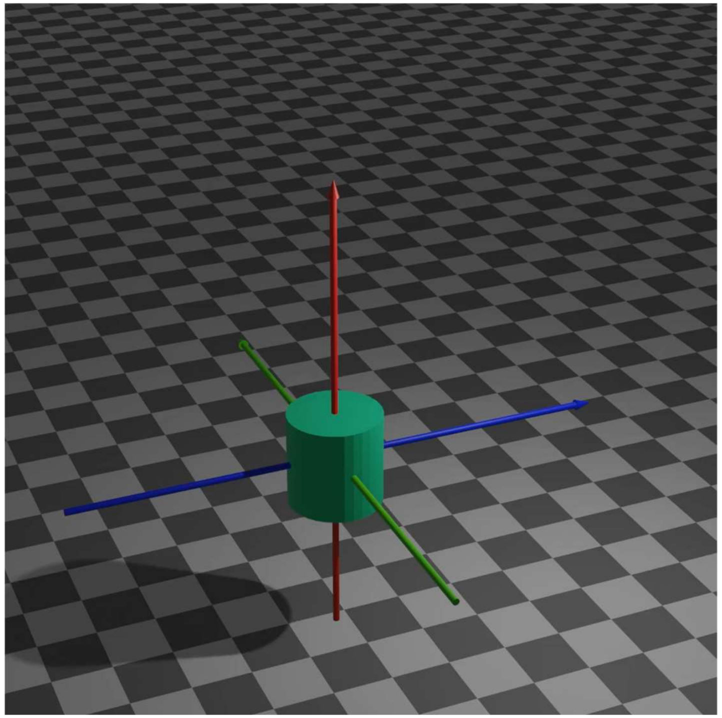


Model transformation 1

- 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**

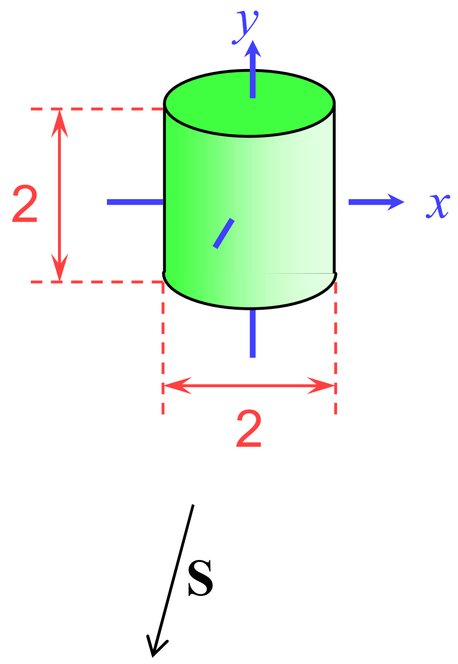


- this is the only cylinder that can be drawn, *but* the package has a complete set of 3D transformations
- we want to draw a cylinder of:
  - **radius 2 units**
  - **the centres of its two ends located at (1,2,3) and (2,4,5)**
    - its length is thus 3 units
- what transforms are required? and in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



Model transformation 2

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

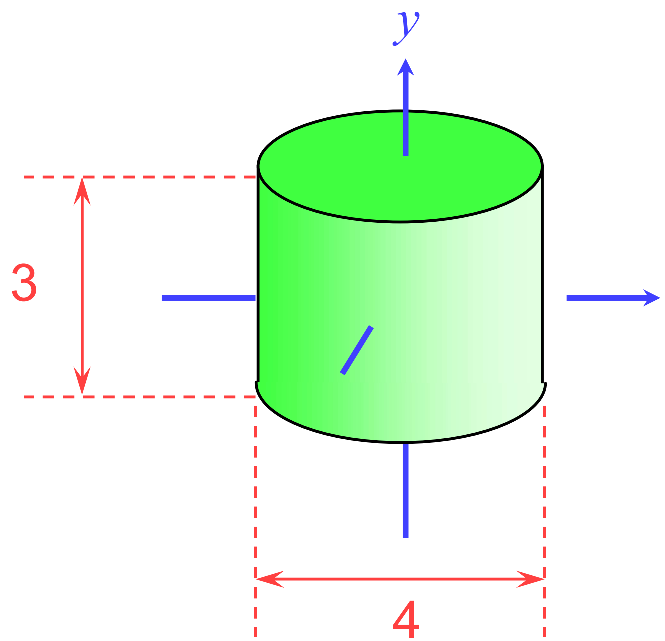


$$\mathbf{S} = \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 1.5 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

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)

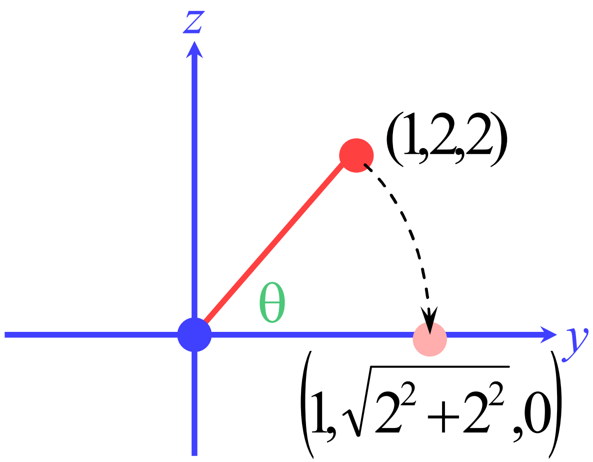
Model transformation 4

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

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

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

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



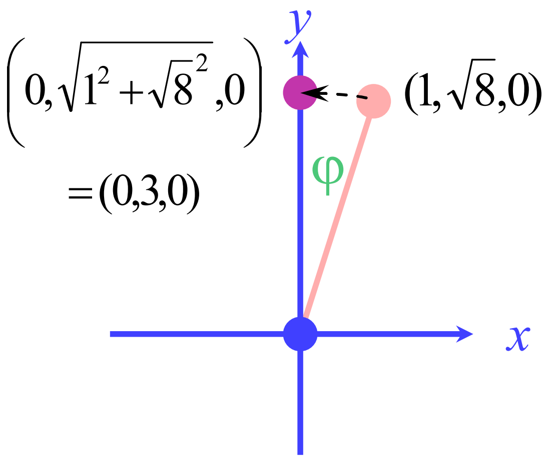
$$\left(1,\sqrt{2^2+2^2},0\right)$$
$$=(1,\sqrt{8},0)$$

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

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

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



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

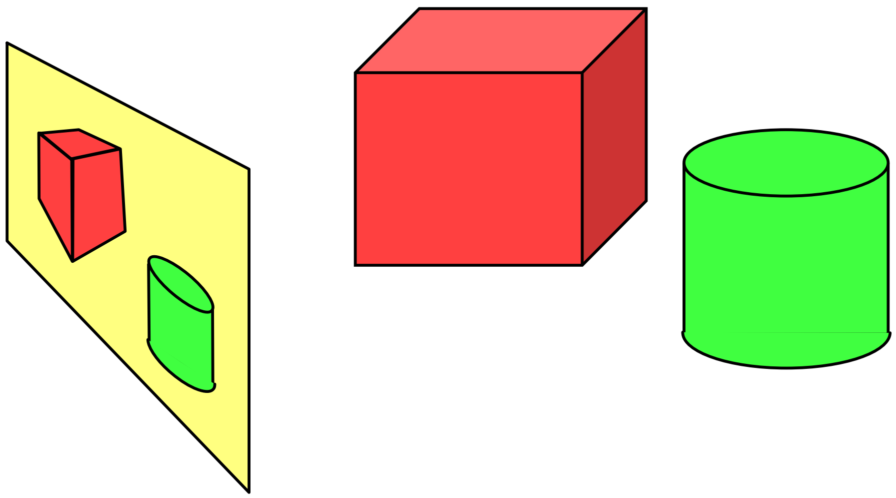
$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \mathbf{T} \times \mathbf{R}_1^{-1} \times \mathbf{R}_2^{-1} \times \mathbf{S} \times \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

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

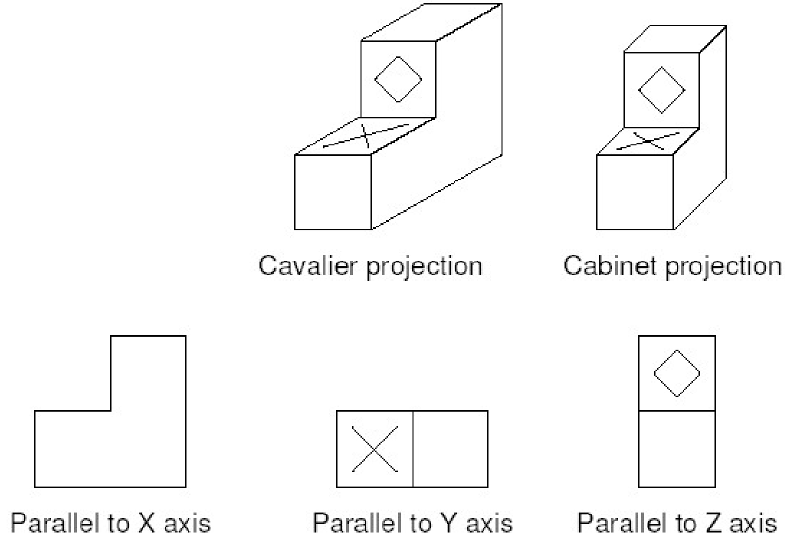


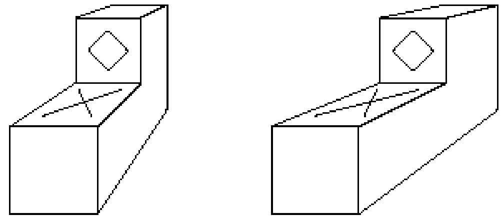
The 3D world is described as a set of (mathematical) objects

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

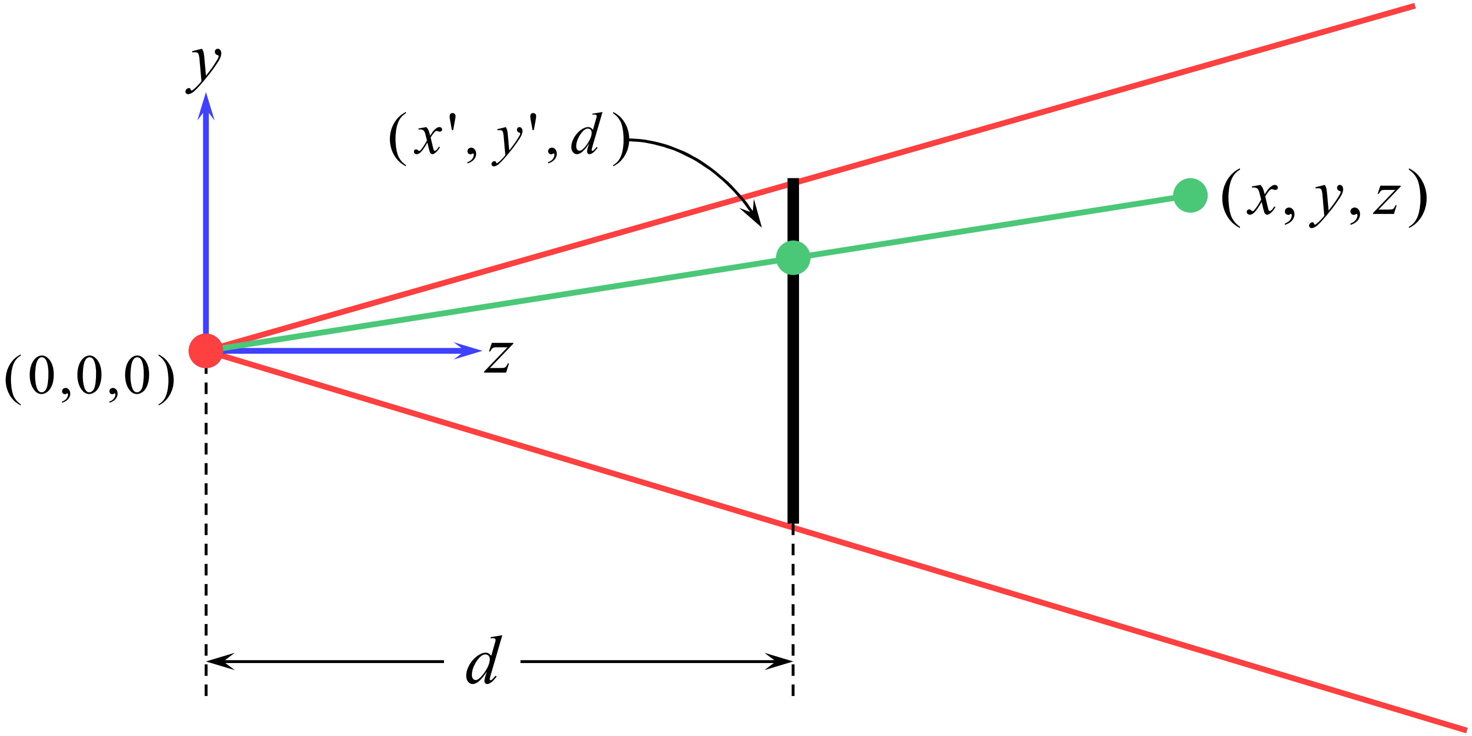
Types of projection

- 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





Geometry of perspective projection



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

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

Projection as a matrix operation

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

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

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

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

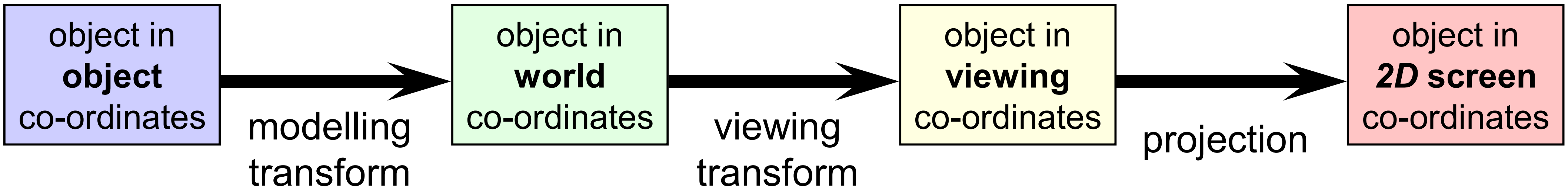
remember $\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} \rightarrow \begin{bmatrix} x/w \\ y/w \\ z/w \end{bmatrix}$

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

Perspective projection with an arbitrary camera

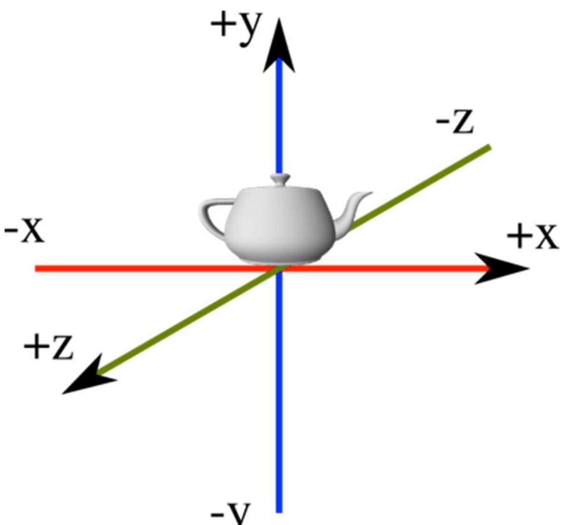
- we have assumed that:
  - screen centre at **(0,0,d)**
  - screen parallel to xy-plane
  - z-axis into screen
  - y-axis up and x-axis to the right
  - eye (camera) at origin **(0,0,0)**
- for an arbitrary camera we can either:
  - work out equations for projecting objects about an arbitrary point 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



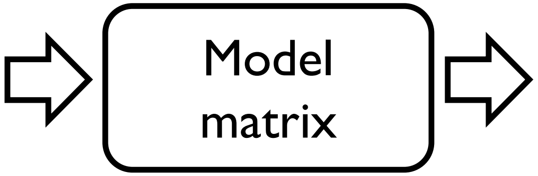
- the modelling transform and viewing transform can be multiplied together to produce a single matrix taking an object directly from object co-ordinates into viewing co-ordinates
- either or both of the modelling transform and viewing transform matrices can be the identity matrix
  - e.g. objects can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

Model, View, Projection matrices

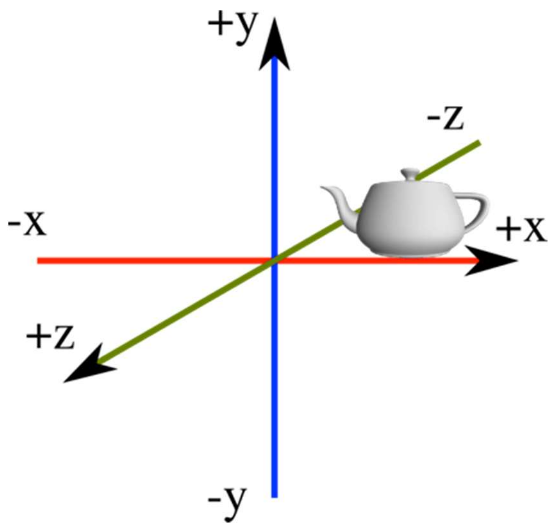


Object coordinates

Object centred at the origin

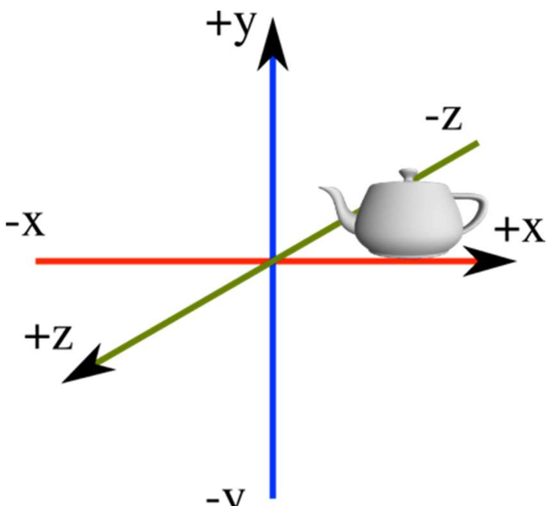


To position each object in the scene. Could be different for each object.



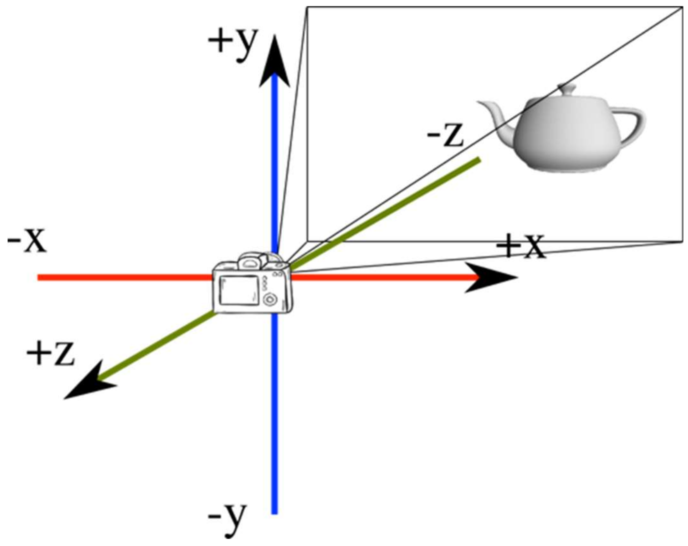
World coordinates

Model, View, Projection matrices



World coordinates

To position all objects relative to the camera

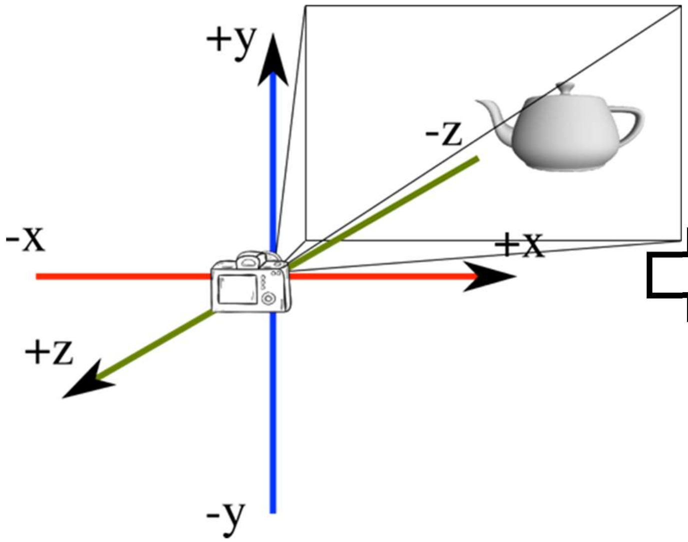


View (camera) coordinates

Camera at the origin, pointing at -z

Model, View, Projection matrices

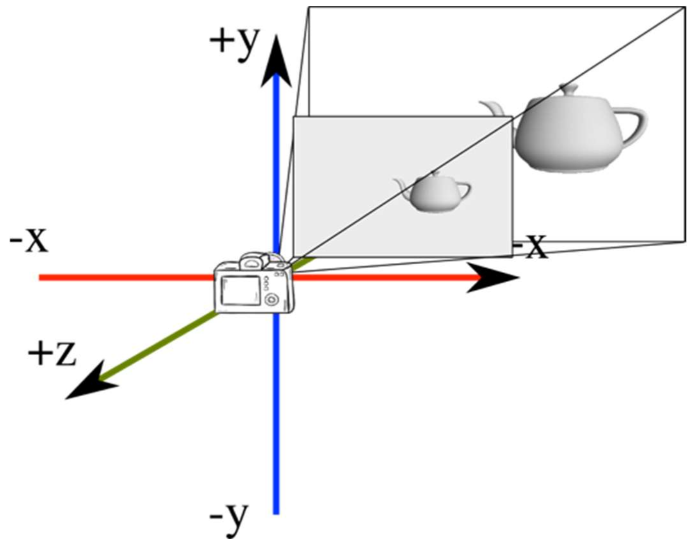
The default OpenGL coordinate system is right-handed



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

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

Screen coordinates

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

3D world vertex coordinates

Projection, view and model matrices

Viewing transform: look at

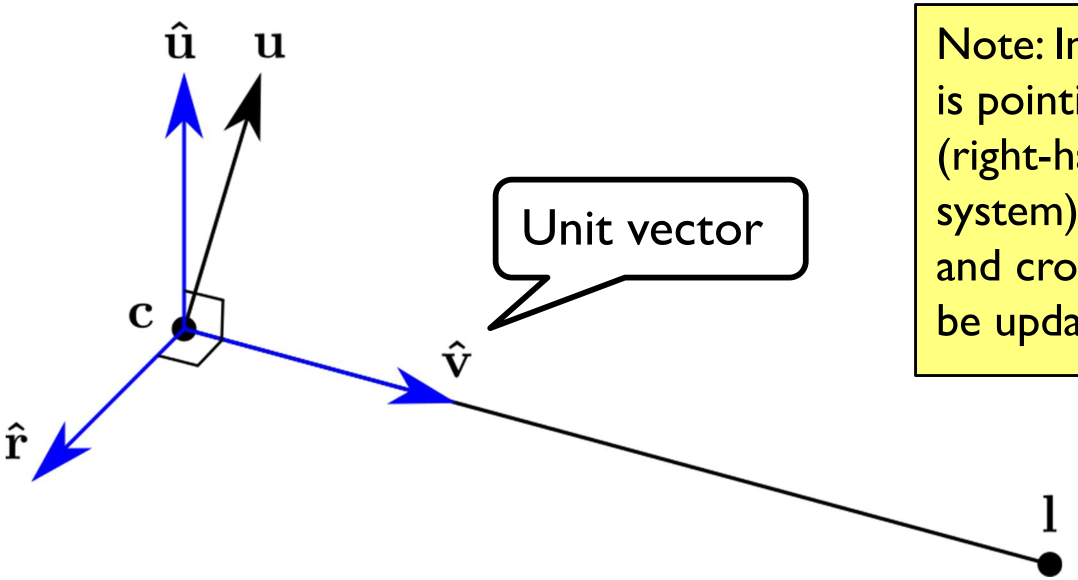
- Task: find a viewing transform so that the camera centre is at $\boldsymbol{c}$, is directed towards $\boldsymbol{l}$ and vector $\boldsymbol{u}$ is the “up” direction

For a left-handed coordinate system:

$$\hat{\boldsymbol{v}} = \frac{\boldsymbol{l} - \boldsymbol{c}}{\|\boldsymbol{l} - \boldsymbol{c}\|}$$

$$\hat{\boldsymbol{r}} = \frac{\hat{\boldsymbol{v}} \times \boldsymbol{u}}{\|\hat{\boldsymbol{v}} \times \boldsymbol{u}\|}$$

$$\hat{\boldsymbol{u}} = \hat{\boldsymbol{r}} \times \hat{\boldsymbol{v}}$$

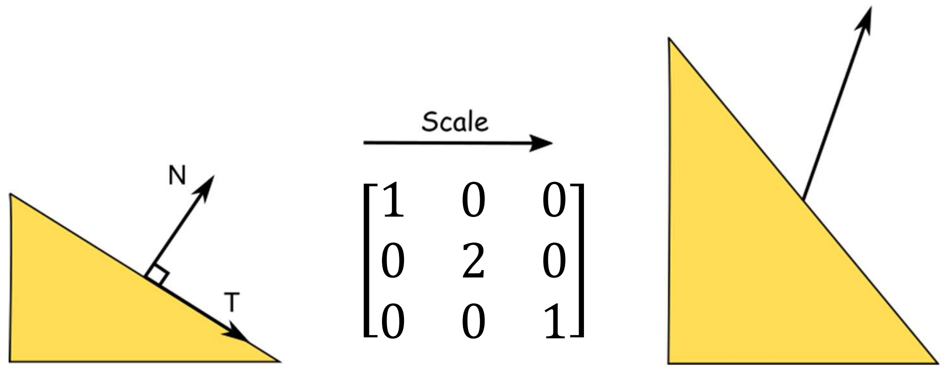


Note: In OpenGL, vector $\hat{\boldsymbol{v}}$ is pointing away from $\boldsymbol{l}$ (right-handed coordinate system). Therefore, the signs and cross produced must be updated accordingly

$$V = \underbrace{\begin{bmatrix} \hat{r}_x & \hat{r}_y & \hat{r}_z & 0 \\ \hat{u}_x & \hat{u}_y & \hat{u}_z & 0 \\ \hat{v}_x & \hat{v}_y & \hat{v}_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{\text{change of basis}} \underbrace{\begin{bmatrix} 1 & 0 & 0 & -c_x \\ 0 & 1 & 0 & -c_y \\ 0 & 0 & 1 & -c_z \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{\text{translation}} = \begin{bmatrix} \hat{r}_x & \hat{r}_y & \hat{r}_z & -\boldsymbol{c} \cdot \hat{\boldsymbol{r}} \\ \hat{u}_x & \hat{u}_y & \hat{u}_z & -\boldsymbol{c} \cdot \hat{\boldsymbol{u}} \\ \hat{v}_x & \hat{v}_y & \hat{v}_z & -\boldsymbol{c} \cdot \hat{\boldsymbol{v}} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Transforming normal vectors

- Transformation by a nonorthogonal matrix does not preserve angles



- Since:

$$N \cdot T = 0$$

Normal transform

$$N' \cdot T' = (GN) \cdot (\mathrm{MT}) = 0$$

Transformed normal and tangent vector

Vertex position transform

- We can find that: $G = (M^{-1})^T$
  - Derivation shown in the lecture

[FCG 6.2.2/7.2.2]

Scene construction

- We will build a robot from basic parts
- Body transformation

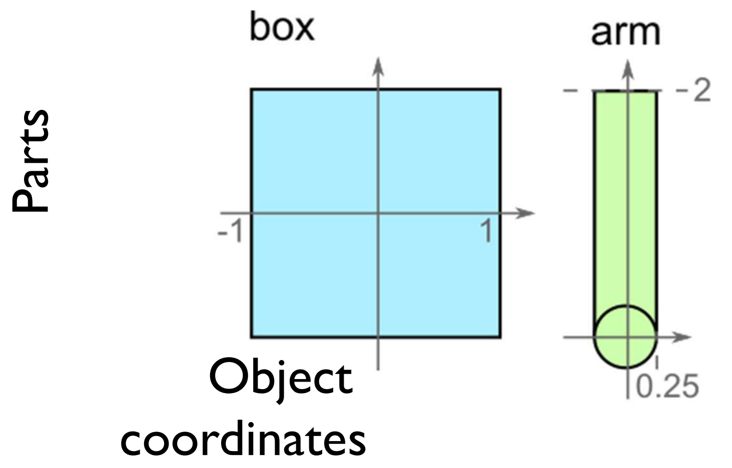
$$M_{body} = ?$$

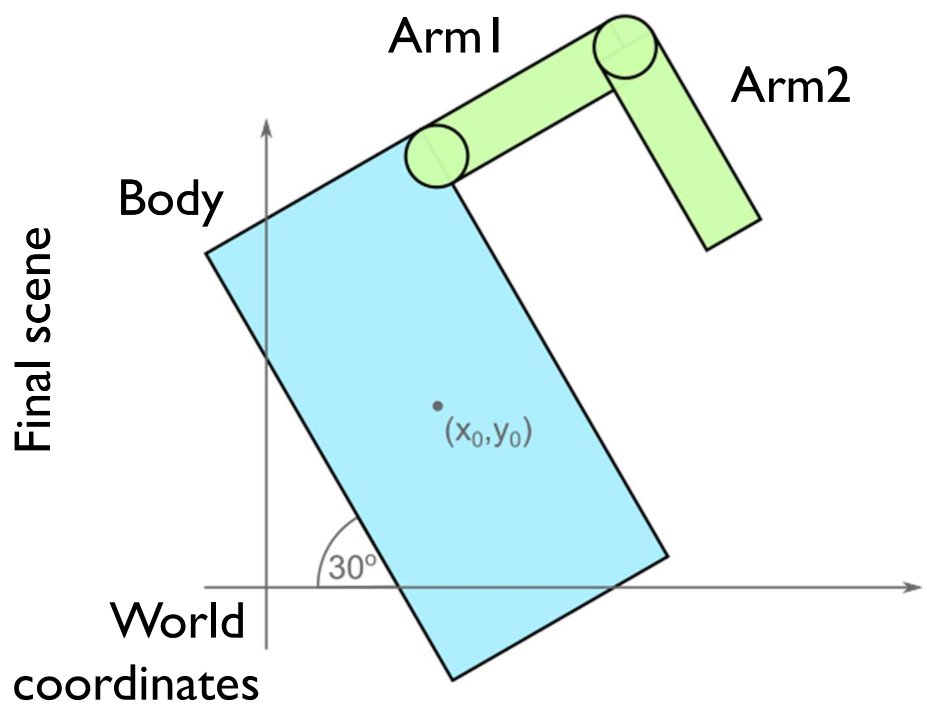
- Arm1 transformation

$$M_{arm1} = ?$$

- Arm 2 transformation

$$M_{arm2} = ?$$





Scene construction

✦ Body transformation

$$E_{body} = scale \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

$$T_{body} = translate \begin{bmatrix} x_0 \\ y_0 \end{bmatrix} \cdot rotate(30^o)$$

$$M_{body} = T_{body} E_{body}$$

✦ Arm1 transformation

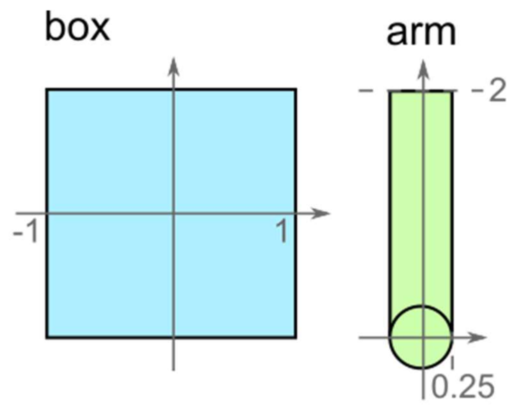
$$T_{arm1} = translate \begin{bmatrix} 1 \\ 1.75 \end{bmatrix} \cdot rotate(-90^o)$$

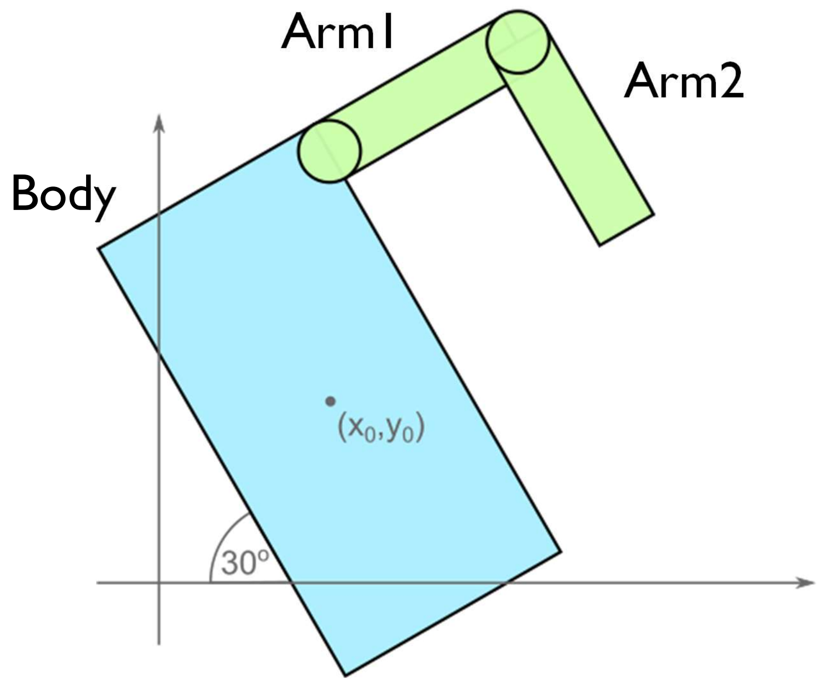
$$M_{arm1} = T_{body} T_{arm1}$$

✦ Arm2 transformation

$$T_{arm2} = translate \begin{bmatrix} 0 \\ 2 \end{bmatrix} \cdot rotate(-90^o)$$

$$M_{arm2} = T_{body} T_{arm1} T_{arm2}$$

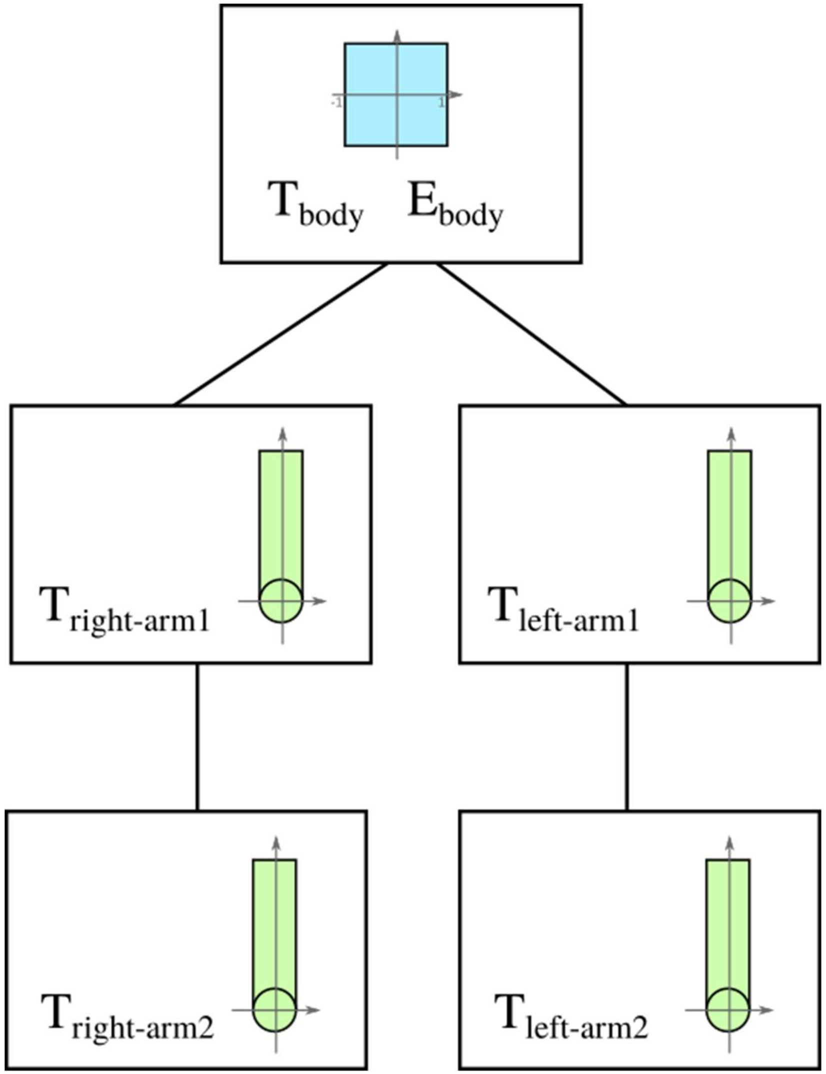




Scene Graph

- A scene can be drawn by traversing a scene graph:

```
traverse( node, T_parent ) {
   M = T_parent * node.T * node.E
   node.draw(M)
   for each child {
         traverse( child, T_parent * node.T )
   }
}
```



[FCG 12.2/12.2]

Introduction to Computer Graphics

- Background
- Rendering
- Graphics pipeline
- **Rasterization**
- Graphics hardware and modern OpenGL
- Human vision and colour & tone mapping

Rasterization algorithm(*)

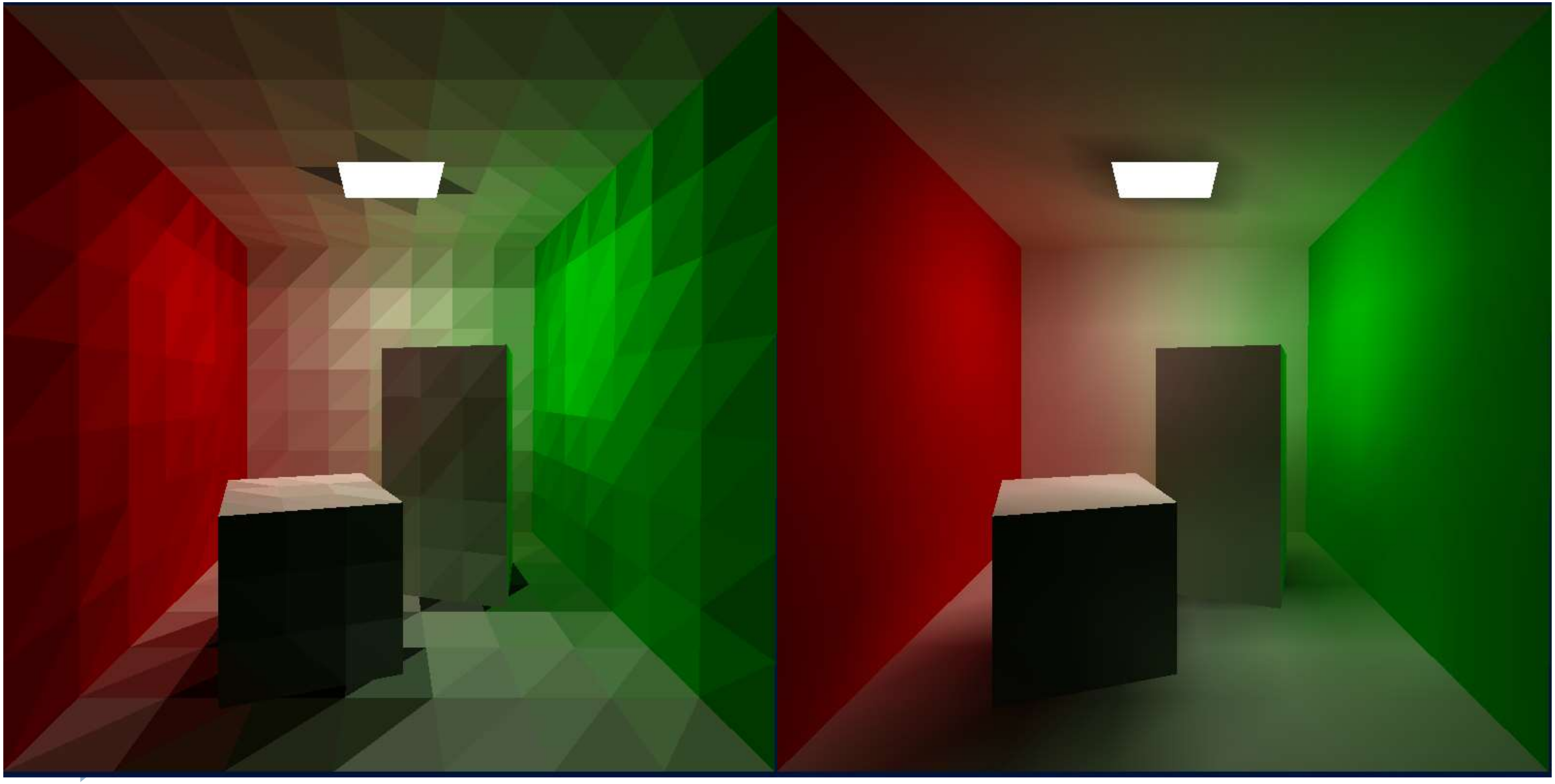
```
Set model, view and projection (MVP) transformations

FOR every triangle in the scene
    transform its vertices using MVP matrices
    IF the triangle is within a view frustum
        clip the triangle to the screen border
        FOR each fragment in the triangle
            interpolate fragment position and attributes between vertices
            compute fragment colour
            IF the fragment is closer to the camera than any pixel drawn so far
                update the screen pixel with the fragment colour
            END IF ;
        END FOR ;
    END IF ;
END FOR ;
```

fragment – a candidate pixel in the triangle

(*) simplified

Illumination & shading

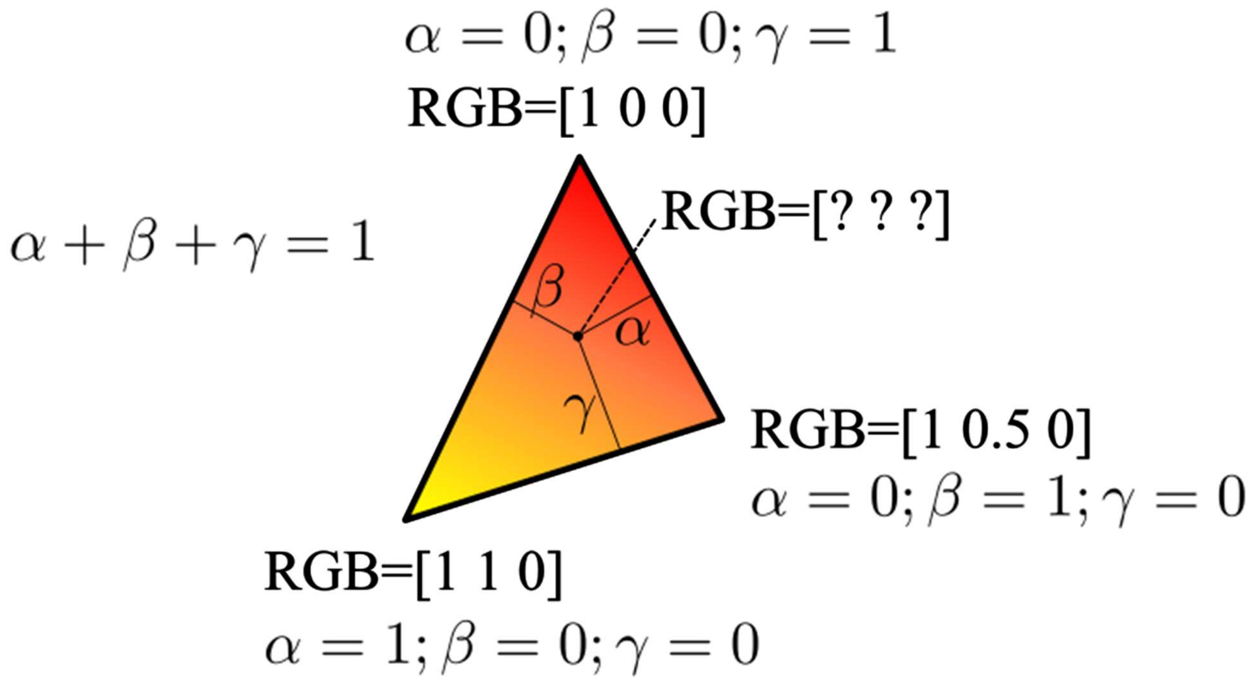
- Drawing polygons with uniform colours gives poor results
- Interpolate colours across polygons



Rasterization

- Efficiently draw (thousands of) triangles
  - Interpolate vertex attributes inside the triangle

- **Homogenous barycentric coordinates** are used to interpolate colours, normals, texture coordinates and other attributes inside the triangle



[FCG 2.7/2.9]

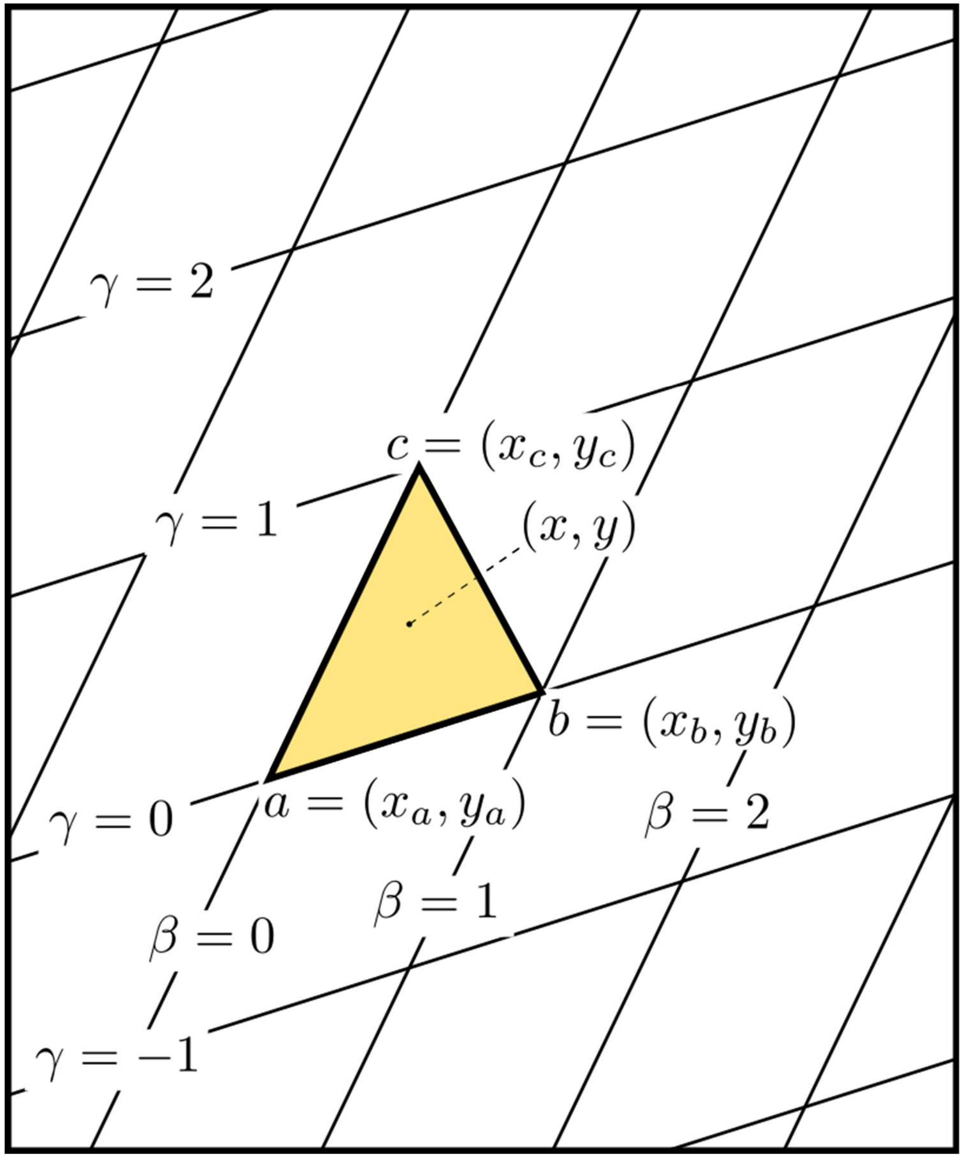
Homogenous barycentric coordinates

- Find barycentric coordinates of the point (x,y)
  - Given the coordinates of the vertices
  - Derivation in the lecture

$$\alpha = \frac{f_{cb}(x,y)}{f_{cb}(x_a,y_a)} \quad \beta = \frac{f_{ac}(x,y)}{f_{ac}(x_b,y_b)}$$

$f_{ab}(x,y)$ is the implicit line equation:

$f_{ab}(x,y) = (y_a - y_b)x + (x_b - x_a)y + x_a y_b - x_b y_a$



Triangle rasterization

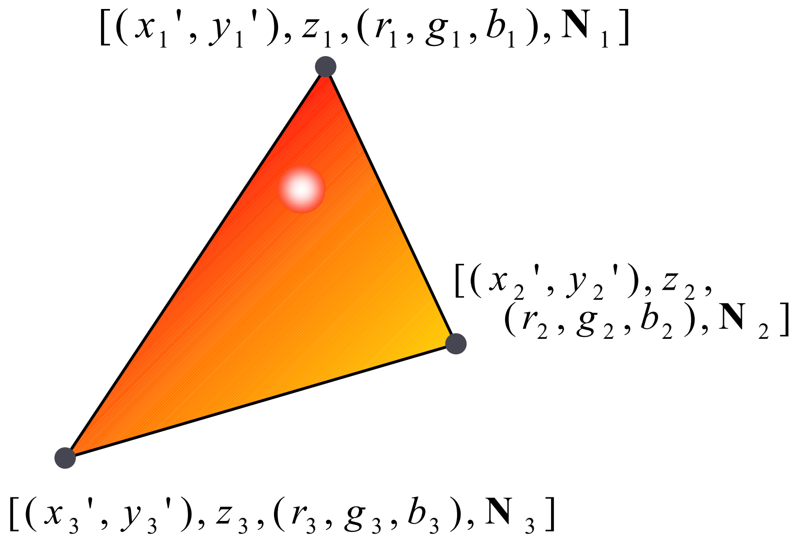
```
for y=y_min to y_max do
        for x=x_min to x_max do
                α = f_cb(x,y)/f_cb(x_a,y_a)
                β = f_ac(x,y)/f_ac(x_b,y_b)
                γ = 1 − α − β
                if (α > 0 and β > 0 and γ > 0 ) then
                        c = αc_a + βc_b + γc_c
                        draw pixels (x,y) with colour c
```

- Optimization: the barycentric coordinates will change by the same amount when moving one pixel right (or one pixel down) regardless of the position
  - Precompute increments $\Delta\alpha, \Delta\beta, \Delta\gamma$ and use them instead of computing barycentric coordinates when drawing pixels sequentially

Surface normal vector interpolation

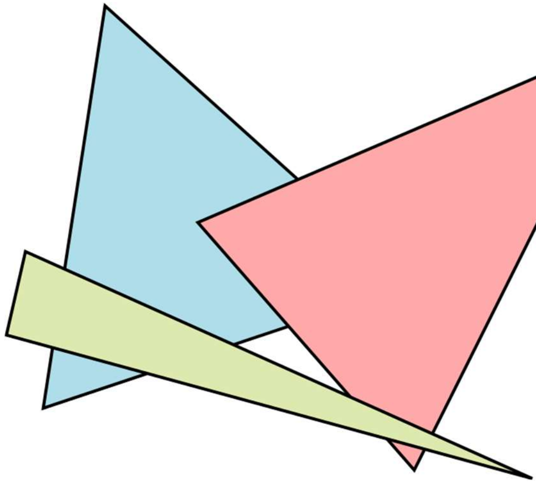
- for a polygonal model, interpolate normal vector between the vertices
  - Calculate colour (Phong reflection model) for each pixel
  - Diffuse component can be either interpolated or computed for each pixel

- N.B. Phong's approximation to specular reflection ignores (amongst other things) the effects of glancing incidence (the Fresnel term)

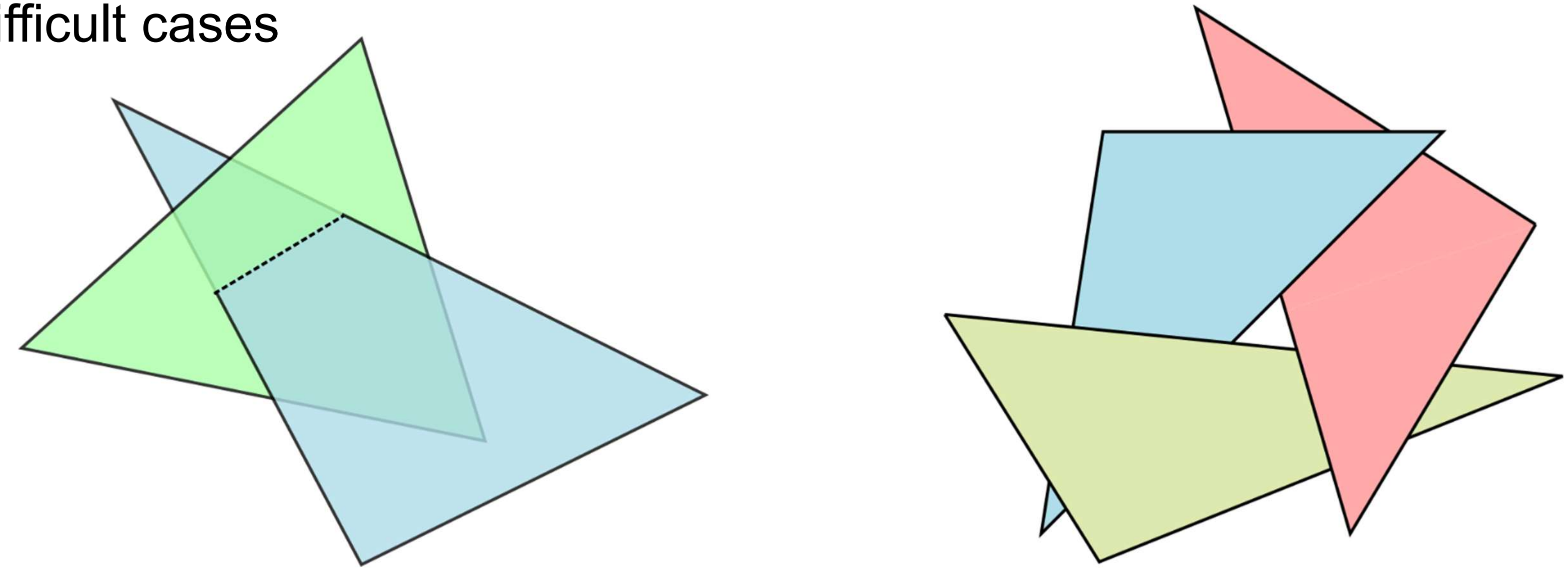


Occlusions (hidden surfaces)

Simple case



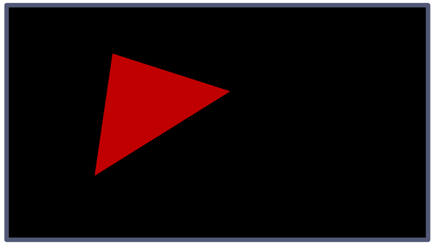
More difficult cases



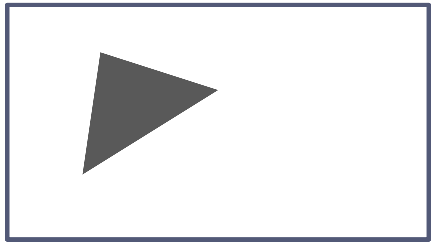
[FCG 8.2.3/9.2.3]

Z-Buffer - algorithm

Colour buffer



Depth buffer



- Initialize the depth buffer and image buffer for all pixels

  colour(x, y) = Background_colour,

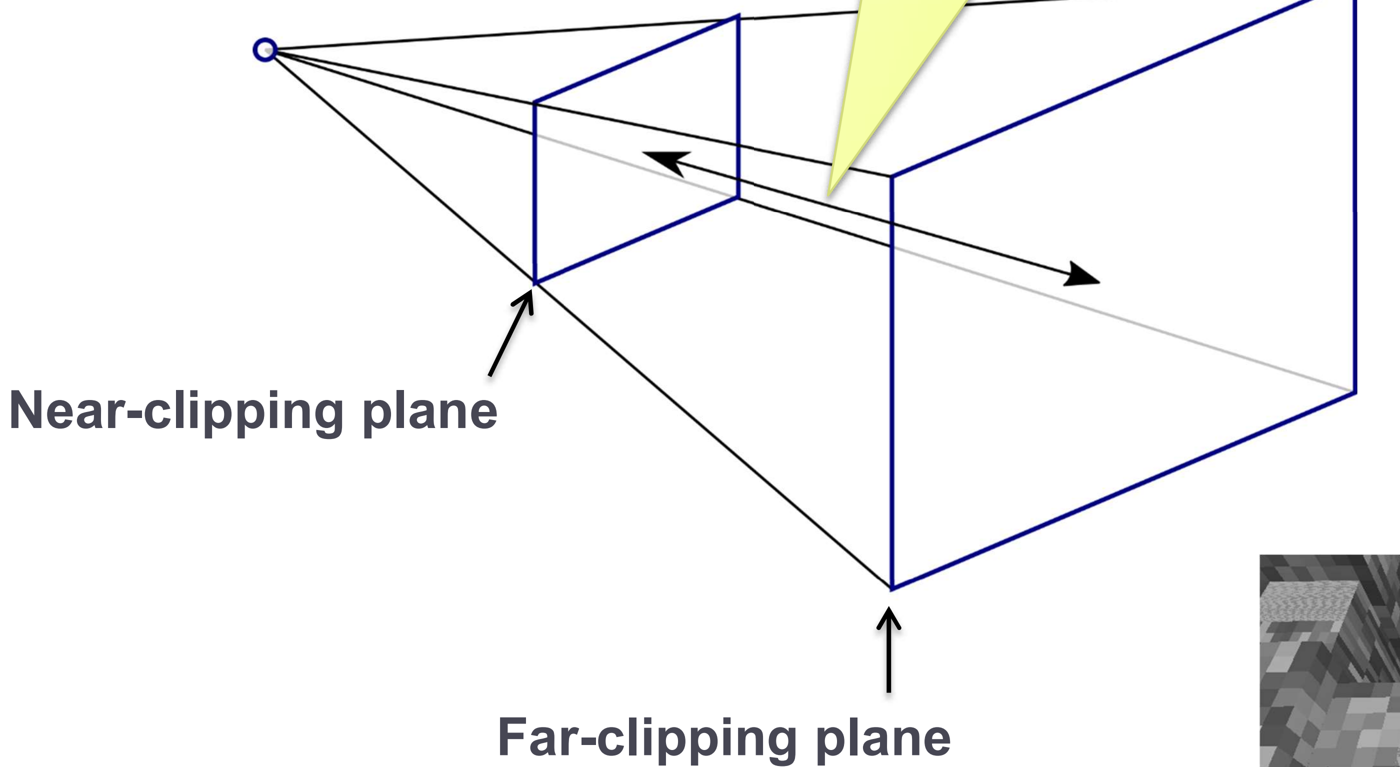
  depth(x, y) = z_{max} // position of the far clipping plane

- **For** every triangle in a scene **do**
  - **For** every fragment (x, y) in this triangle **do**
    - Calculate z for current (x, y)
    - **if** (z < depth(x, y)) and (z > z_{min}) **then**
      - depth(x, y) = z
      - colour(x, y) = fragment_colour(x, y)

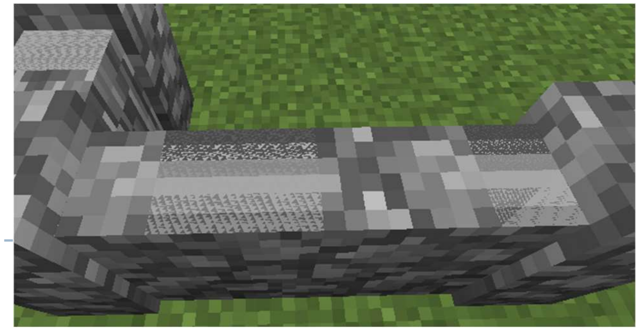
View frustum and Z-buffer

- Z-buffer must store depth with sufficient precision
  - 24 or 32 bit
  - Integer or float
  - Often $\frac{1}{z}$ instead of z

Range of values mapped to the Z-Buffer



Z-fighting

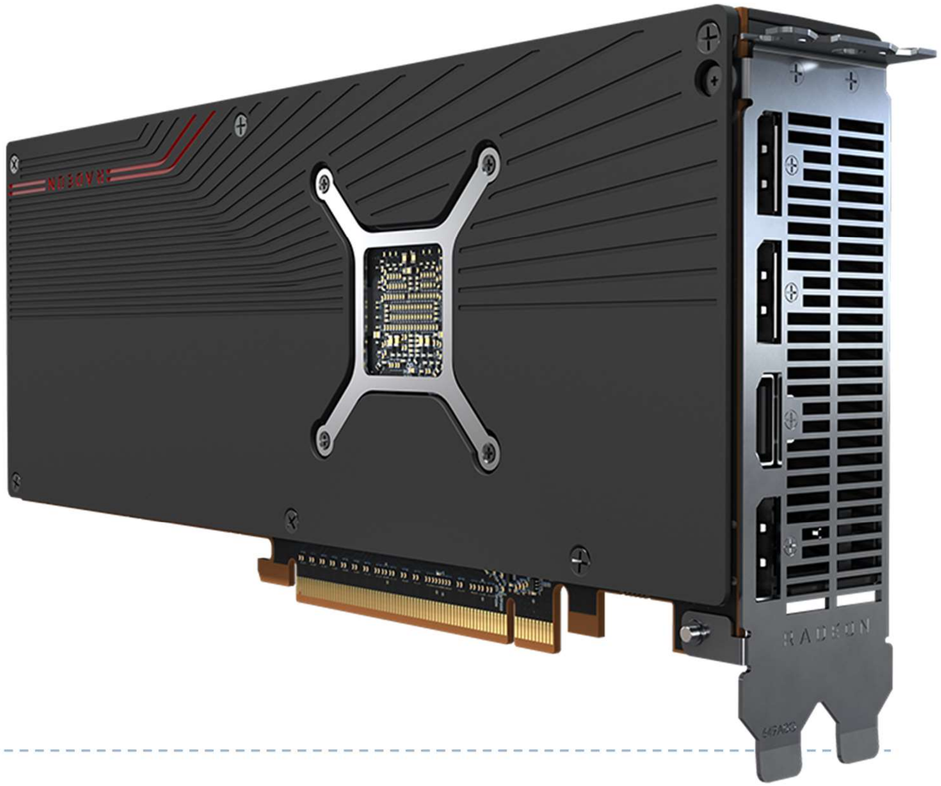


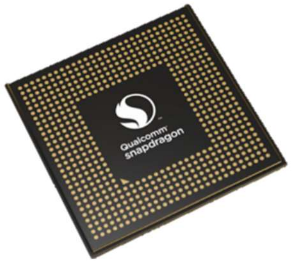
Introduction to Computer Graphics

- Background
- Rendering
- Graphics pipeline
- Rasterization
- **Graphics hardware and modern OpenGL**
  - **GPU & APIs**
  - **OpenGL Rendering pipeline**
  - **Example OpenGL code**
  - **GLSL**
  - **Textures**
  - **Raster buffers**
- Human vision, colour & tone mapping

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.





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, ...
  - Physics engines
- Full programmability at several pipeline stages
  - 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 independently 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
  - >>1000 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

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

And one more

- Metal (Apple iOS8)
  - low-level, low-overhead 3D GFX and compute shaders API
  - Support for Apple chips, Intel HD and Iris, AMD, Nvidia
  - Similar design as modern APIs, such as Vulcan
  - Swift or Objective-C API
  - Used mostly on iOS

GPGPU - general purpose computing

- OpenGL and DirectX are not meant to be used for general purpose computing
  - Example: physical simulation, machine learning
- 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 some CPUs
  - Supported by AMD, Intel and NVidia, Qualcomm, Apple, …

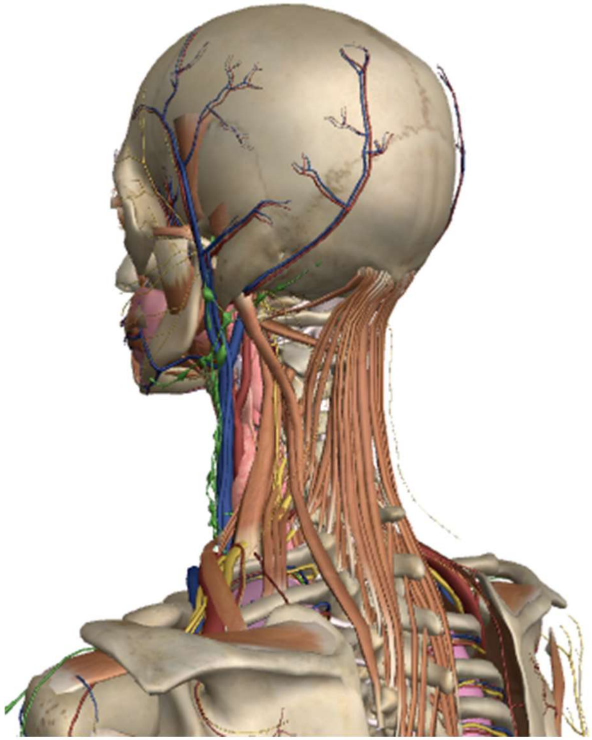
GPU and mobile devices

- OpenGL ES 1.0-3.2
  - Stripped version of OpenGL
  - Removed functionality that is not strictly necessary on mobile devices
- Devices
  - iOS: iPhone, iPad
  - 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
- WebGL 2.0 – based on OpenGL ES 3.0
- Most modern browsers support WebGL
- Used in 3D JavaScipt libraries
  - https://threejs.org/
  - WebXR



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, …)

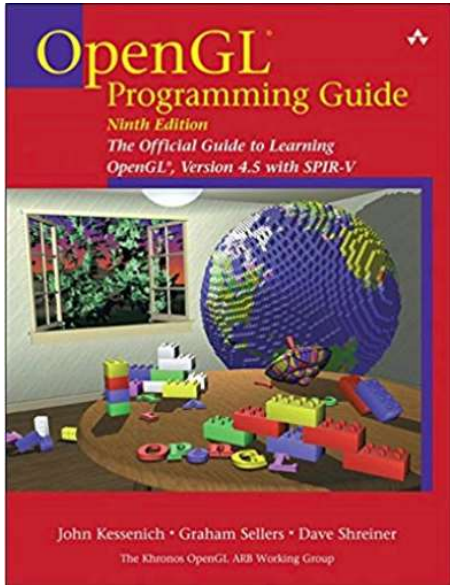
- We also need a linear algebra library
  - JOML – Java OpenGL Math Library
  - Operations on 2, 3, 4-dimensional vectors and matrices

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 4.6 (2017)
  - SPIR-V shaders

How to learn OpenGL?

- Lectures – algorithms behind OpenGL, general principles
- Tick 2 – detailed tutorial, learning by doing
- References
  - OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.5 with SPIR-V by John Kessenich, Graham Sellers, Dave Shreiner ISBN-10: 0134495497
  - OpenGL quick reference guide https://www.opengl.org/documentation/glsl/
  - Google search: „man gl......”



OpenGL rendering pipeline

OpenGL programming model

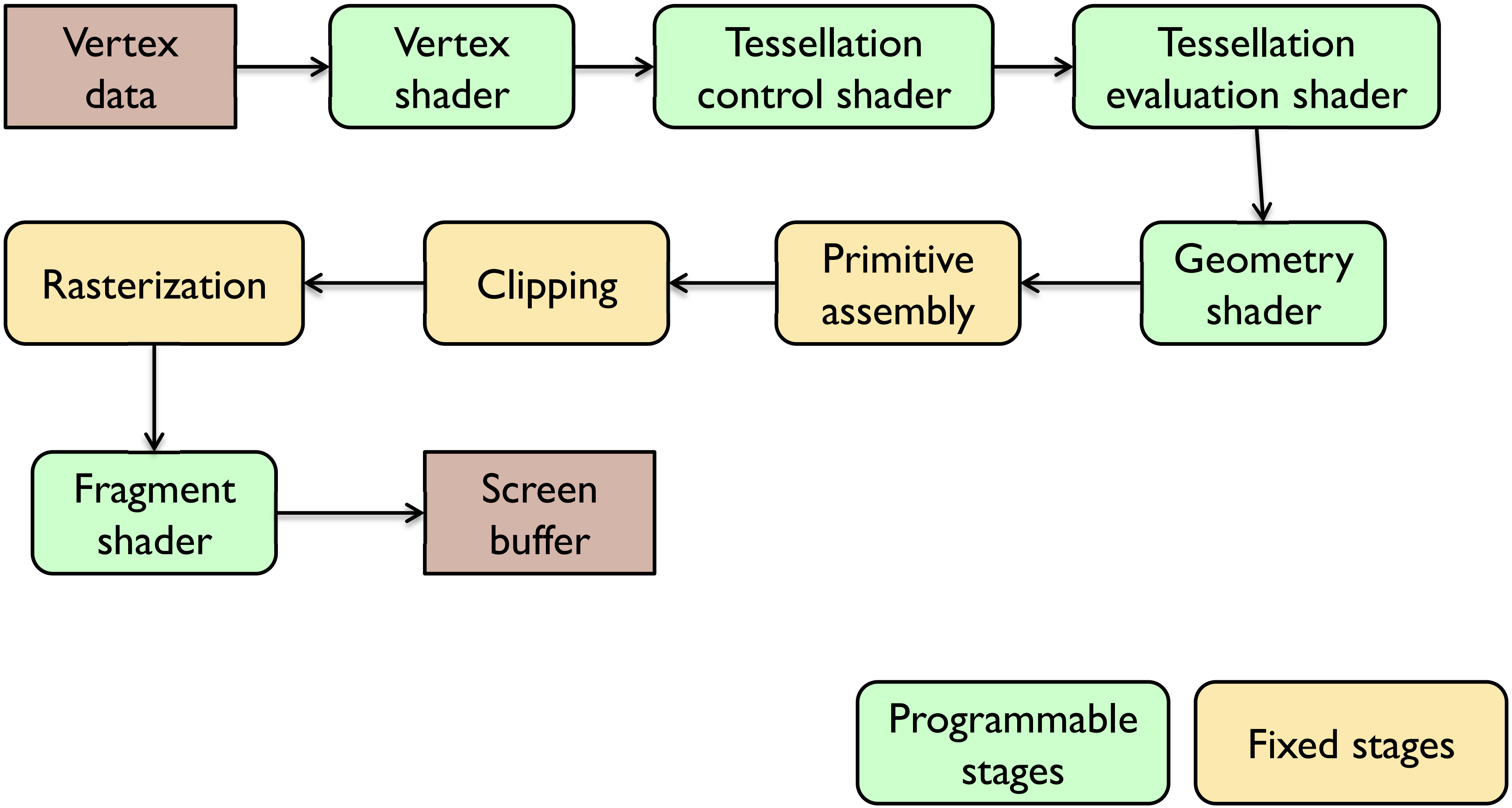
CPU code

- gl* functions that
  - Create OpenGL objects
  - Copy data CPU<->GPU
  - Modify OpenGL state
  - Enqueue operations
  - Synchronize CPU & GPU
- C99 library
- Wrappers in most programming language

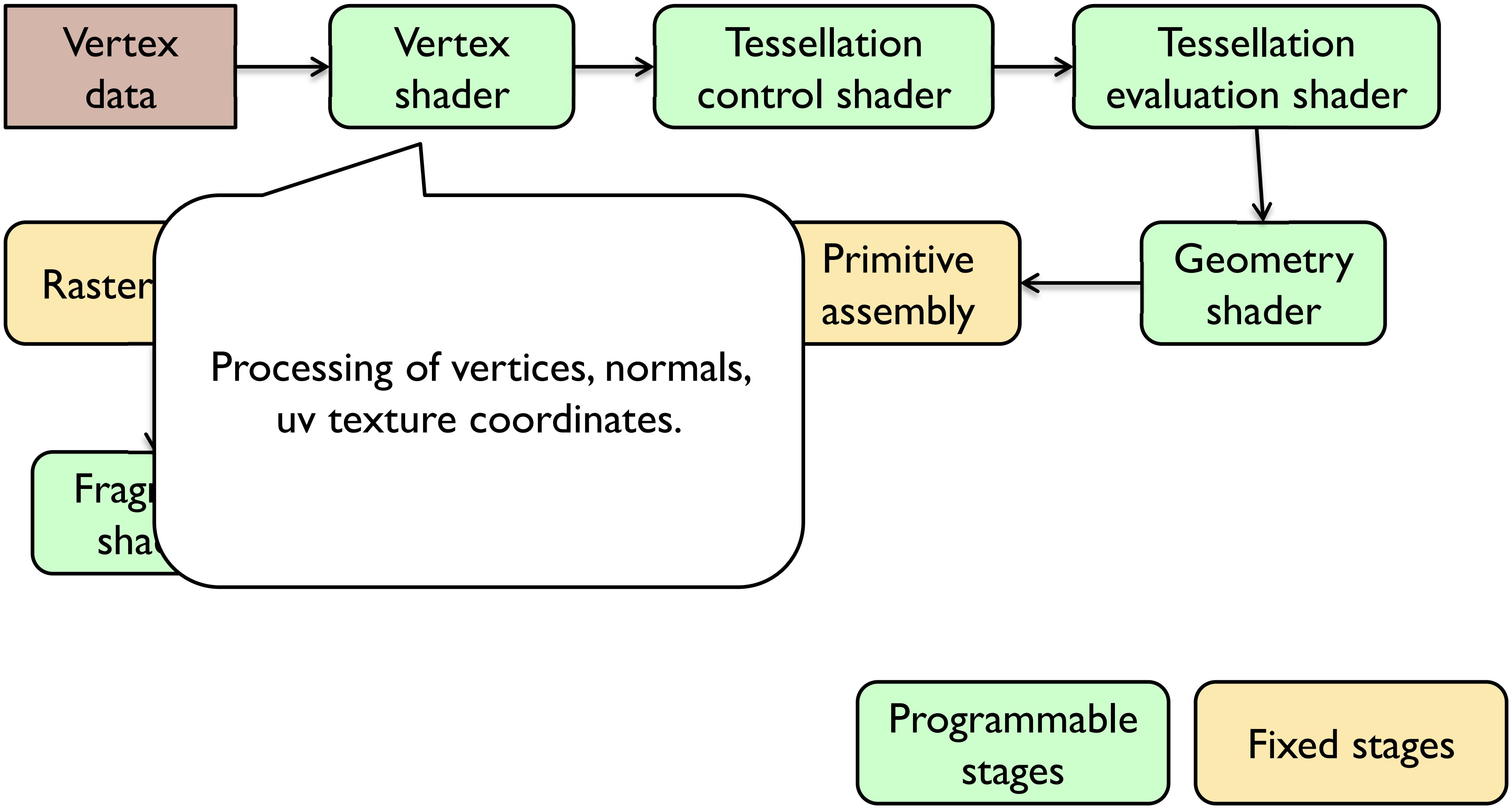
GPU code

- Fragment shaders
- Vertex shaders
- and other shaders
- Written in GLSL
  - Similar to C
  - From OpenGL 4.6 could be written in other language and compiled to SPIR-V

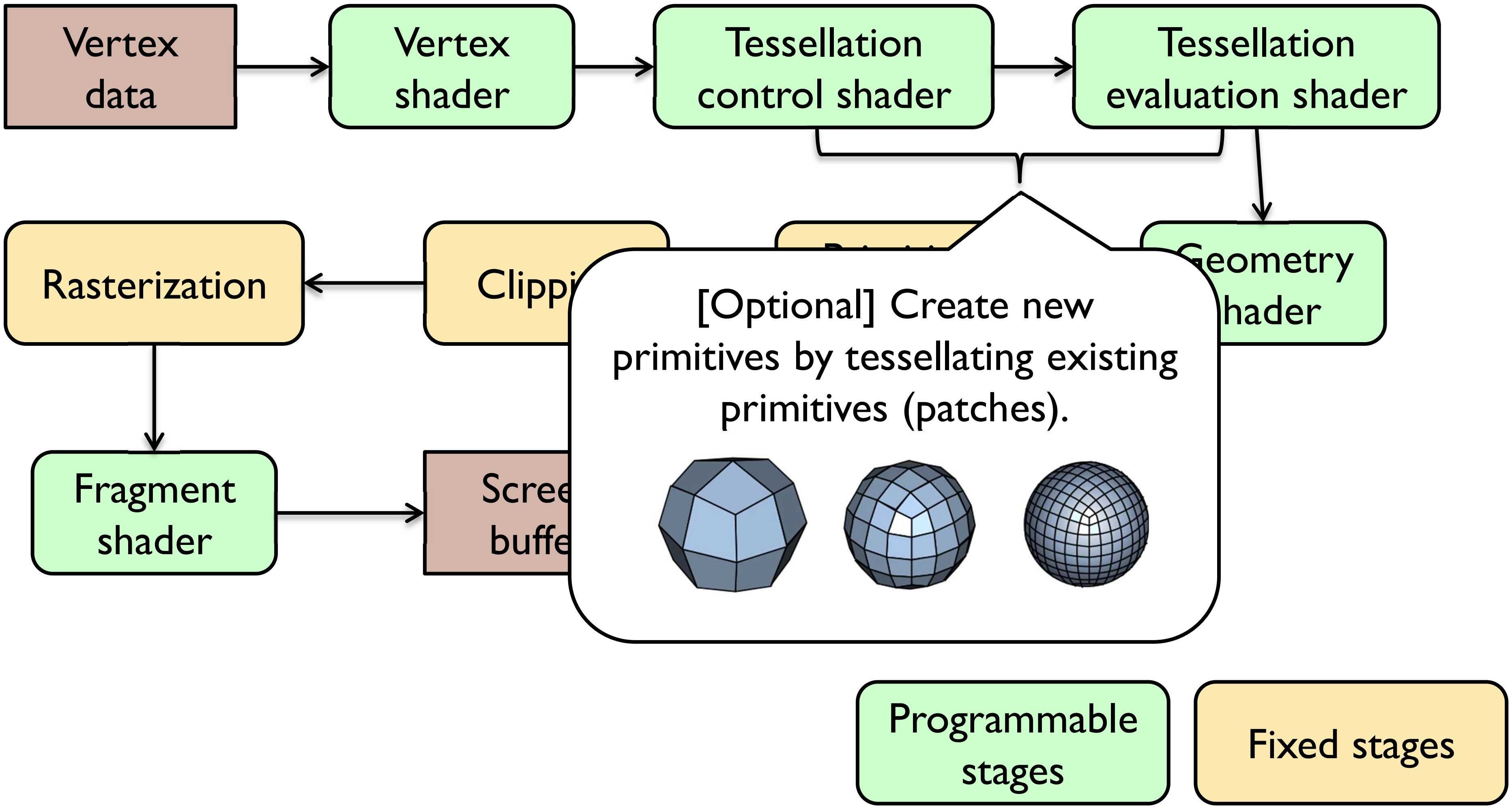
OpenGL rendering pipeline



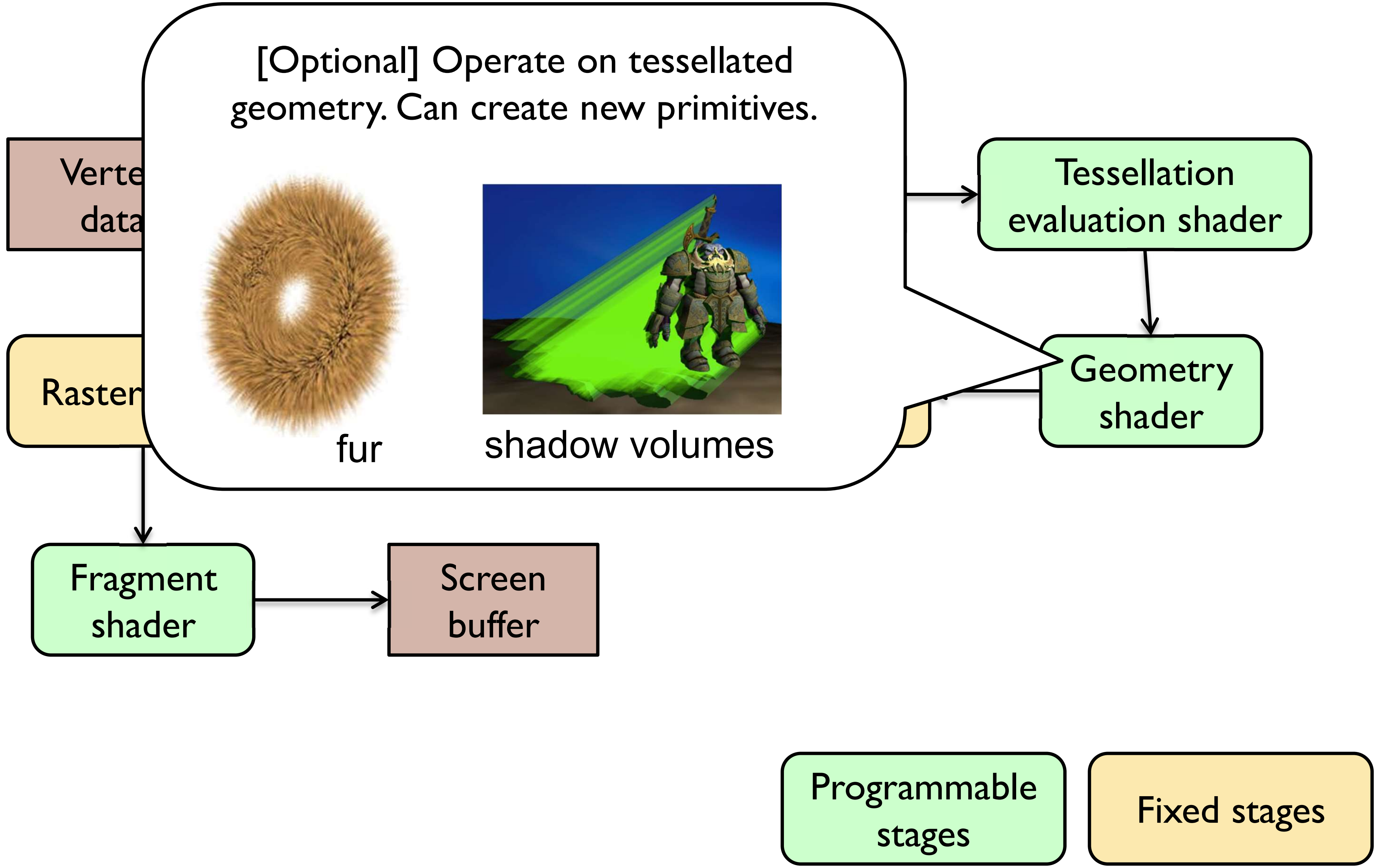
OpenGL rendering pipeline



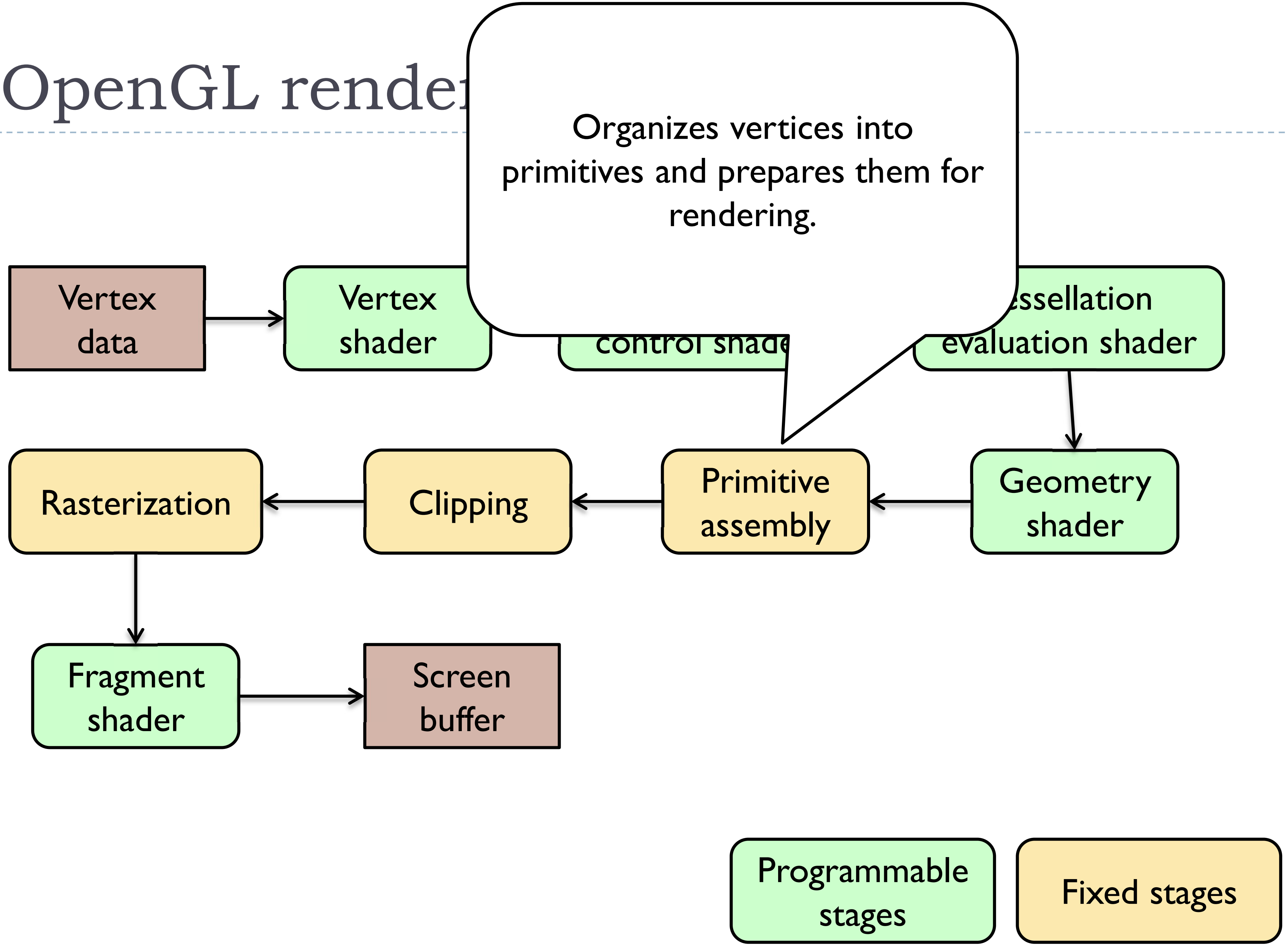
OpenGL rendering pipeline



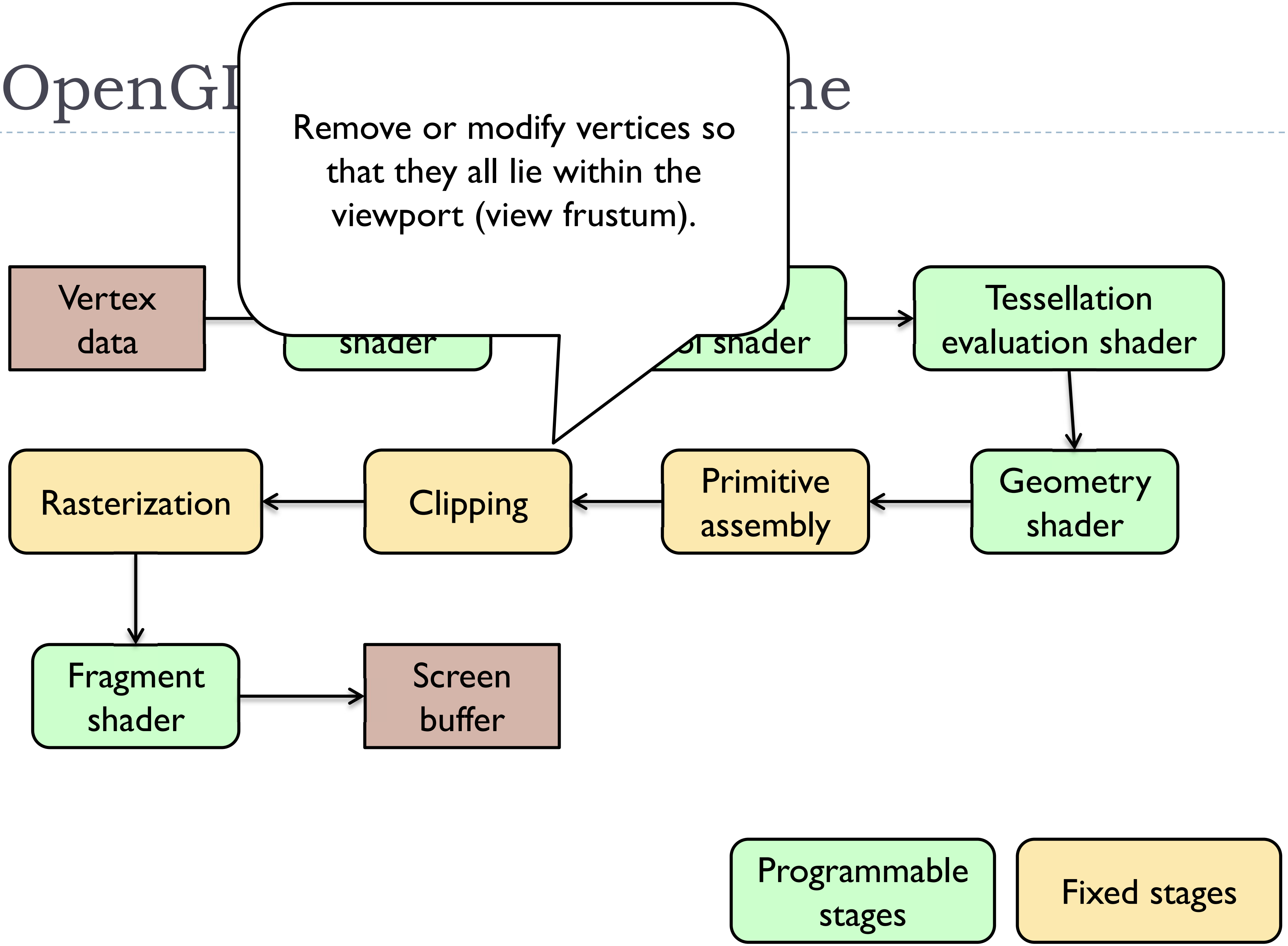
OpenGL rendering pipeline

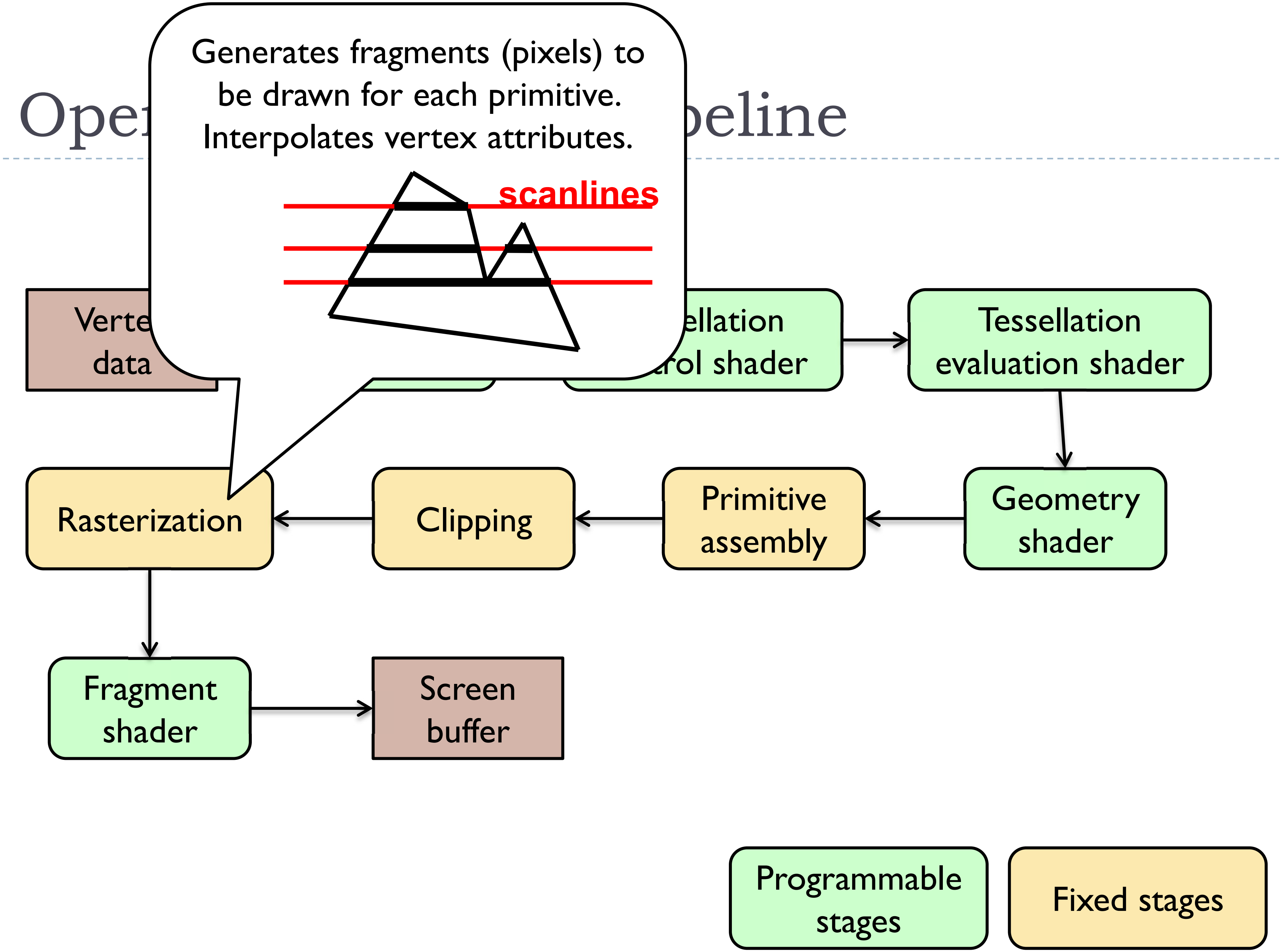


OpenGL rende



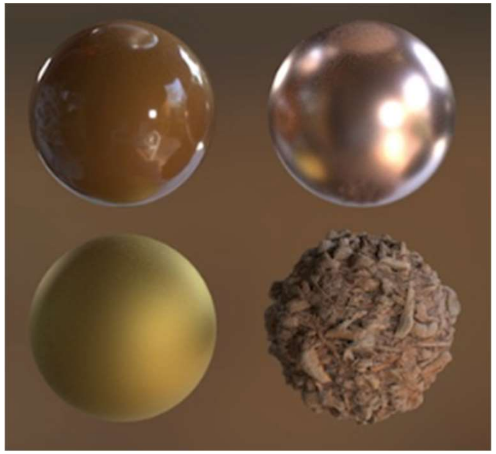
OpenGL ne


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


Ope
peline
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

Computes colour per each fragment (pixel). Can lookup colour in the texture. Can modify pixels' depth value.



Physically accurate materials

Non-Photorealistic-Rendering shader

Also used for tone mapping.

Vertex data

Rasterization

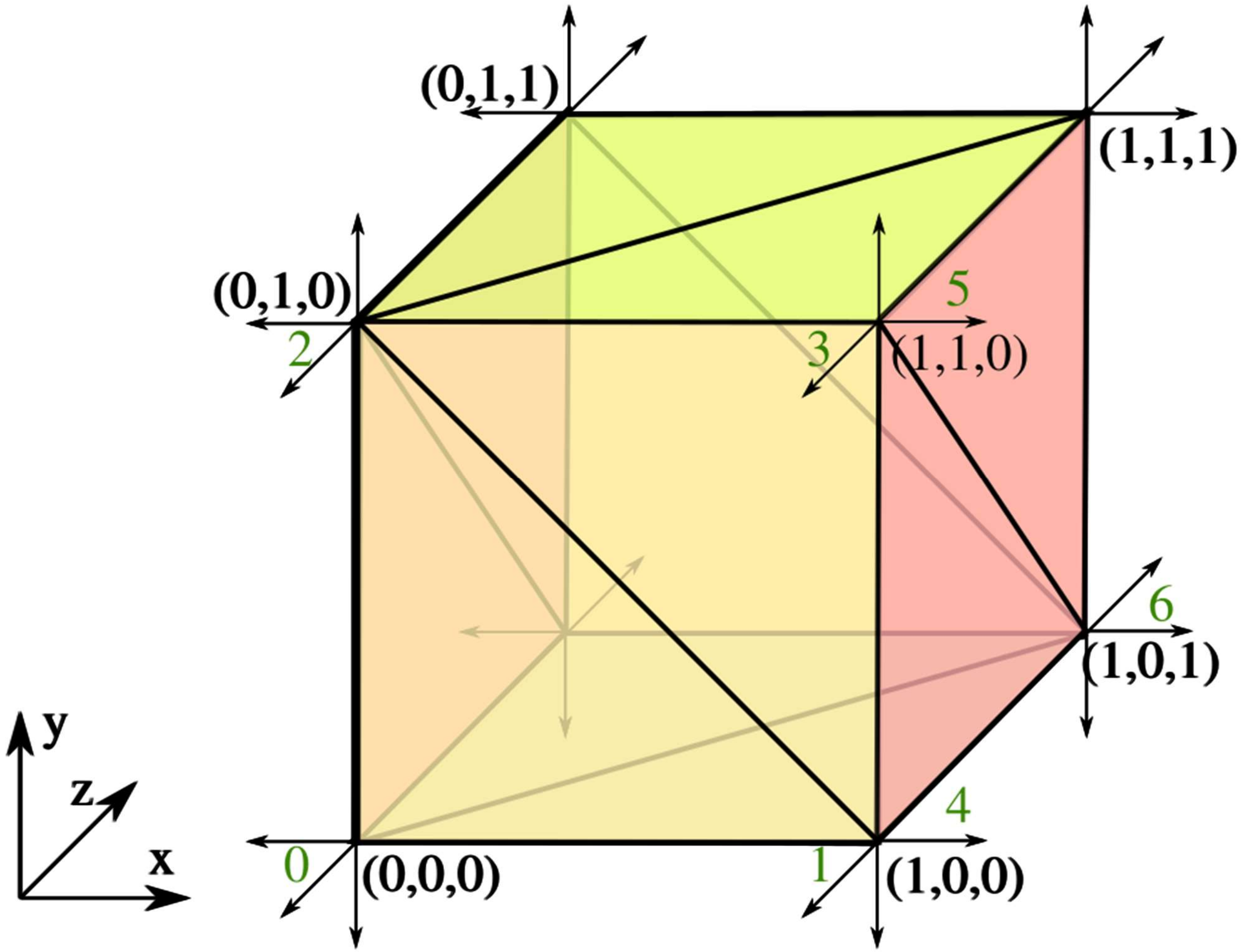
Fragment shader

buffer

Programmable stages

Fixed stages

Example: preparing vertex data for a cube



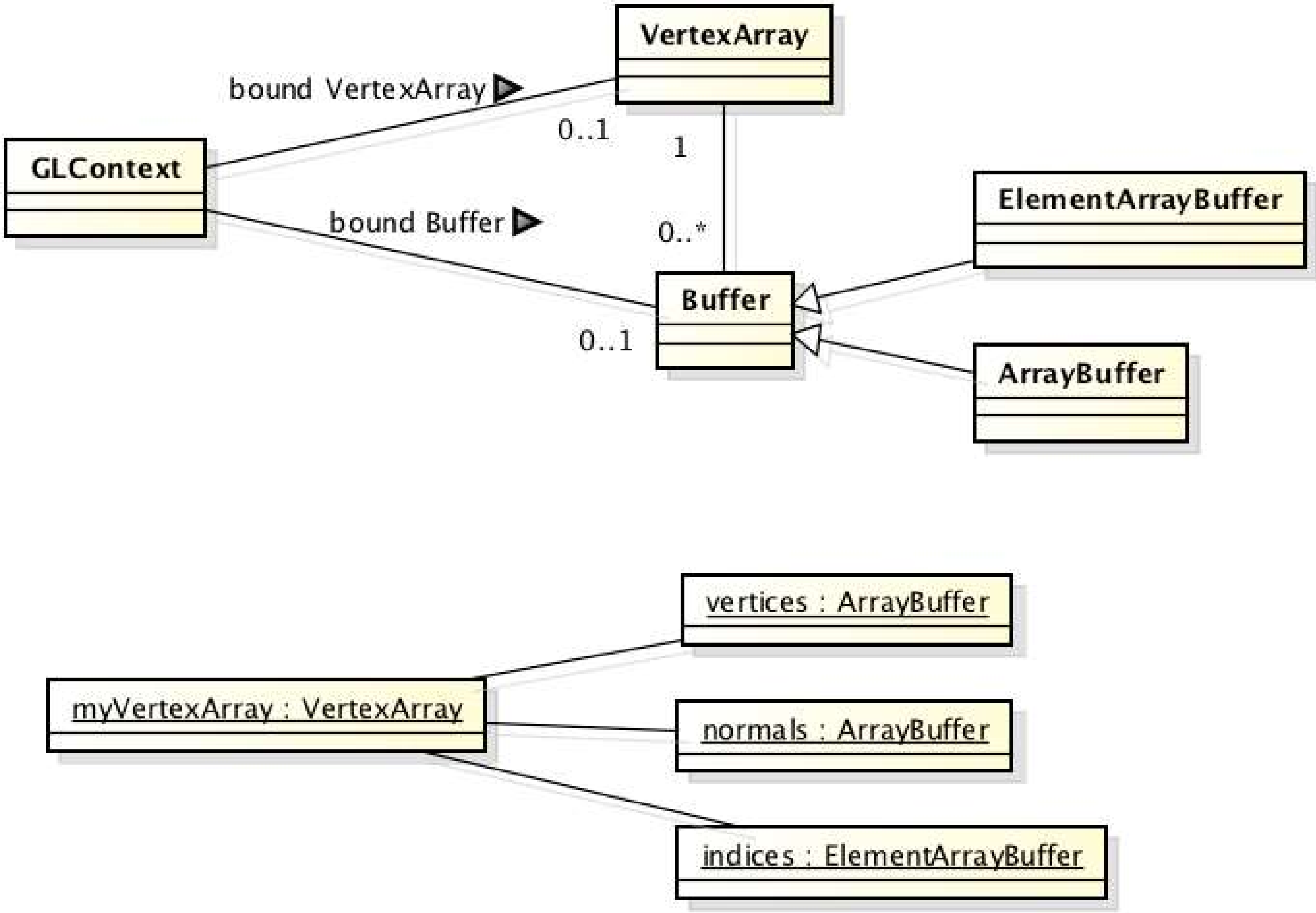
Primitives (triangles)

Indices
0, 1, 2
…

Vertex attributes

Ind	Positions	Normals
0	0, 0, 0	0, 0, -1
…	…	…

Geometry objects in OpenGL (OO view)



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()`"

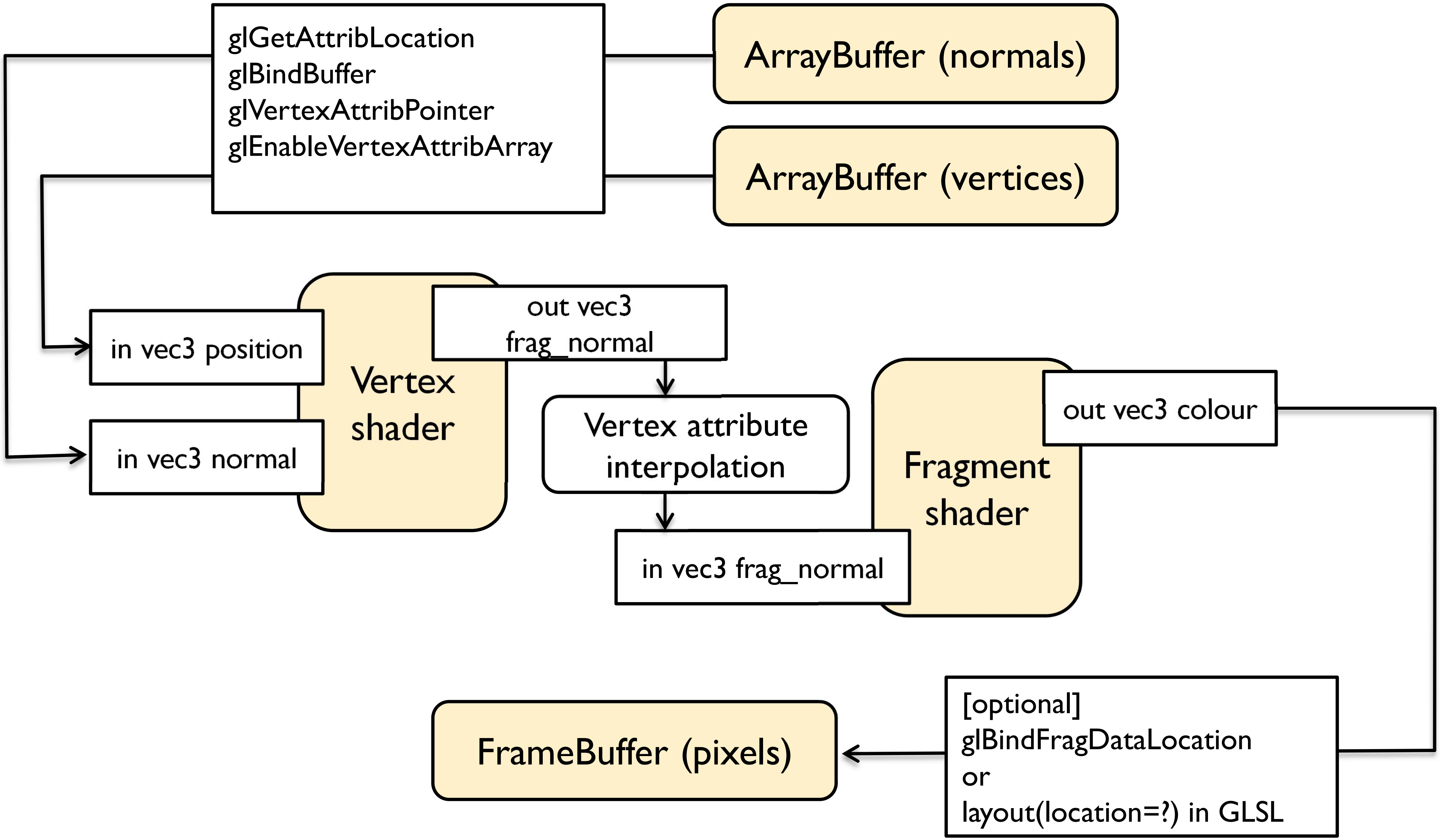
```
for( int i = 0; i < lut.length(); i++ ) {
   lut[i] *= 2;
}
```

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 drawn geometry
- `buffer` – shared with the application
- `shared` – shared with local work group (compute shaders only)

- Example: `const float pi=3.14;`

Shader inputs and outputs



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

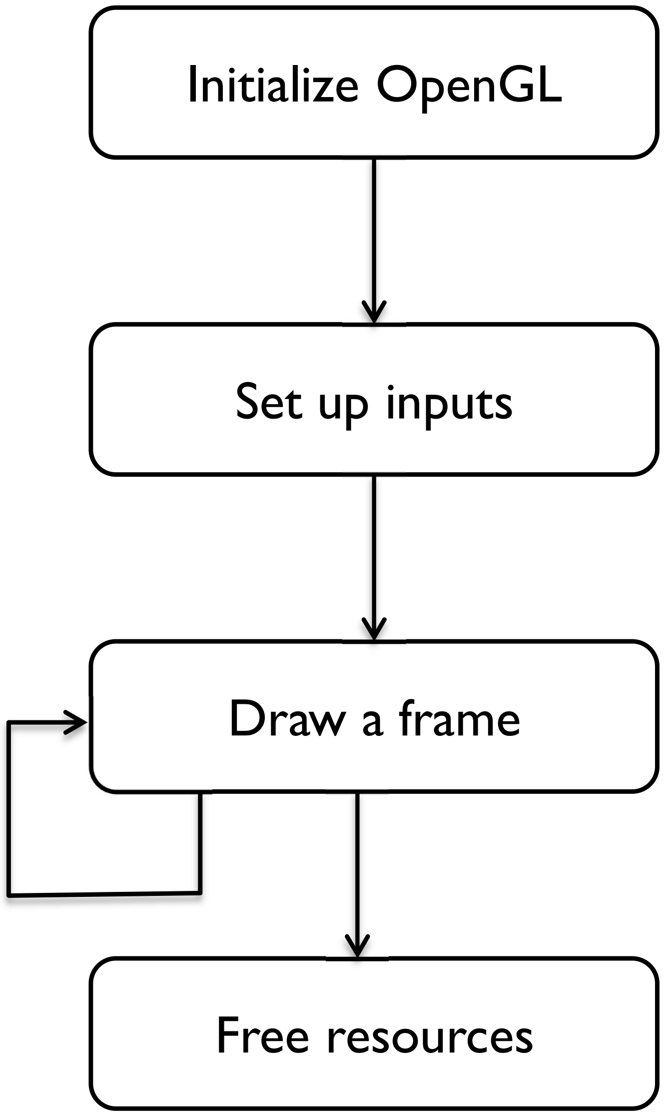
- Trigonometric:
  - `radians( deg ), degrees( rad ), sin, cos, tan, asin, acos, atan, sinh, cosh, tanh, asinh, acosh, atanh`
- Exponential:
  - `pow, exp, log, exp2, log2, sqrt, inversesqrt`
- Common functions:
  - `abs, round, floor, ceil, min, max, clamp,` …
- And many more

- 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 )
```

Simple OpenGL application - flow



- 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 geometry

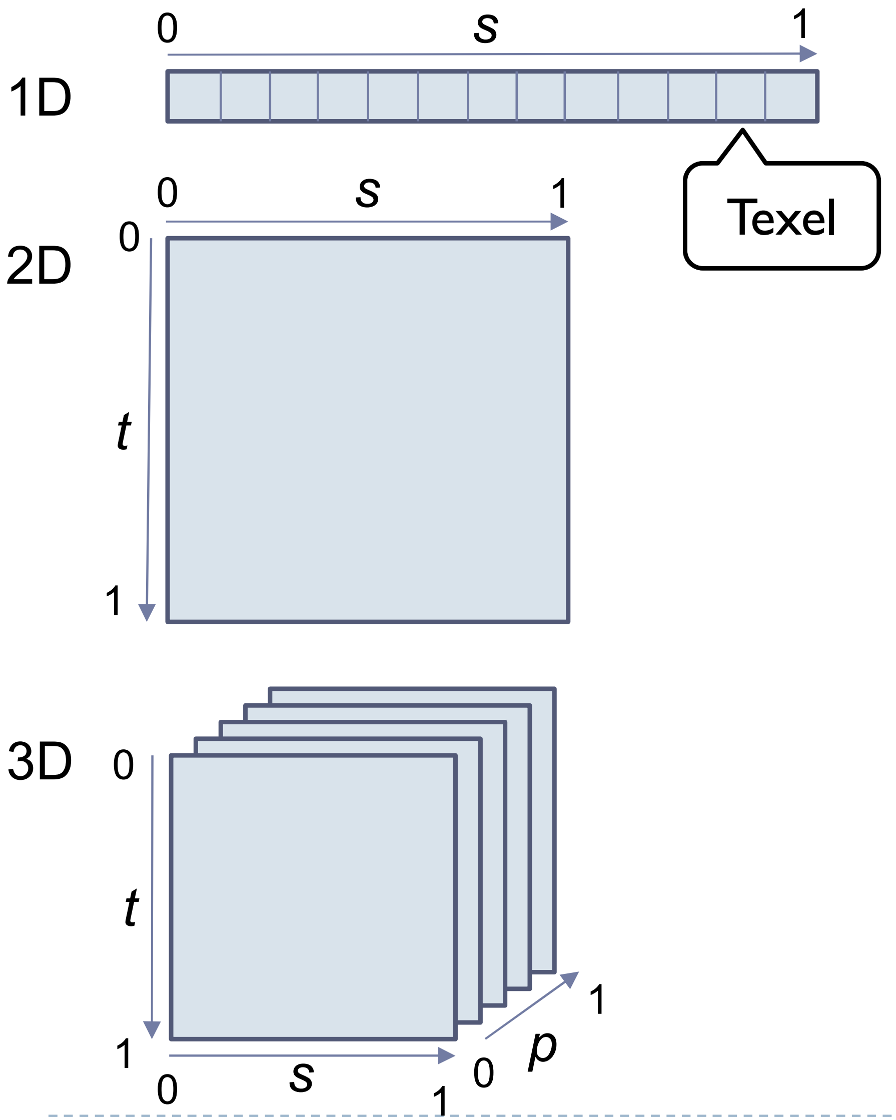
- To render a single object with OpenGL

1. `glUseProgram()` – to activate vertex & fragment shaders
2. `glVertexAttribPointer()` – to indicate which Buffers with vertices and normal should be input to fragment shader
3. `glUniform*()` – to set uniforms (parameters of the fragment/vertex shader)
4. `glBindTexture()` – to bind the texture
5. `glBindVertexArray()` – to bind the vertex array
6. `glDrawElements()` – to queue drawing the geometry
7. Unbind all objects

- OpenGL API is designed around the idea of a state-machine – set the state & queue drawing command

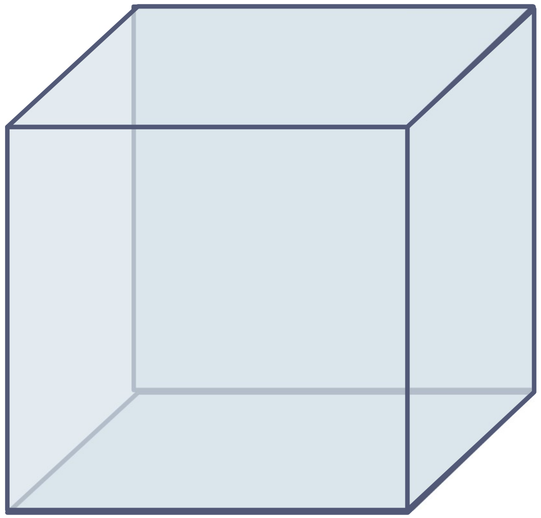
Textures

(Most important) OpenGL texture types

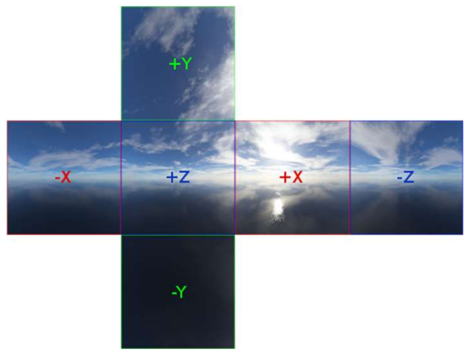


Texture can have any size but the sizes that are powers of two (POT, 2^n) may give better performance.

CUBE_MAP

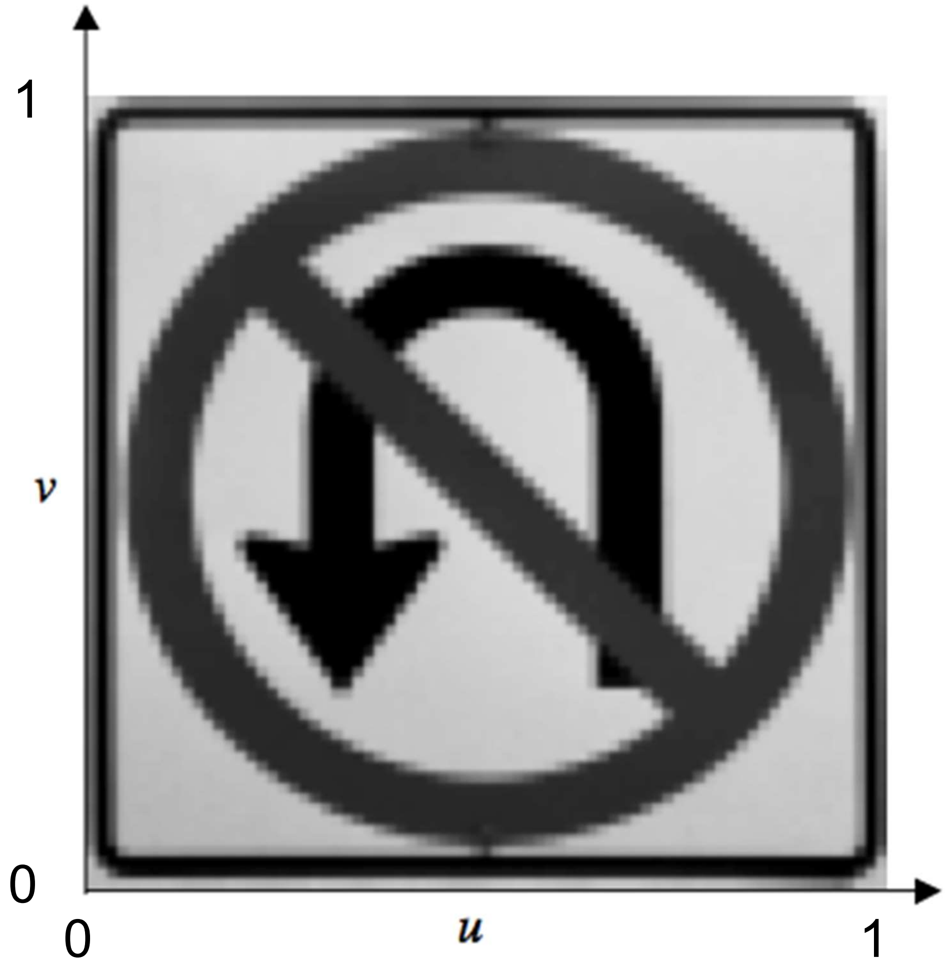


Used for environment mapping



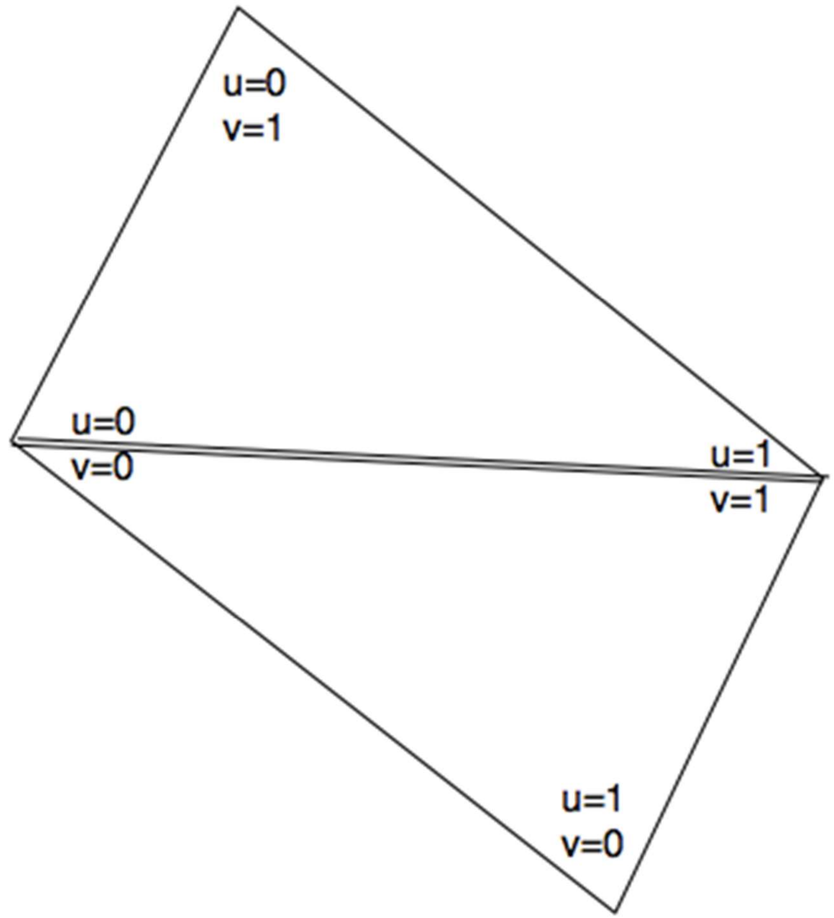
Texture mapping

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



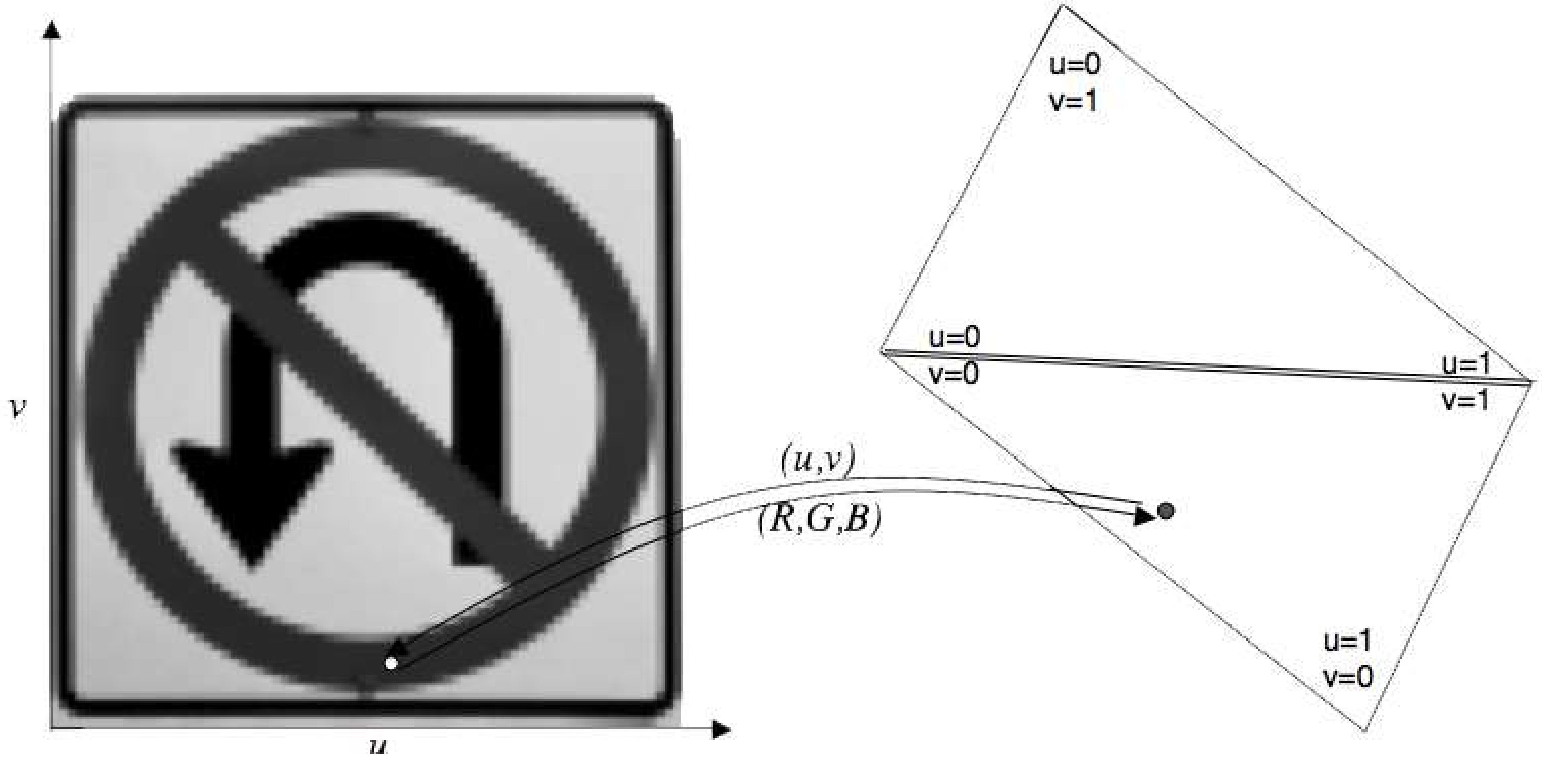
Texture mapping

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

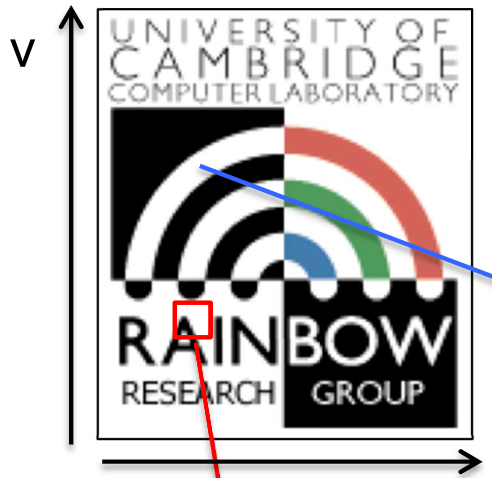


Texture mapping

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

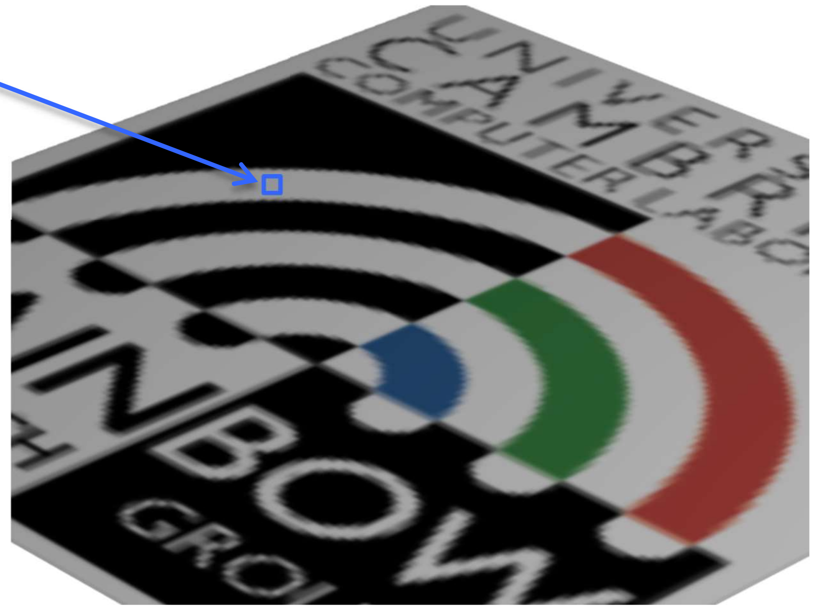


Sampling



Texture

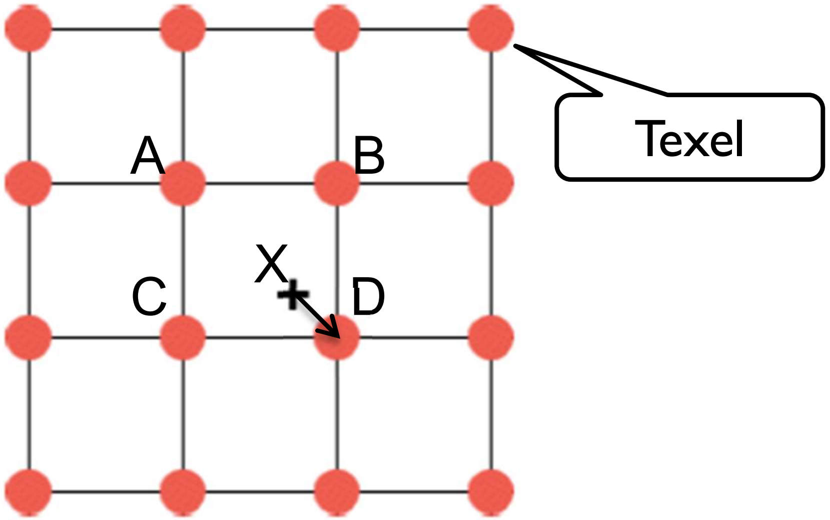
Up-sampling
More pixels than texels
Values need to be interpolated



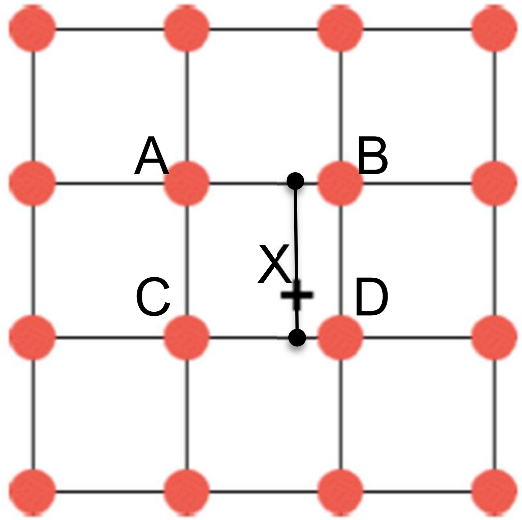
Down-sampling
Fewer pixels than texels
Values need to be averaged over an area of the texture (usually using a mipmap)

Nearest neighbor vs. bilinear interpolation (upsampling)

Nearest neighbour

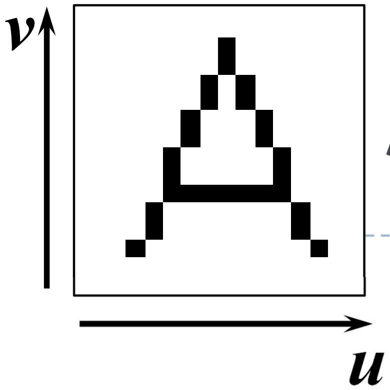


Bilinear interpolation

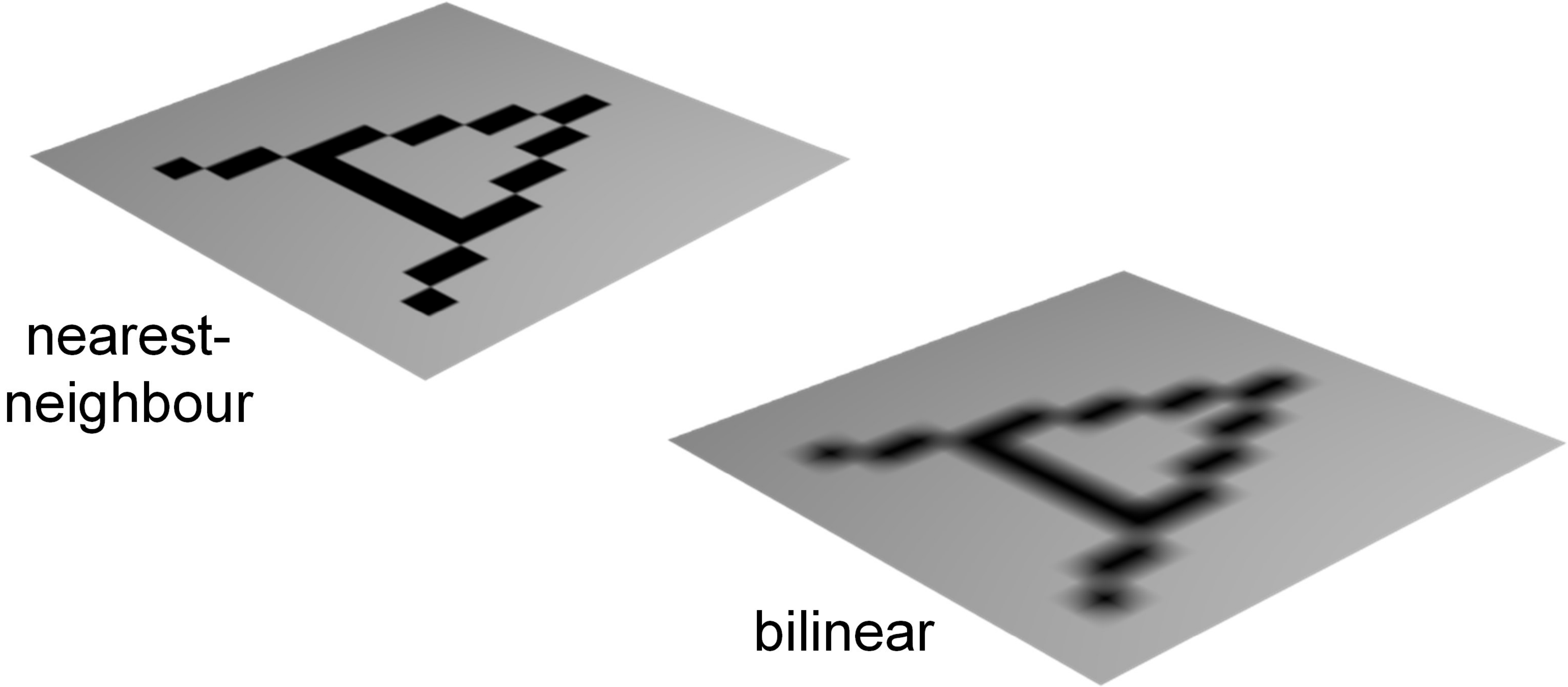


Pick the nearest texel: D

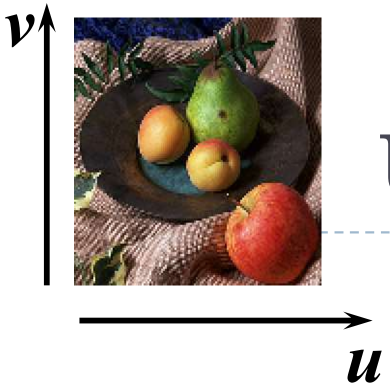
Interpolate first along x-axis between AB and CD, then along y-axis between the interpolated points.



Texture mapping examples



Up-sampling



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

nearest-neighbour

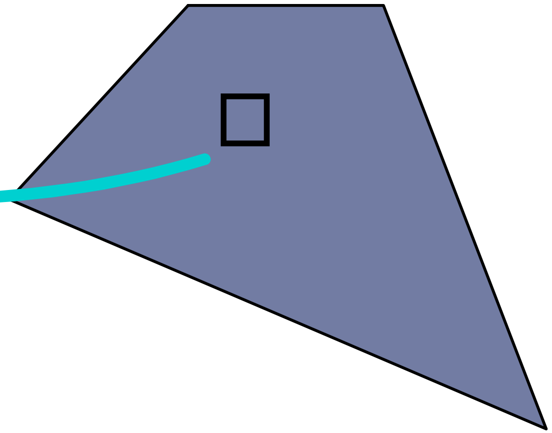
blocky artefacts

bilinear

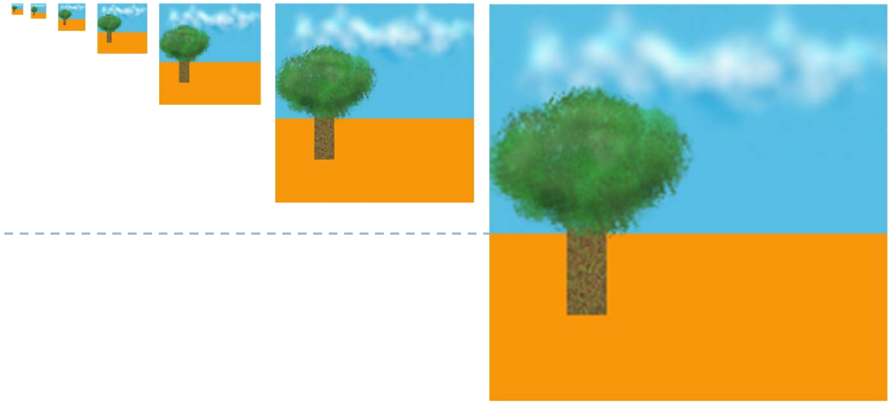
blurry artefacts

Down-sampling

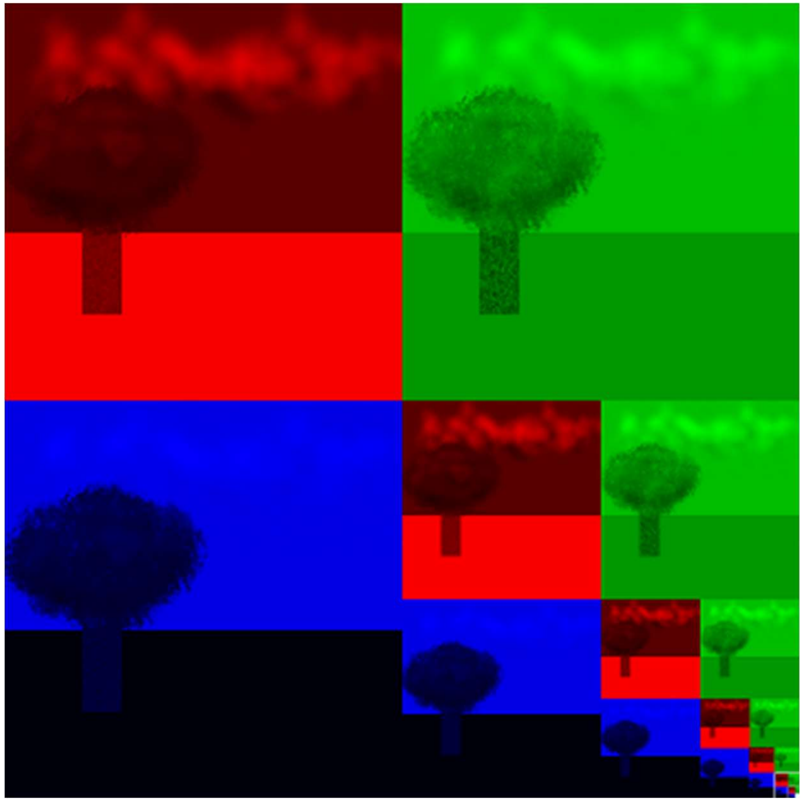
- if the pixel covers quite a large area of the texture, then it will be necessary to average the texture across that area, not just take a sample in the middle of the area



Mipmap



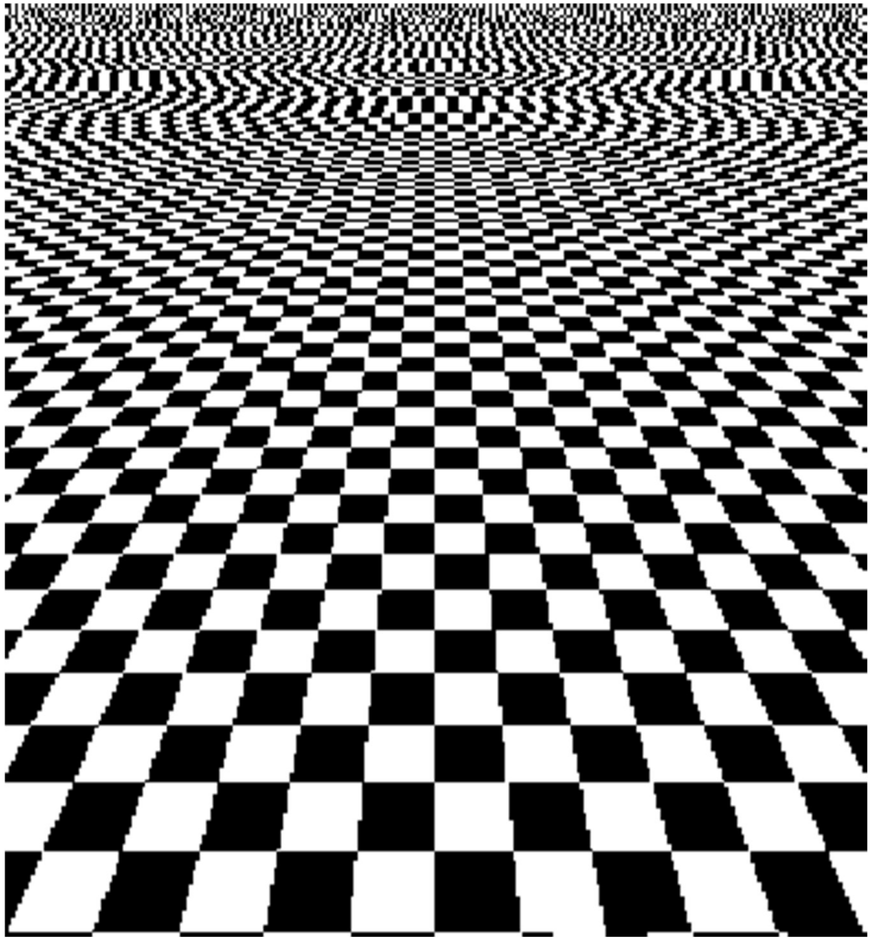
- Textures are often stored at multiple resolutions as a mipmap
  - Each level of the pyramid is half the size of the lower level
  - Mipmap resolution is always power-of-two (1024, 512, 256, 128, ...)
- It provides pre-filtered texture (area-averaged) when screen pixels are larger than the full resolution texels
- Mipmap requires just an additional 1/3 of the original texture size to store
- OpenGL can generate a mipmap with `glGenerateMipmap(GL_TEXTURE_2D)`



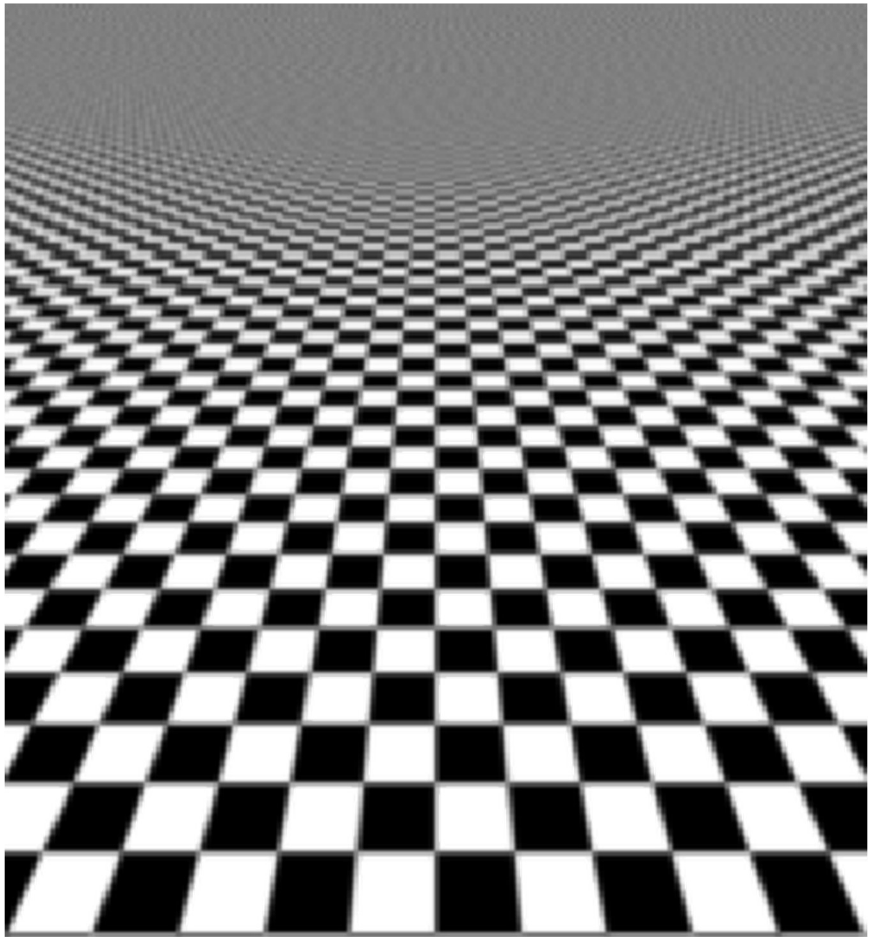
This image is an illustration showing only 1/3 increase in storeage. Mipmaps are stored differently in the GPU memory.

Down-sampling

without area averaging

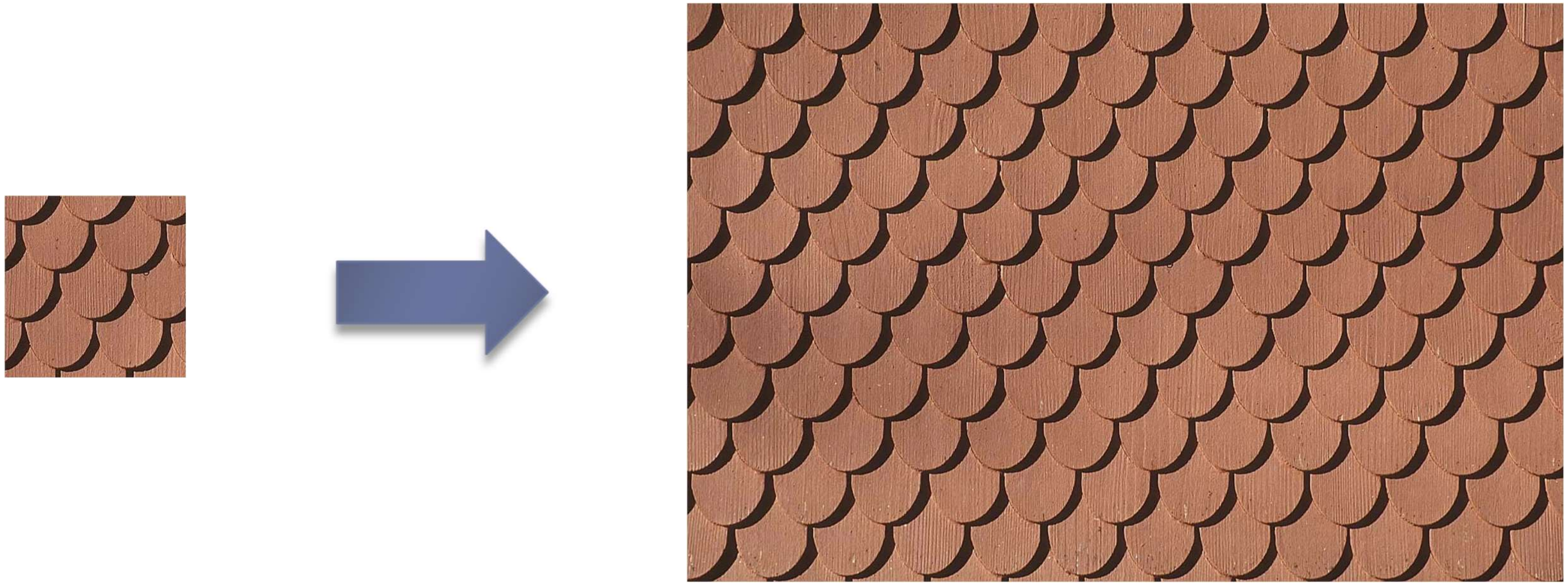


with area averaging



Texture tiling

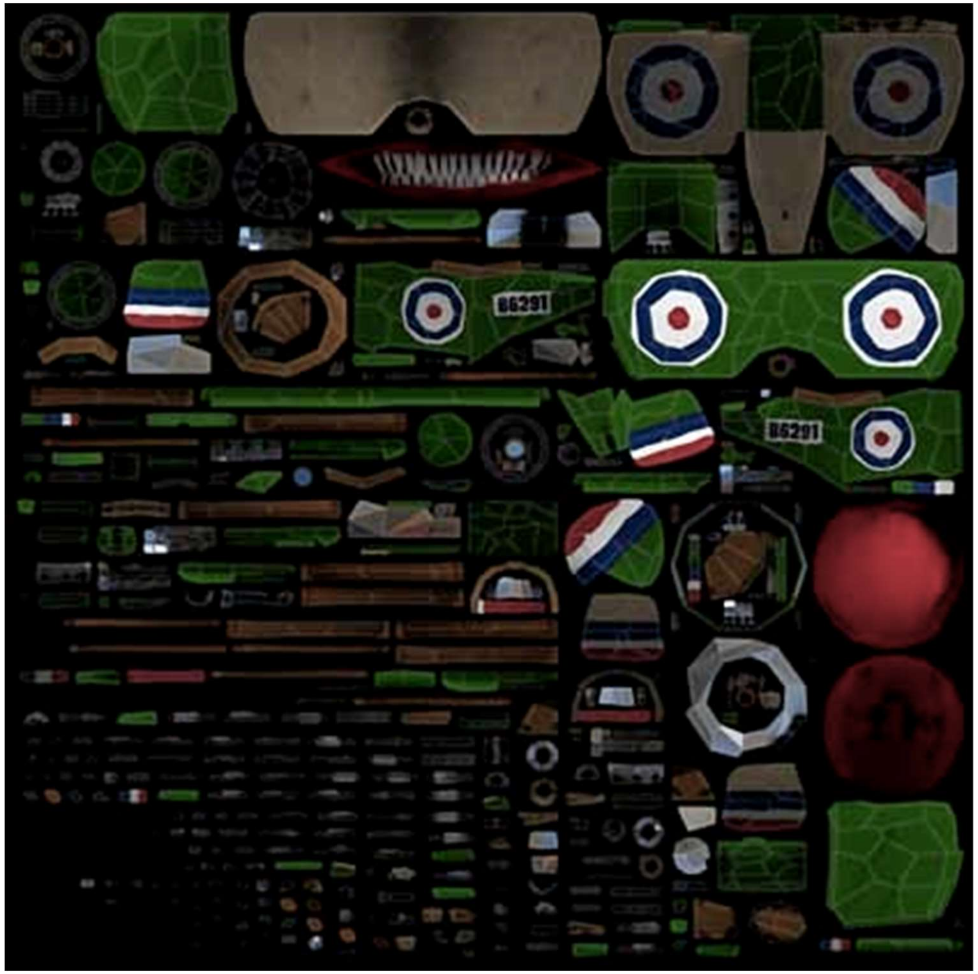
- Repetitive patterns can be represented as texture tiles.
- The texture folds over, so that
  - T(u=1.1, v=0) = T(u=0.1, v=0)



Gimp and other drawing software often offer plugins for creating tiled textures

Multi-surface UV maps

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



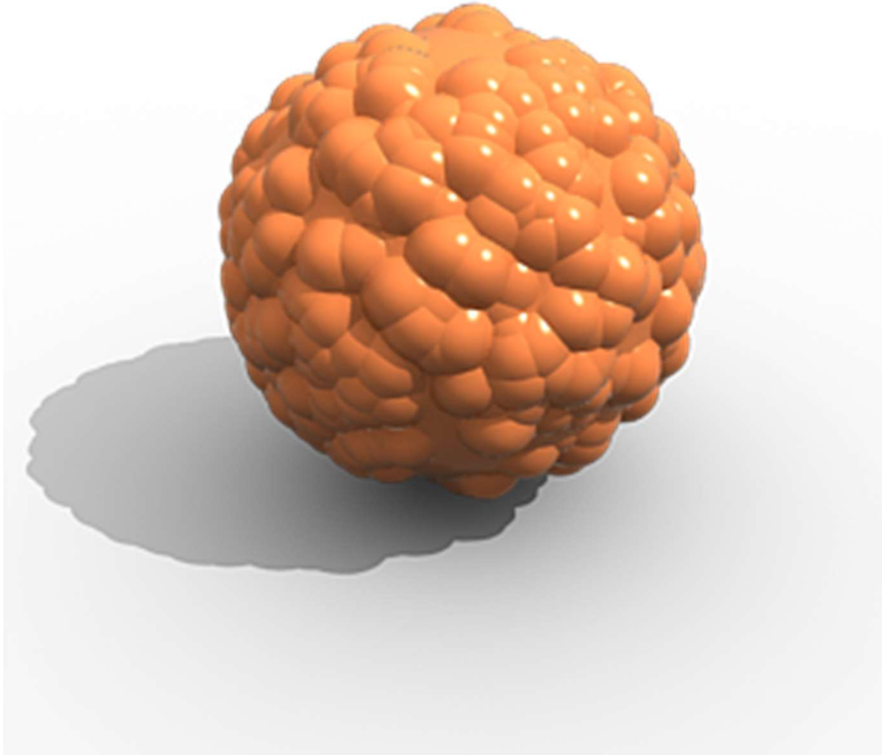
Example from: http://awshub.com/blog/blog/2011/11/01/hi-poly-vs-low-poly/

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

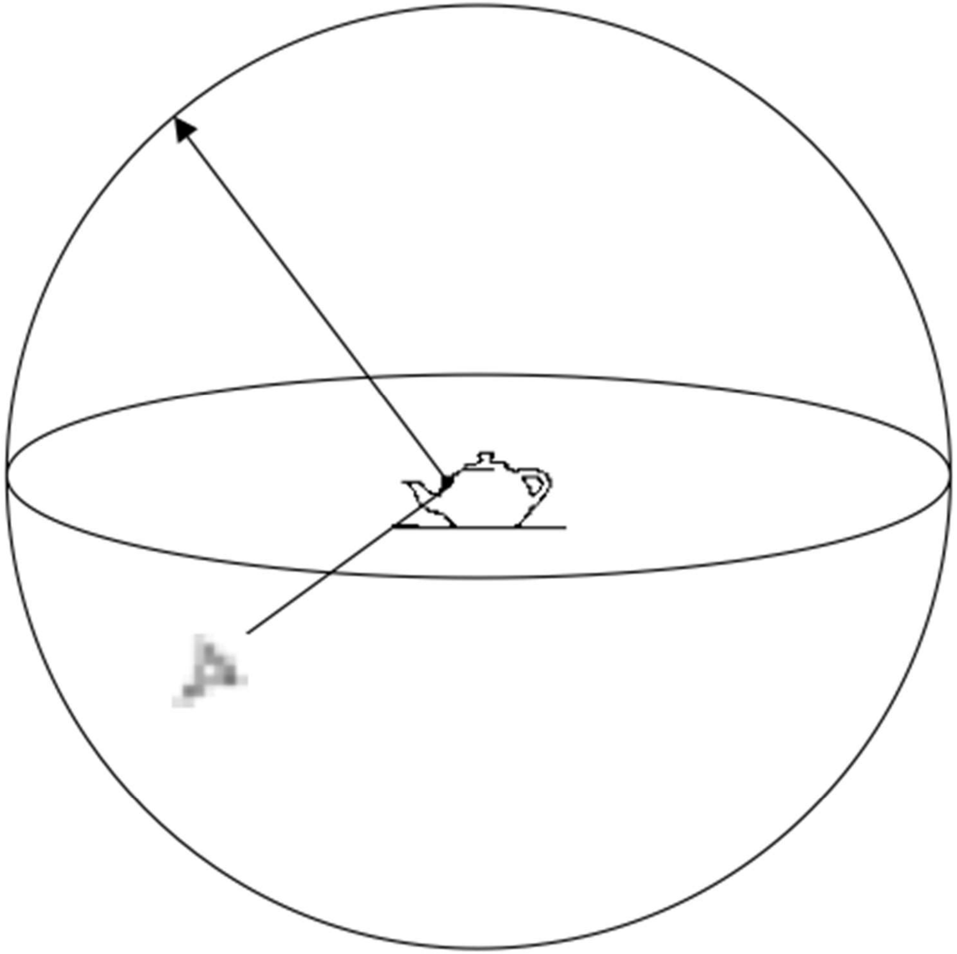
Displacement mapping

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



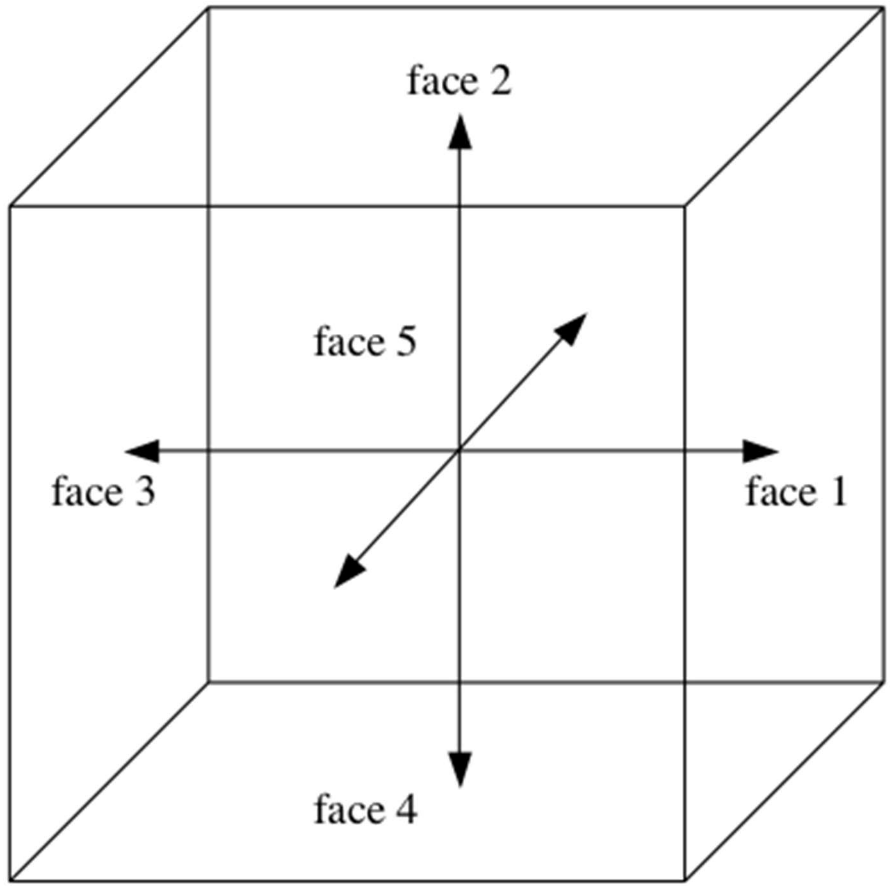
Environment mapping

- To show environment reflected by an object

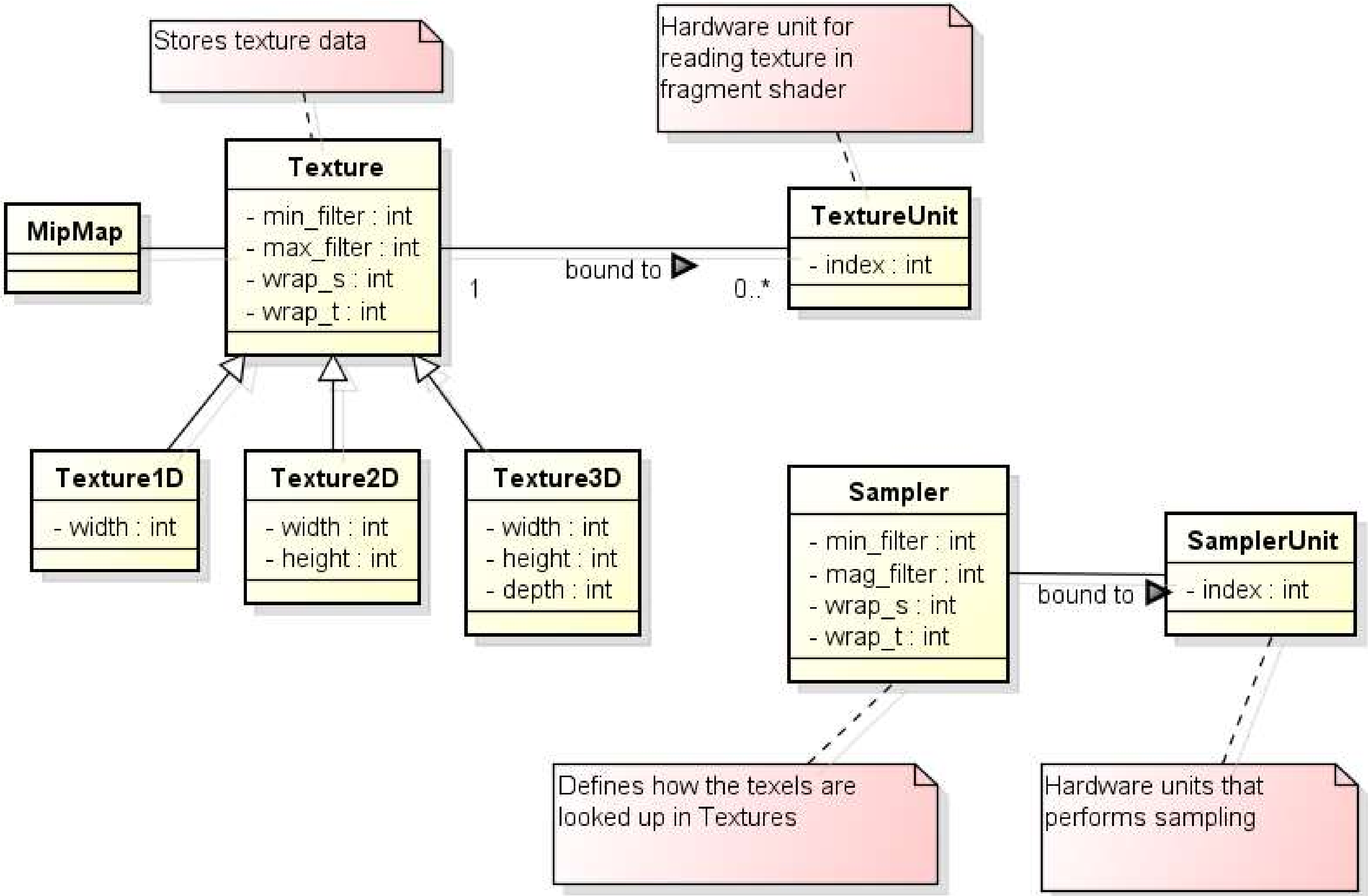


Environment mapping

- Environment cube
- Each face captures environment in that direction



Texture objects in OpenGL



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);
```

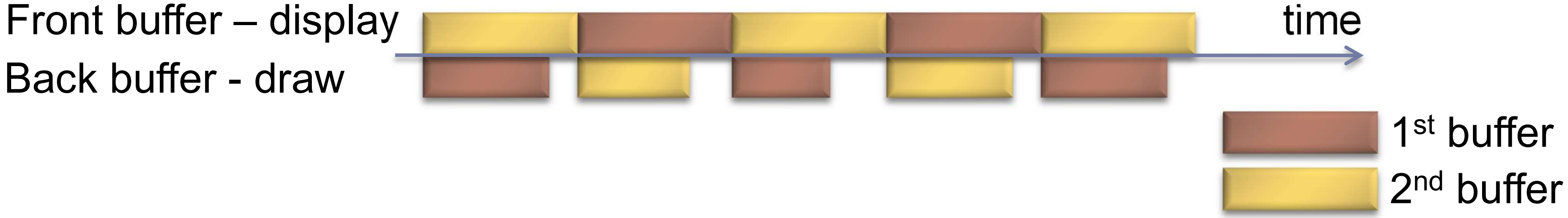
Raster buffers (colour, depth, stencil)

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.	

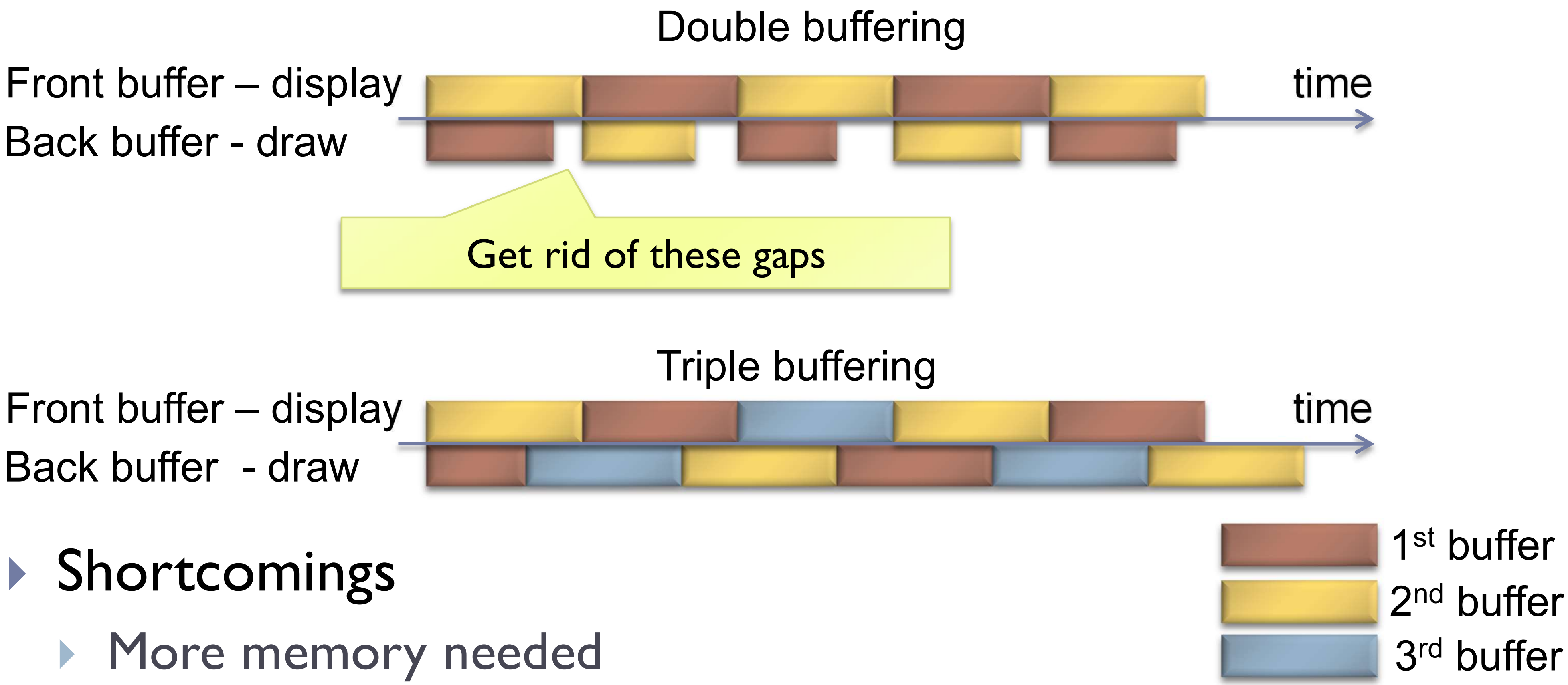
Double buffering

- 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

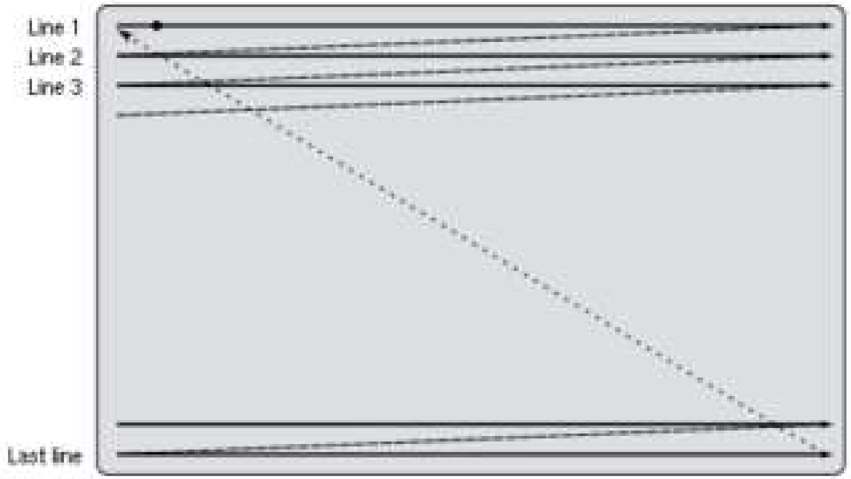


- Shortcomings
  - More memory needed
  - Higher delay between drawing and displaying a frame

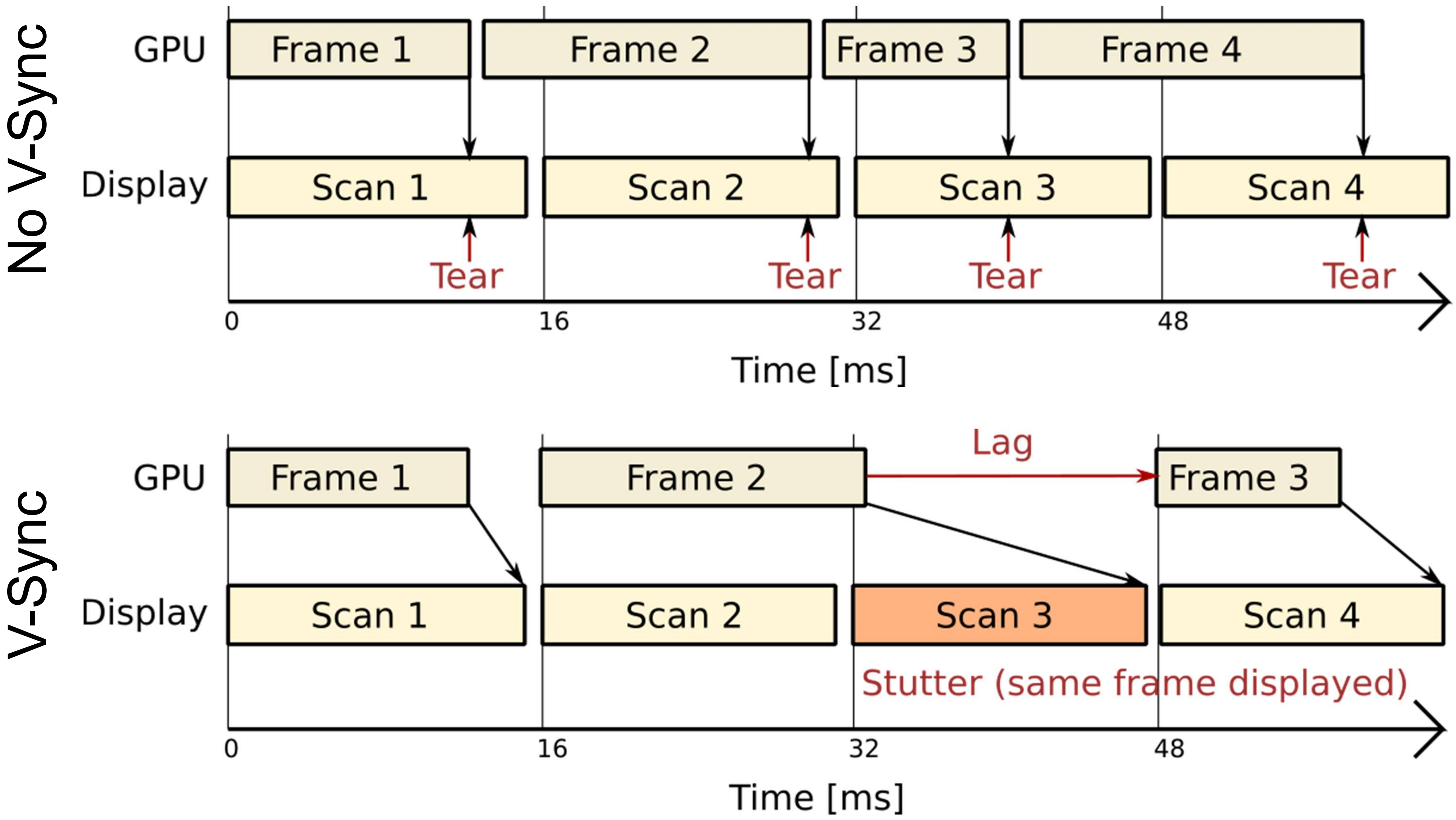
Vertical Synchronization: V-Sync

- Pixels are copied from colour buffer to monitor row-by-row
- 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 row of pixels is copied to the display.

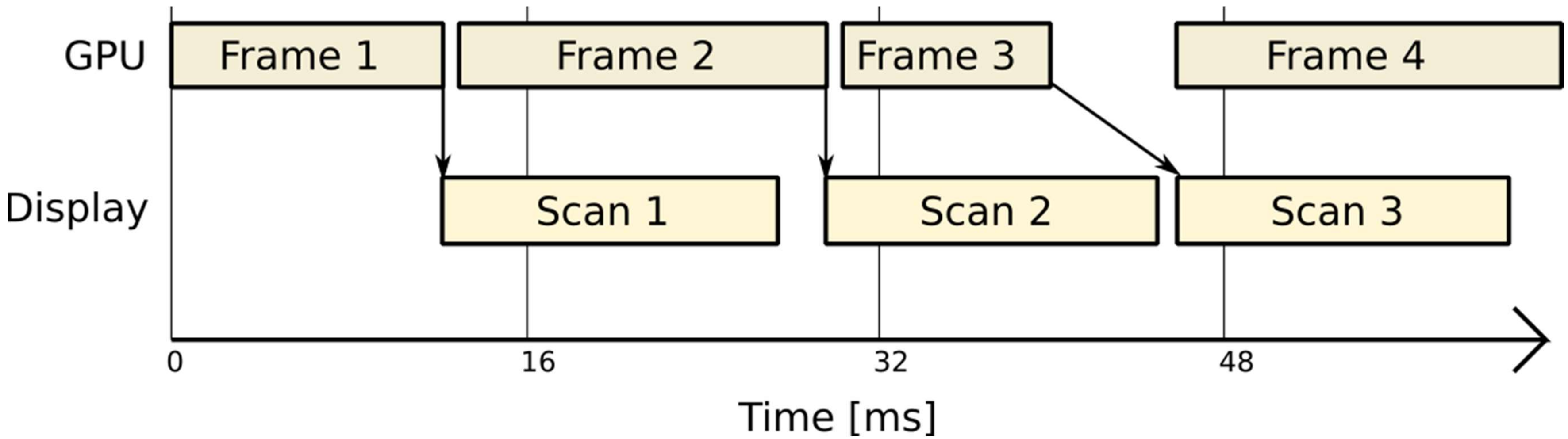


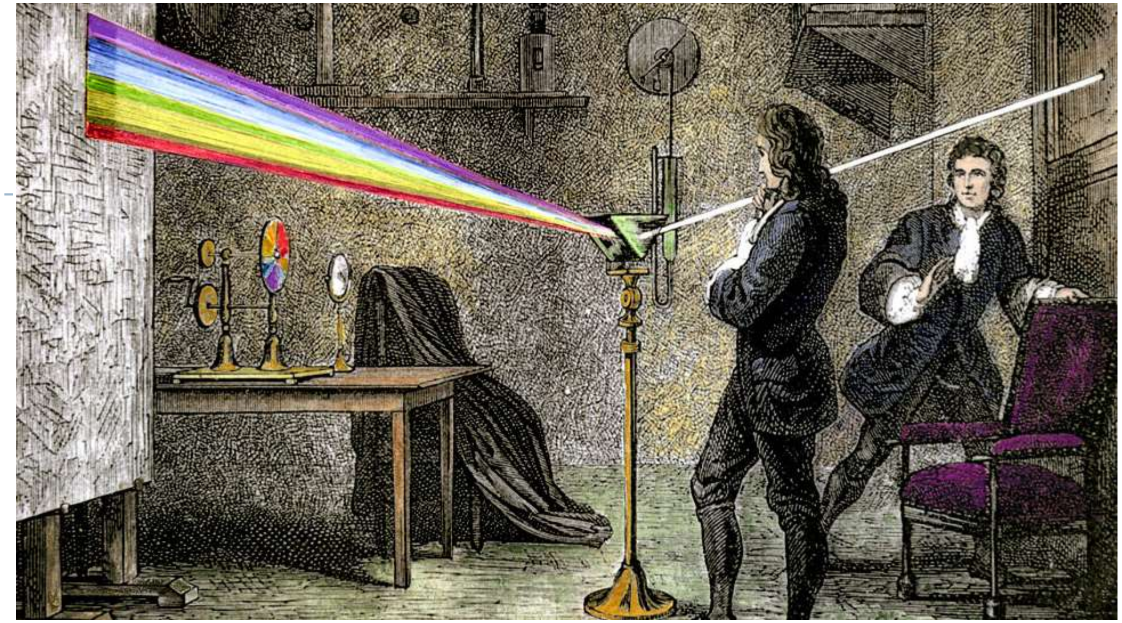
No V-Sync vs. V-Sync



FreeSync (AMD) & G-Sync (Nvidia)

- Adaptive sync
  - Graphics card controls timing of the frames on the display
  - Can save power for 30fps video of when the screen is static
  - Can reduce lag for real-time graphics

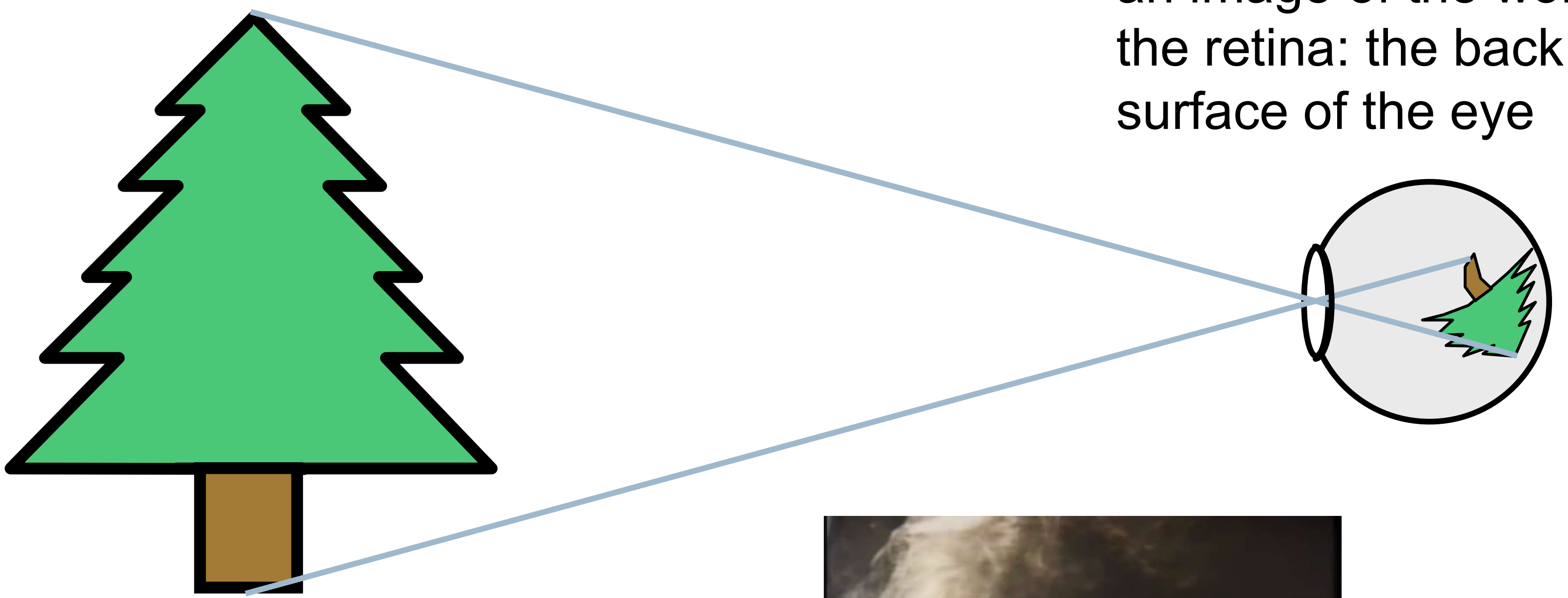




Vision, colour and colour spaces

The workings of the human visual system

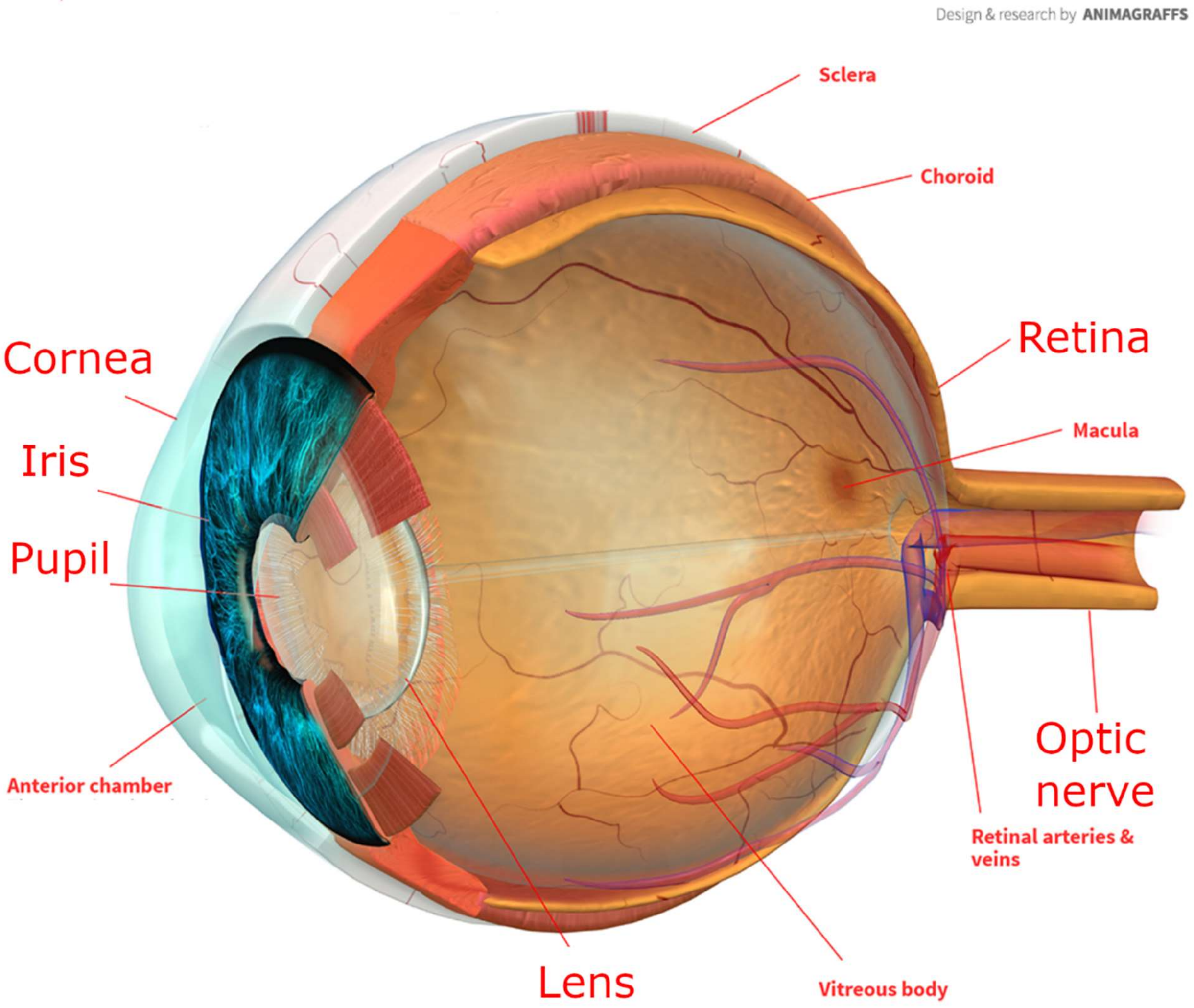
- to understand the requirements of displays (resolution, quantisation and colour) we need to know how the human eye works...



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

Inverted vision experiment

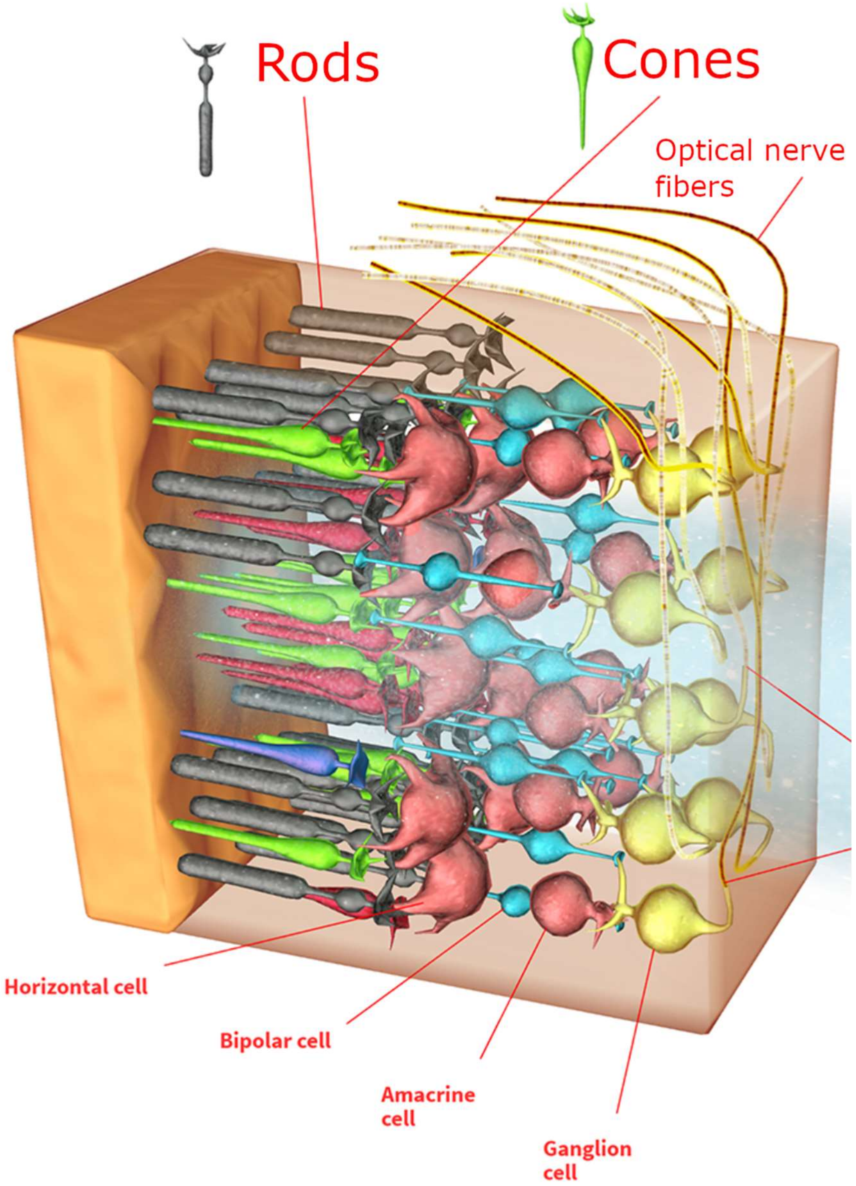
Structure of the human eye



See Animagraffs web page for an animated visualization
https://animagraffs.com/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
- cornea and lens focus the light on the retina
- pupil shrinks and expands to control the amount of light

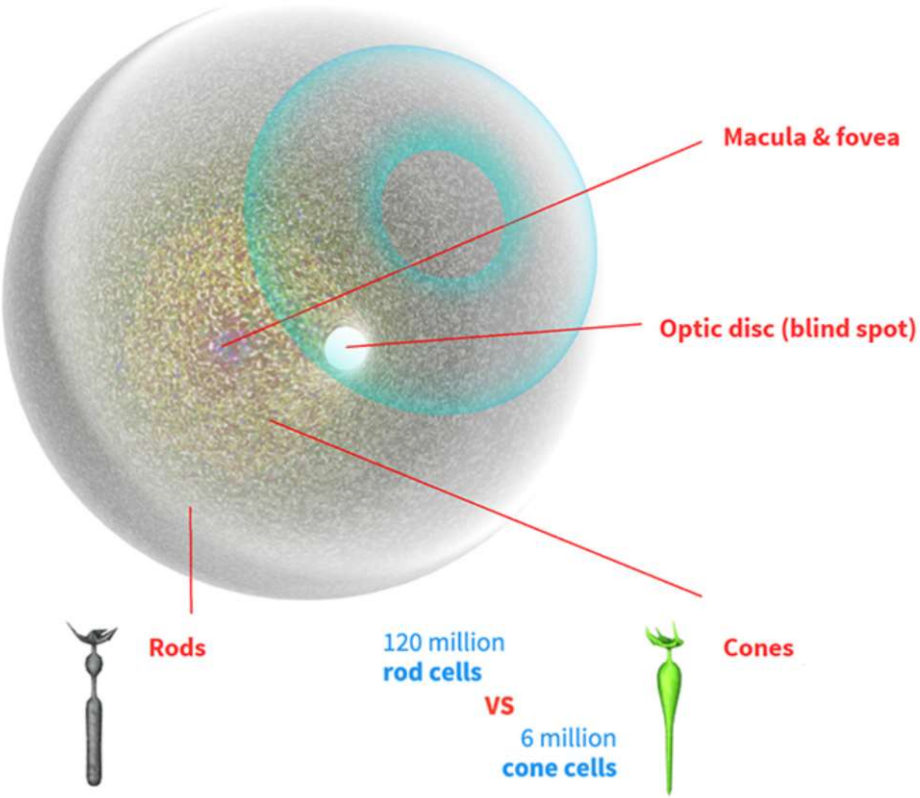
Retina, cones and rods

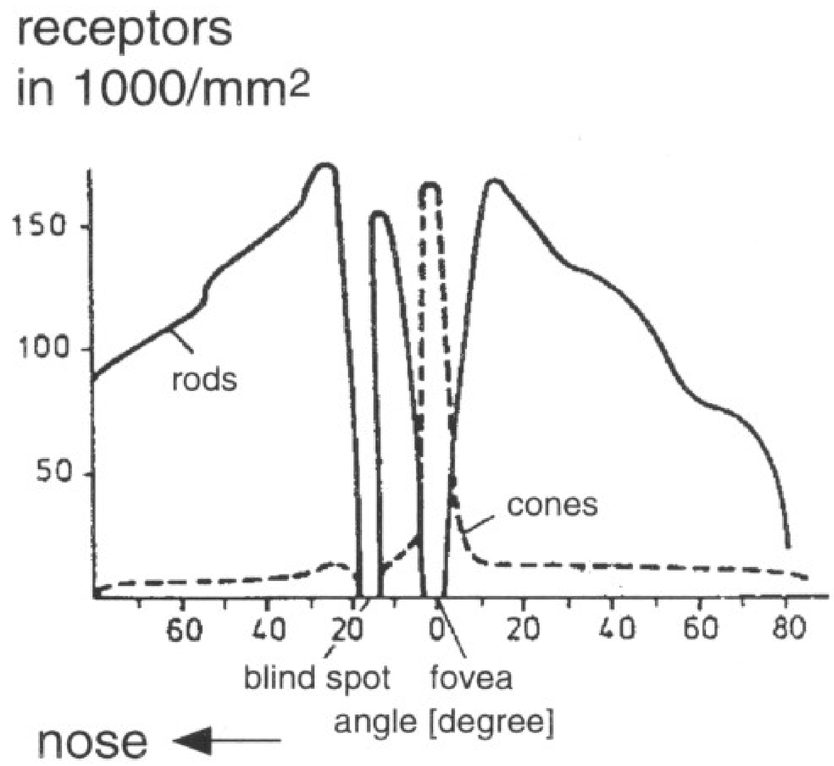


- 2 classes of photoreceptors
  - Cones are responsible for day-light vision and colour perception
    - Three types of cones: sensitive to short, medium and long wavelengths
  - Rods are responsible for night vison

Fovea, distribution of photoreceptors

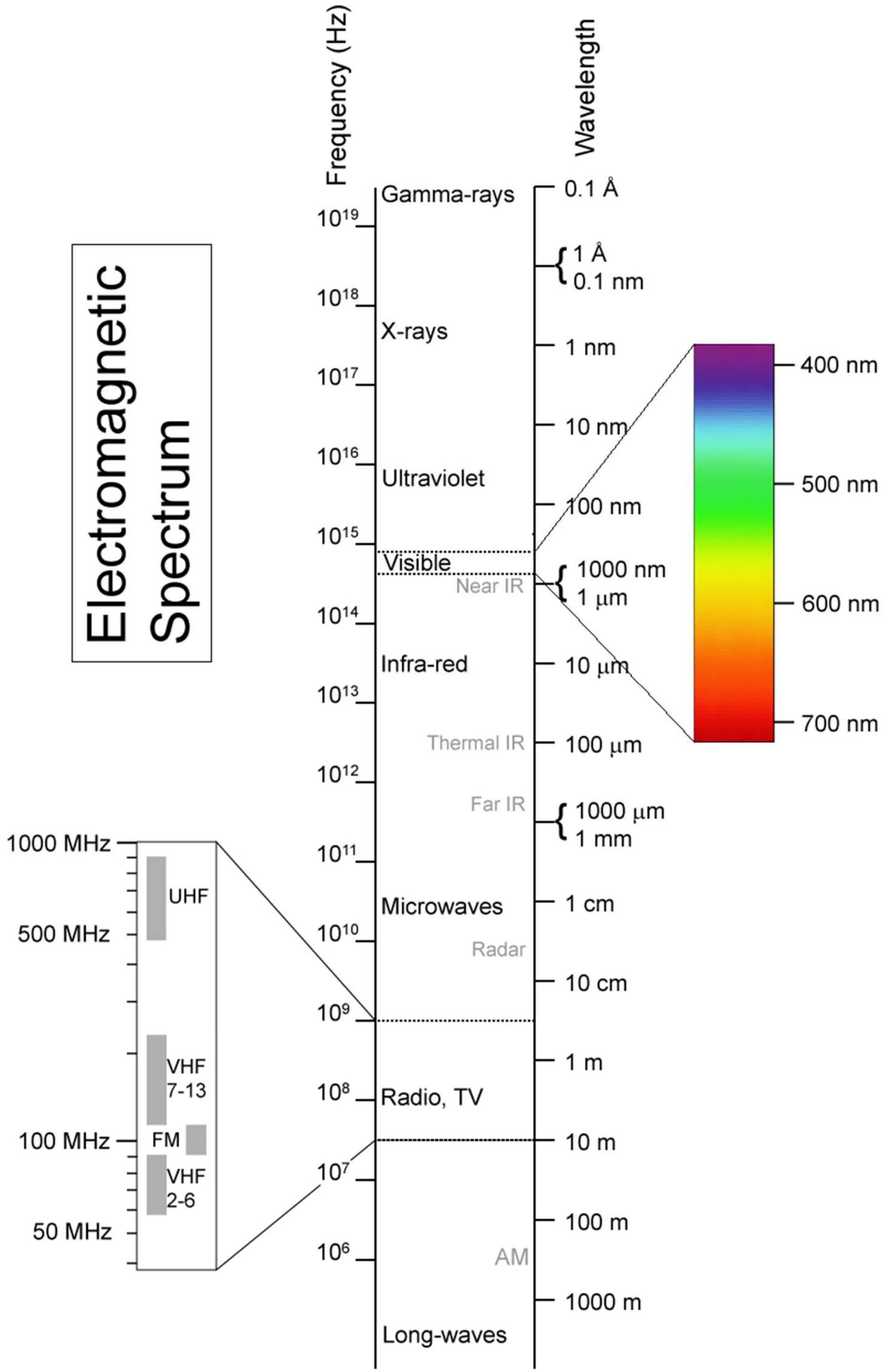
- the fovea is a densely packed region in the centre of the macula
  - contains the highest density of cones
  - provides the highest resolution vision





Electromagnetic spectrum

- Visible light
  - Electromagnetic waves of wavelength in the range 380nm to 730nm
  - Earth's atmosphere lets through a lot of light in this wavelength band
  - Higher in energy than thermal infrared, so heat does not interfere with vision

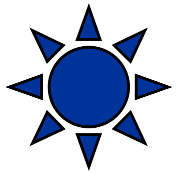


Colour

- There is no physical definition of colour – colour is the result of our perception

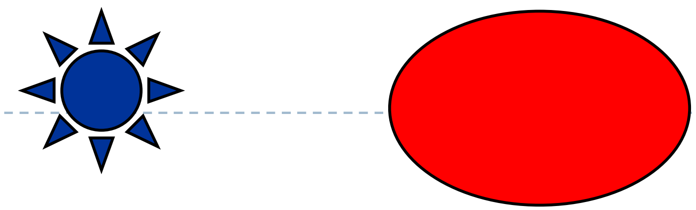
- For emissive displays / objects

colour = perception(spectral_emission)



- For reflective displays / objects

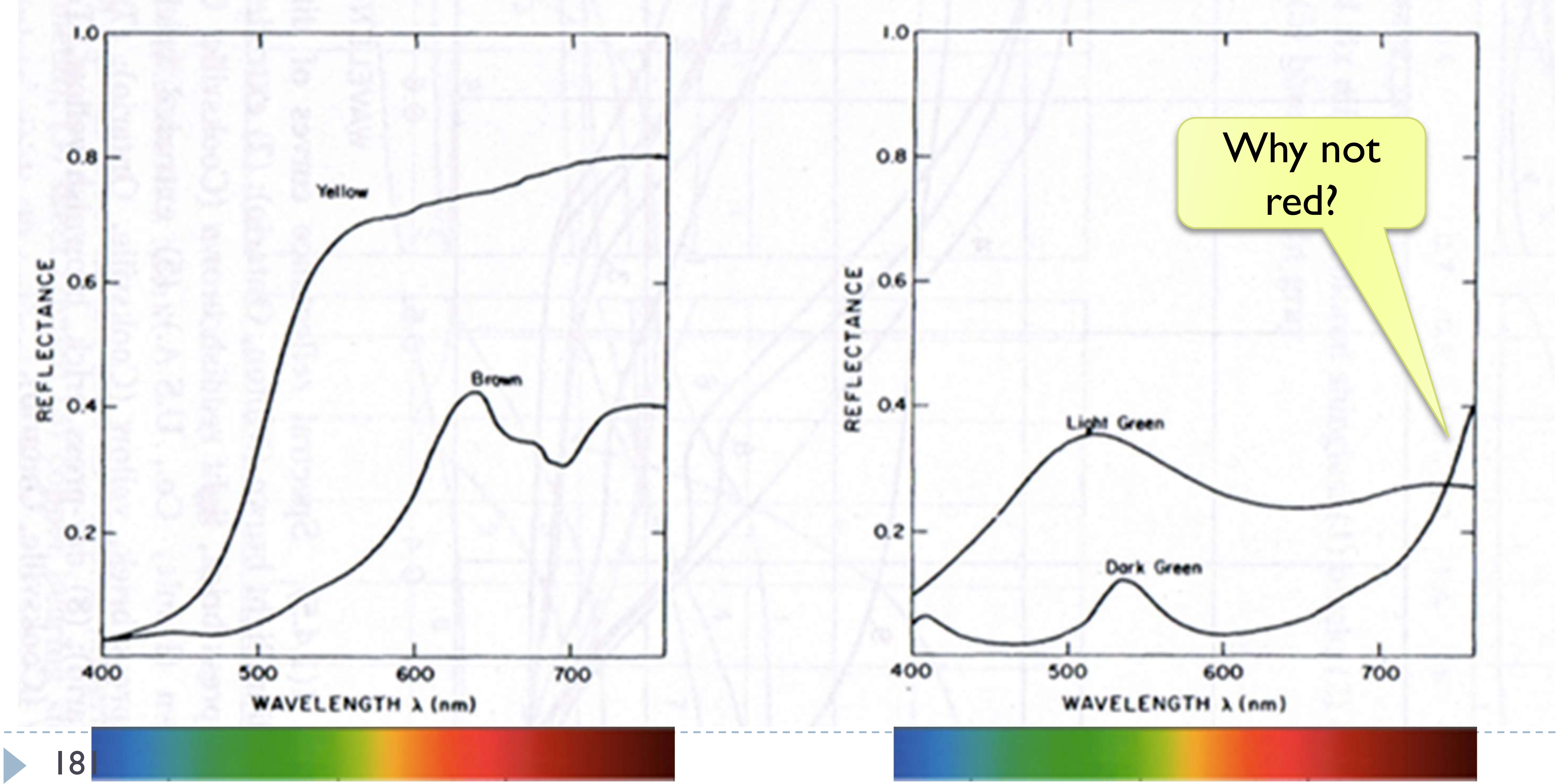
colour = perception(illumination * reflectance)



Reflectance

- Most of the light we see is reflected from objects
- These objects absorb a certain part of the light spectrum

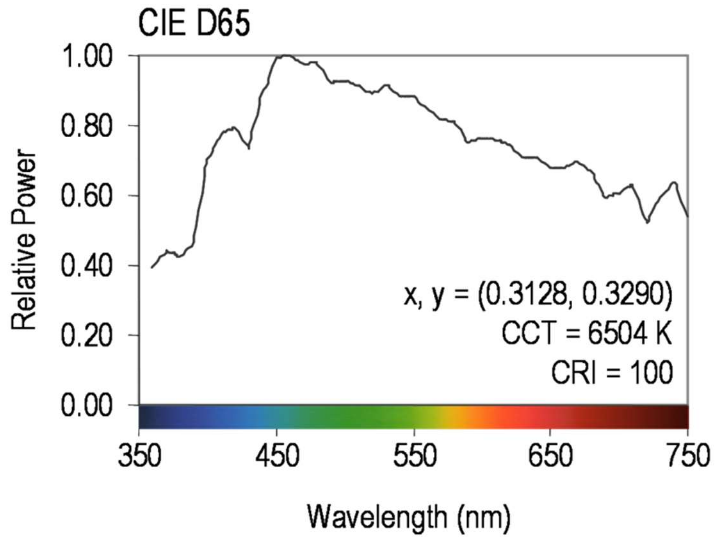
Spectral reflectance of ceramic tiles

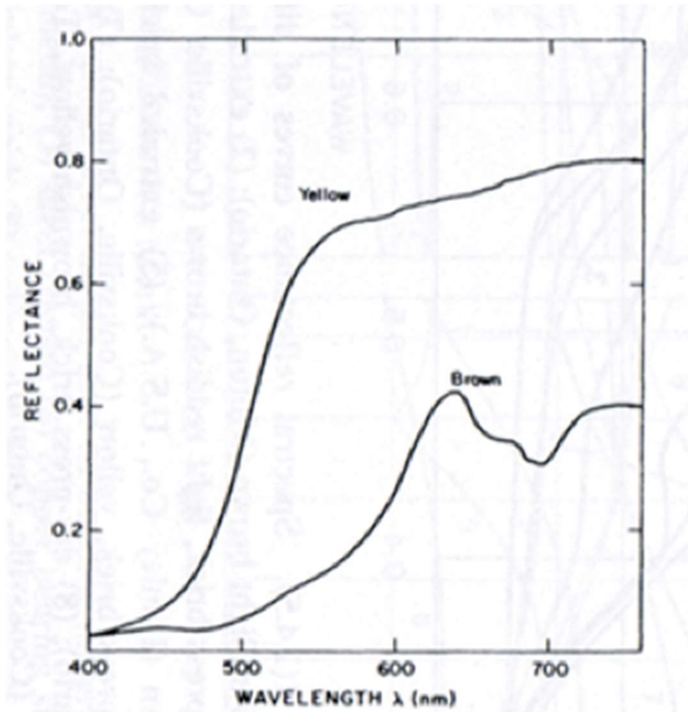


Reflected light

$$L(\lambda) = I(\lambda)R(\lambda)$$

- Reflected light = illumination × reflectance

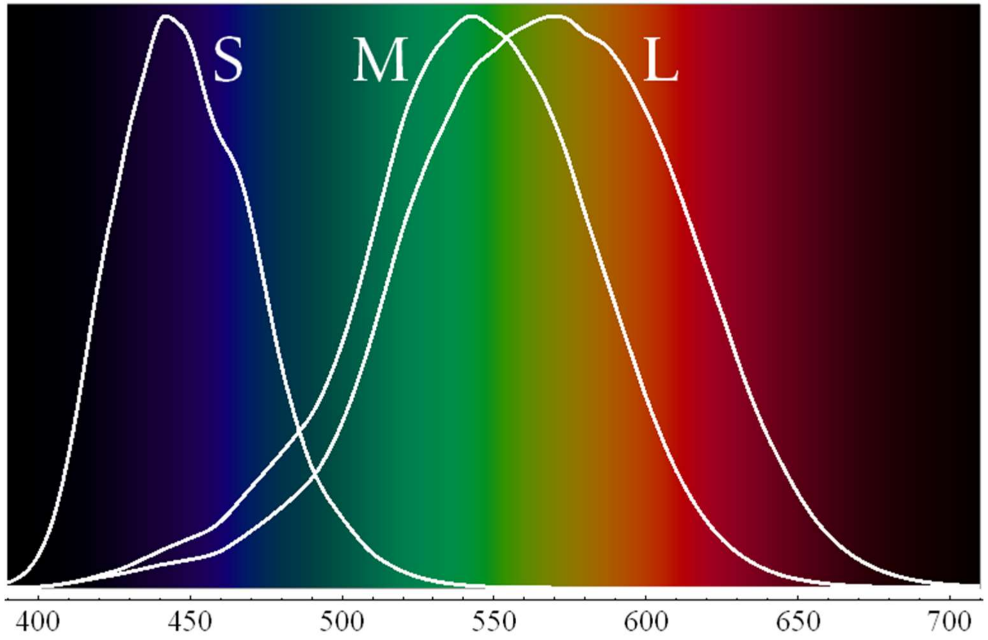




The same object may appear to have different color under different illumination.

Colour vision

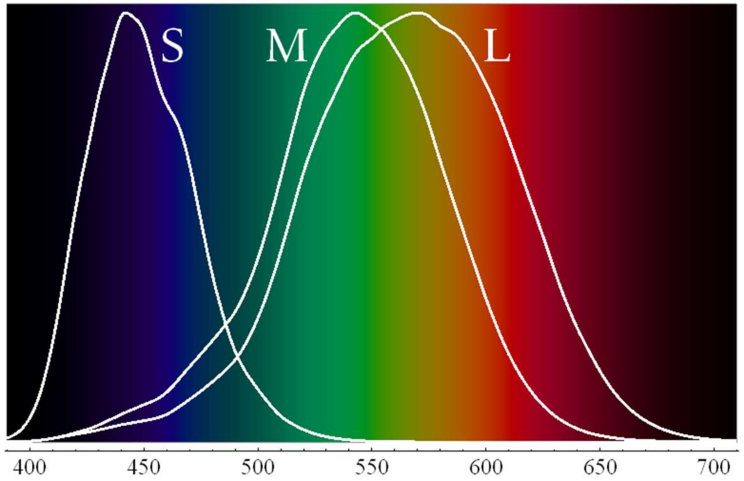
- Cones are the photreceptors responsible for color vision
  - Only daylight, we see no colors when there is not enough light
- Three types of cones
  - S – sensitive to short wavelengths
  - M – sensitive to medium wavelengths
  - L – sensitive to long wavelengths

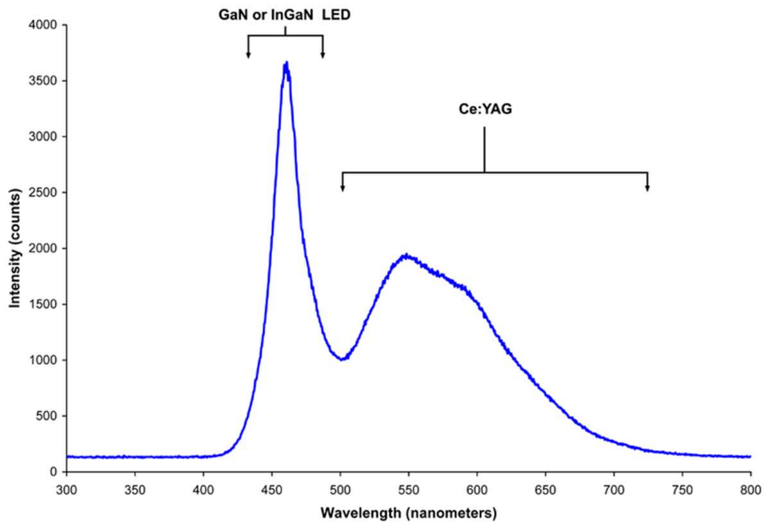


Sensitivity curves – probability that a photon of that wavelengths will be absorbed by a photoreceptor. S,M and L curves are normalized in this plot.

Perceived light

- cone response = sum(sensitivity × reflected light)





Although there is an infinite number of wavelengths, we have only three photoreceptor types to sense differences between light spectra

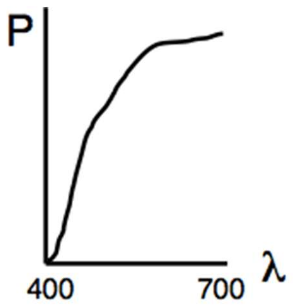
Formally

$$R_S = \int_{380}^{730} S_S(\lambda) \cdot L(\lambda) d\lambda$$

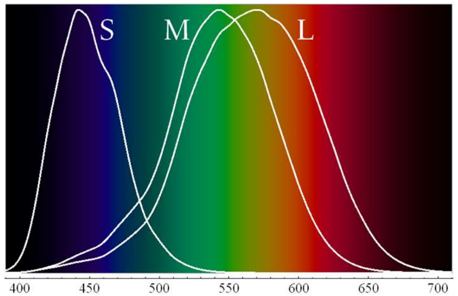
Index S for S-cones

Metamers

- Even if two light spectra are different, they may appear to have the same colour
- The light spectra that appear to have the same colour are called **metamers**
- Example:

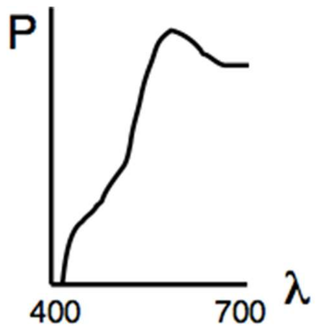


*

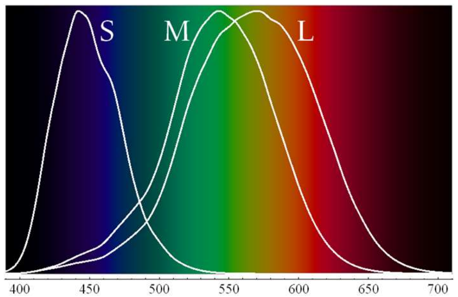


$= [L_1, M_1, S_1]$

$\parallel$



*

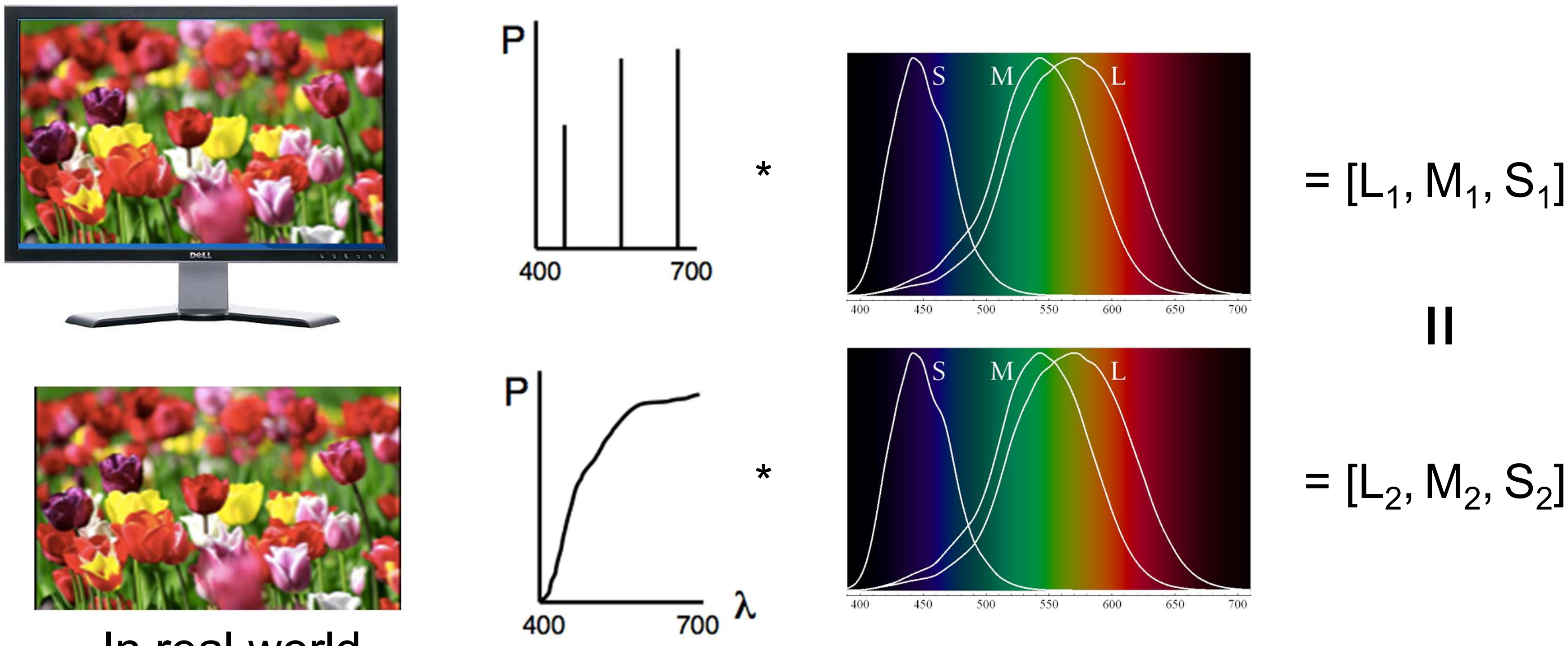


$= [L_2, M_2, S_2]$

Practical application of metamerism

- Displays do not emit the same light spectra as real-world objects
- Yet, the colours on a display look almost identical

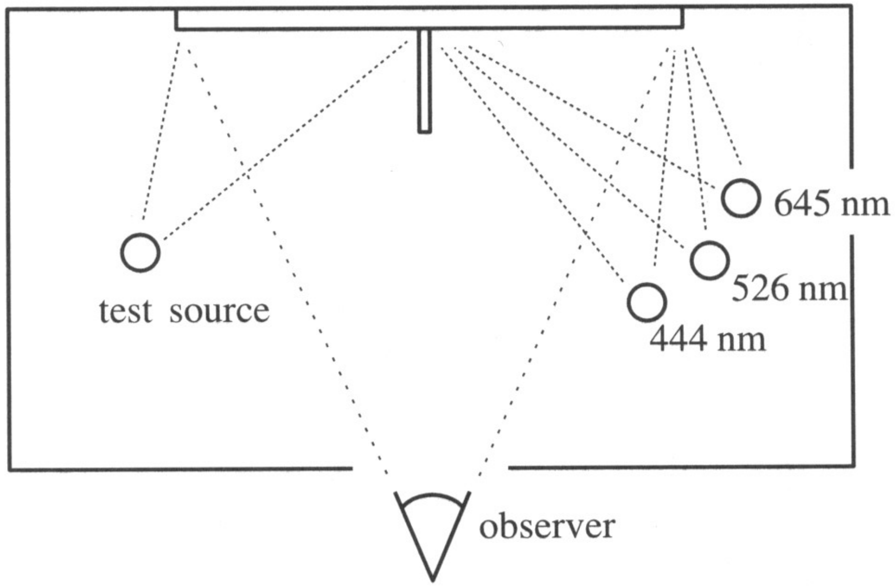
On the display

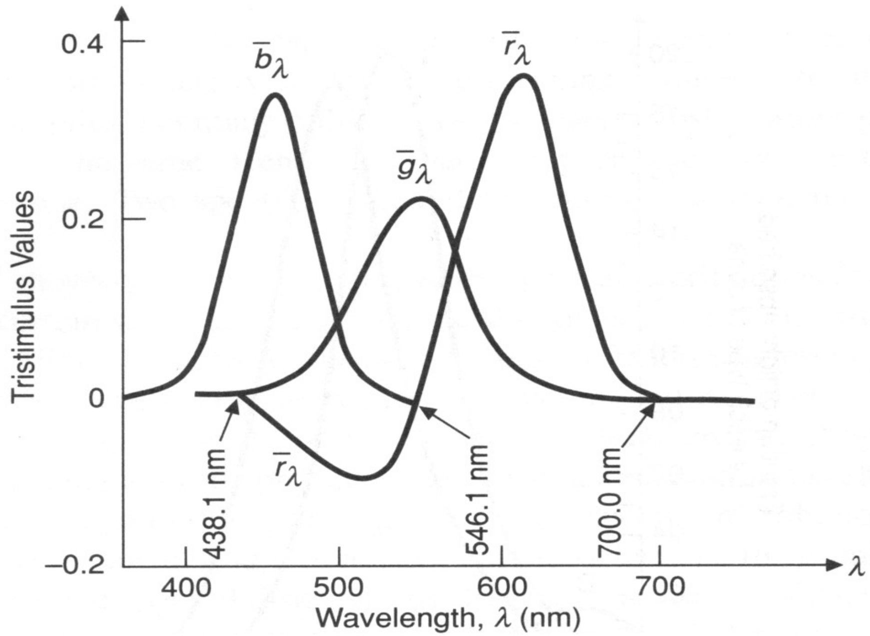


In real world

Tristimulus Colour Representation

- Observation
  - Any colour can be matched using three linear independent reference colours
  - May require "negative" contribution to test colour
  - Matching curves describe the value for matching mono-chromatic spectral colours of equal intensity
    - With respect to a certain set of primary colours



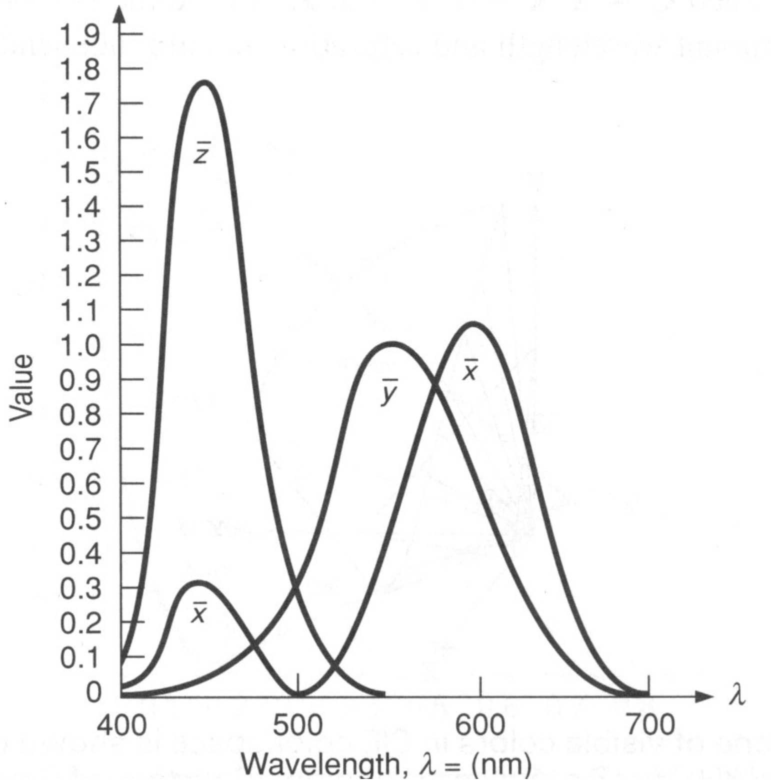


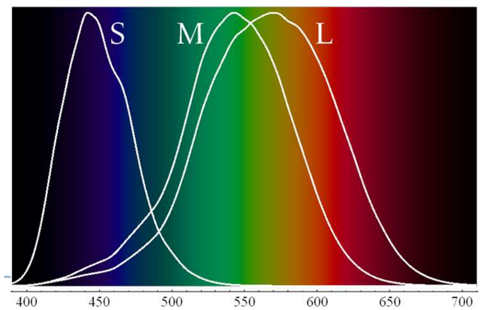
Standard Colour Space CIE-XYZ

- CIE Experiments [Guild and Wright, 1931]
  - Colour matching experiments
  - Group ~12 people with normal colour vision
  - 2 degree visual field (fovea only)
  - Basis for CIE XYZ 1931 colour matching functions
- CIE 2006 XYZ
  - Derived from LMS color matching functions by Stockman & Sharpe
  - S-cone response differs the most from CIE 1931
- CIE-XYZ Colour Space
  - Goals
    - Abstract from concrete primaries used in experiment
    - All matching functions are positive
    - Primary „Y" is roughly proportionally to light intensity (luminance)

Standard Colour Space CIE-XYZ

- Standardized imaginary primaries CIE XYZ (1931)
  - Could match all physically realizable colour stimuli
  - Y is roughly equivalent to luminance
    - Shape similar to luminous efficiency curve
  - Monochromatic spectral colours form a curve in 3D XYZ-space





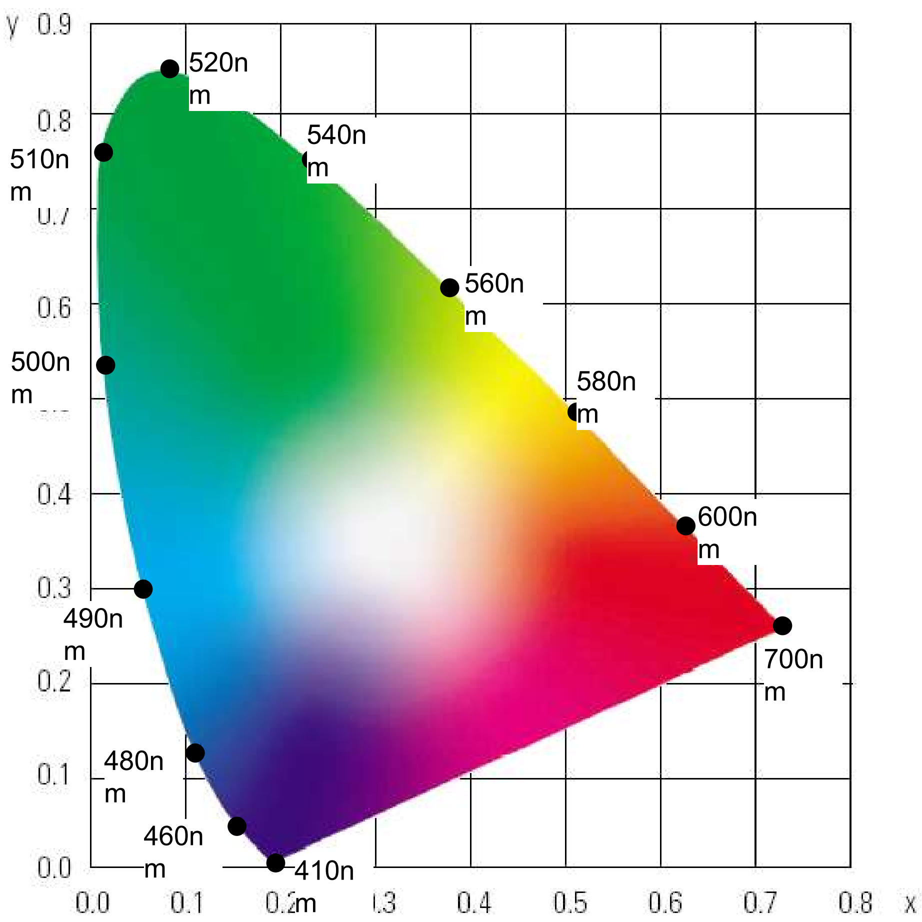
Cone sensitivity curves can be obtained by a linear transformation of CIE XYZ

CIE chromaticity diagram

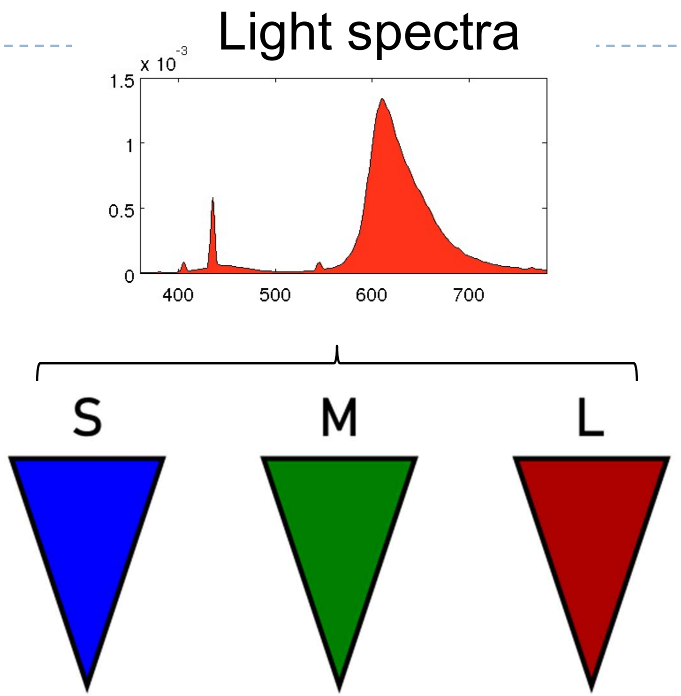
- *chromaticity* values are defined in terms of x, y, z

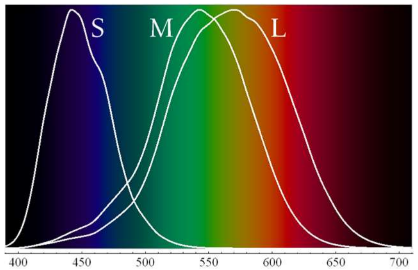
$$x = \frac{X}{X+Y+Z}, \quad y = \frac{Y}{X+Y+Z}, \quad z = \frac{Z}{X+Y+Z} \qquad x + y + z = 1$$

  - 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

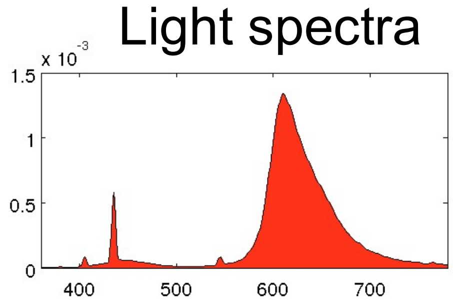


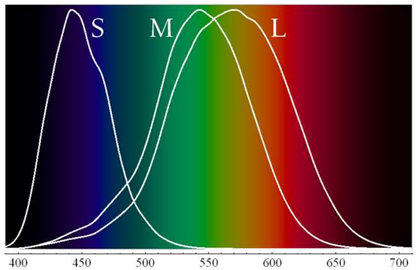
Achromatic/chromatic vision mechanisms

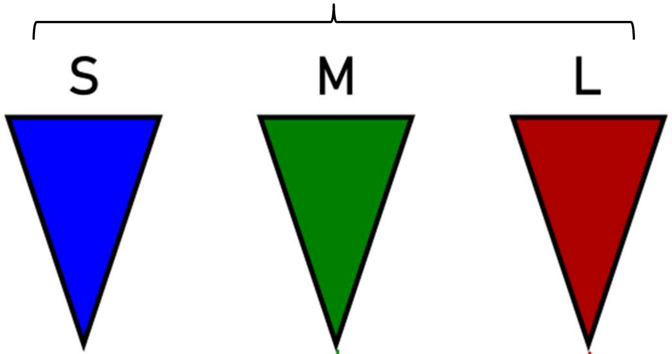




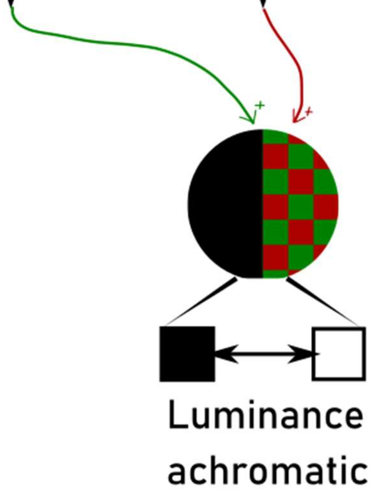
Achromatic/chromatic vision mechanisms

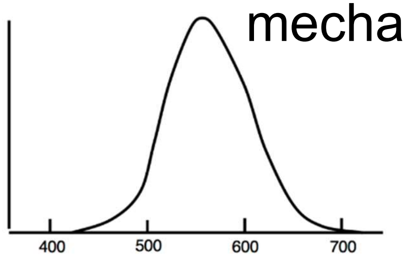






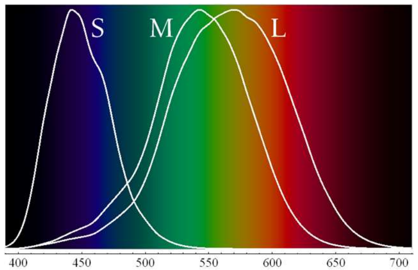
Luminance does NOT explain the brightness of light! [Koenderink et al. Vision Research 2016]

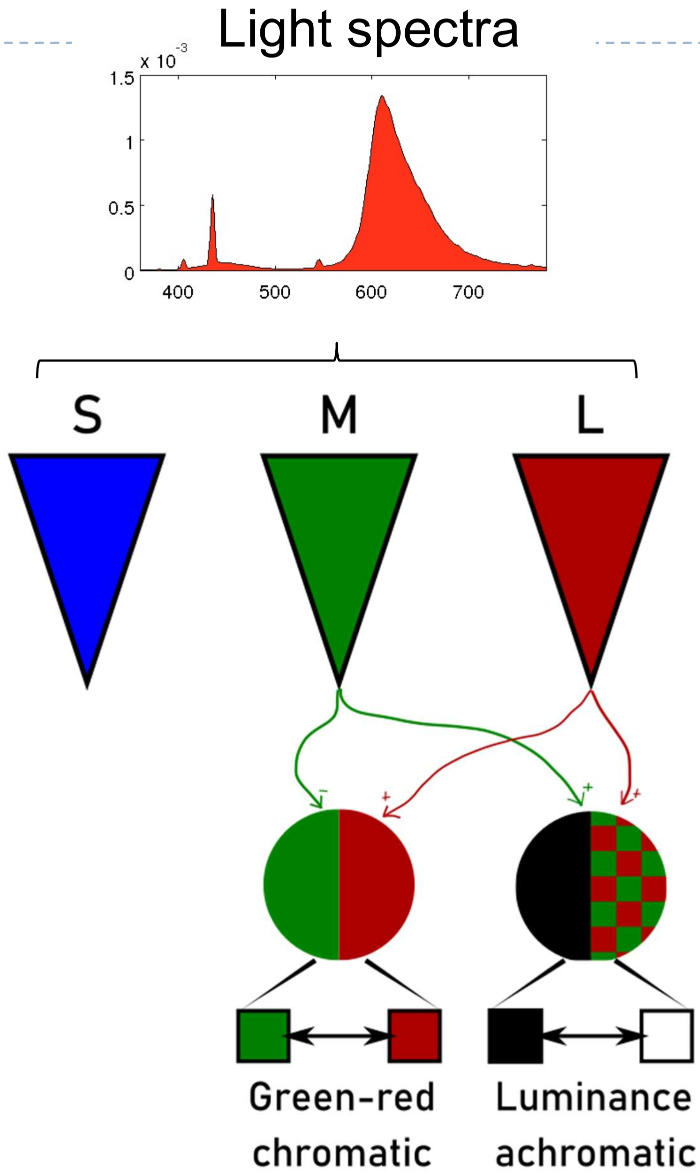




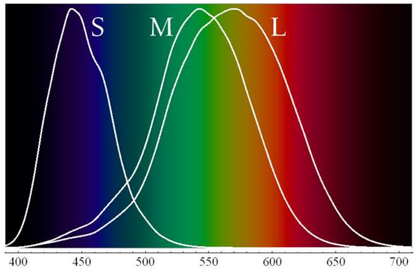
Sensitivity of the achromatic mechanism

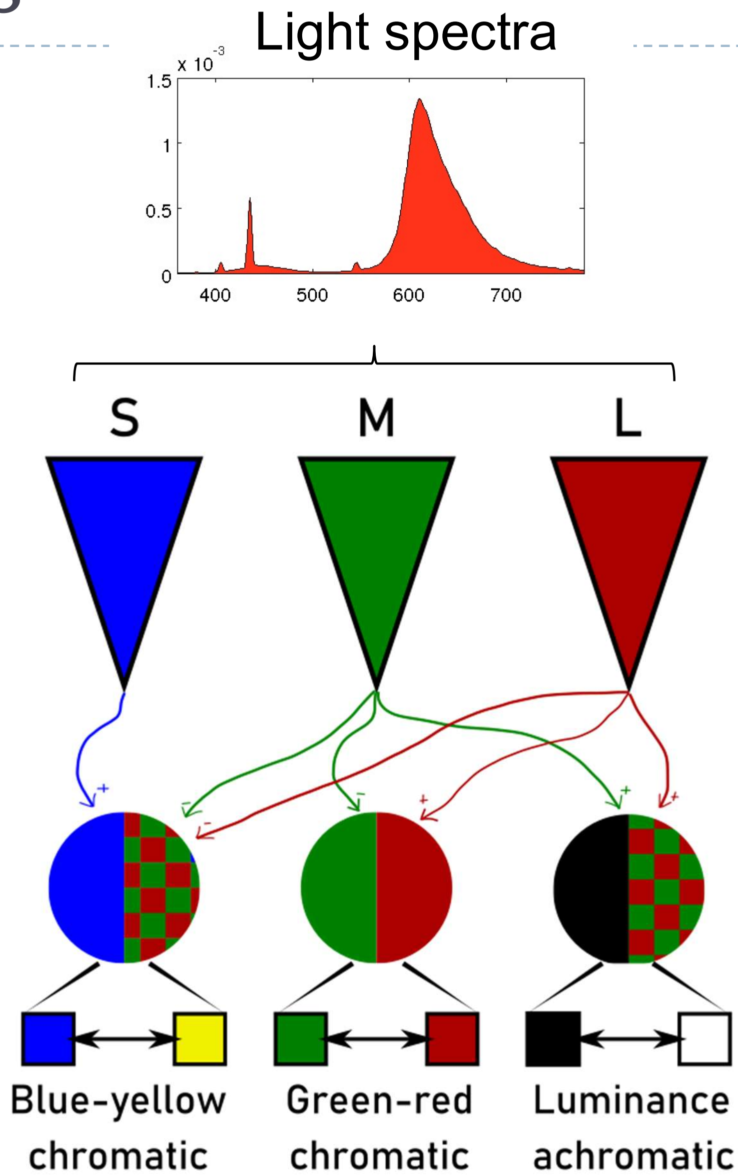
Achromatic/chromatic vision mechanisms



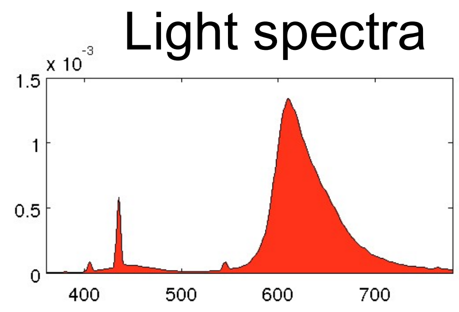


Achromatic/chromatic vision mechanisms

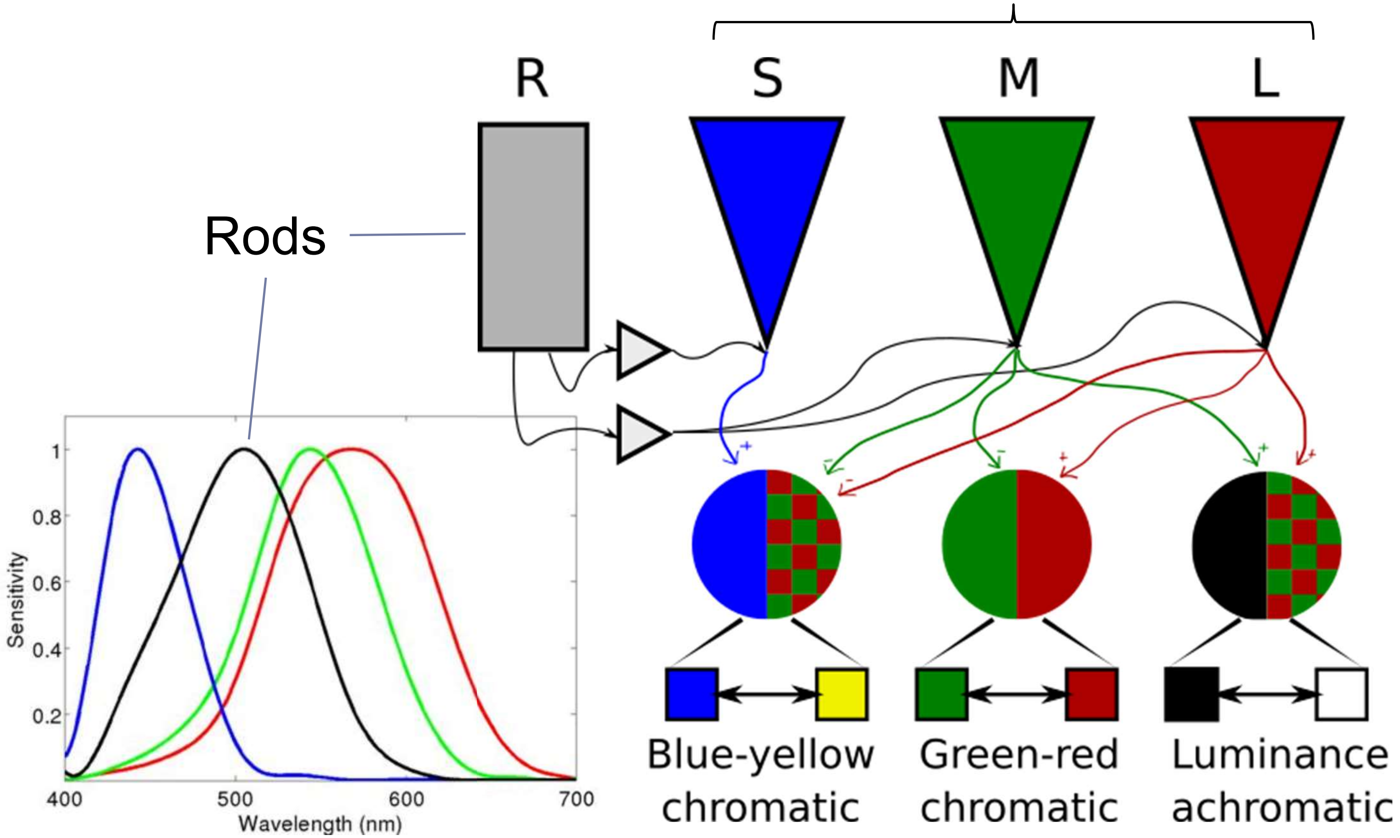




Achromatic/chromatic vision mechanisms



Cao et al. (2008). *Vision Research*, *48*(26), 2586–92.



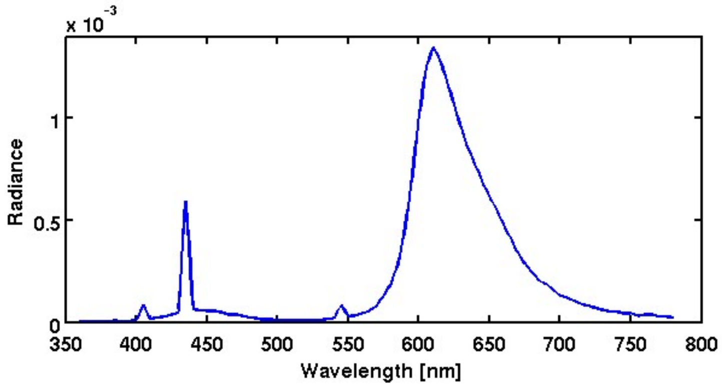
Luminance

- Luminance – measure of light weighted by the response of the achromatic mechanism. Units: cd/m^2

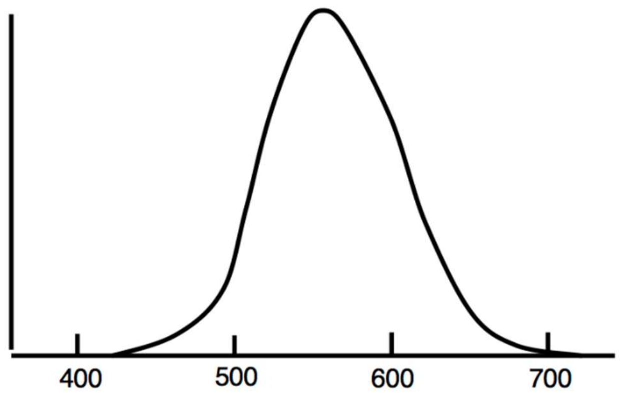
Luminance

$$L_V = \int_{350}^{700} kL(\lambda)V(\lambda)d\lambda \qquad k = \frac{1}{683.002}$$

Light spectrum (radiance)

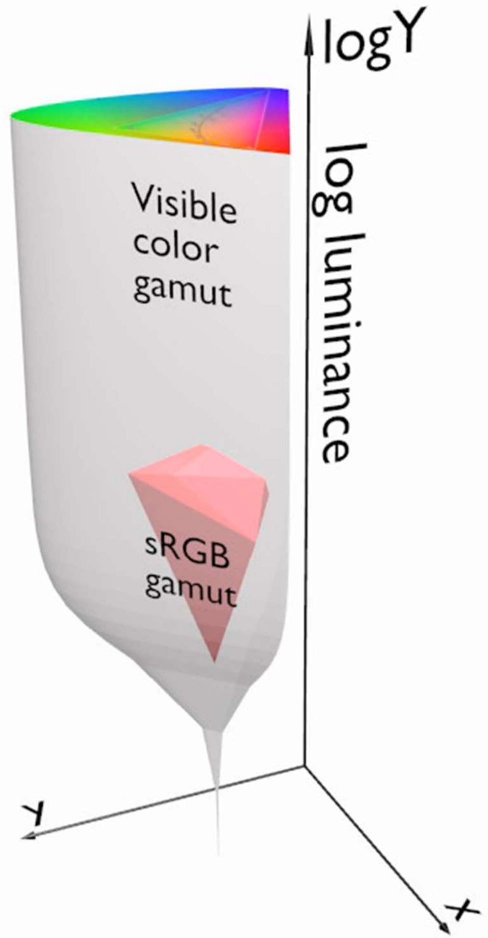


Luminous efficiency function (weighting)



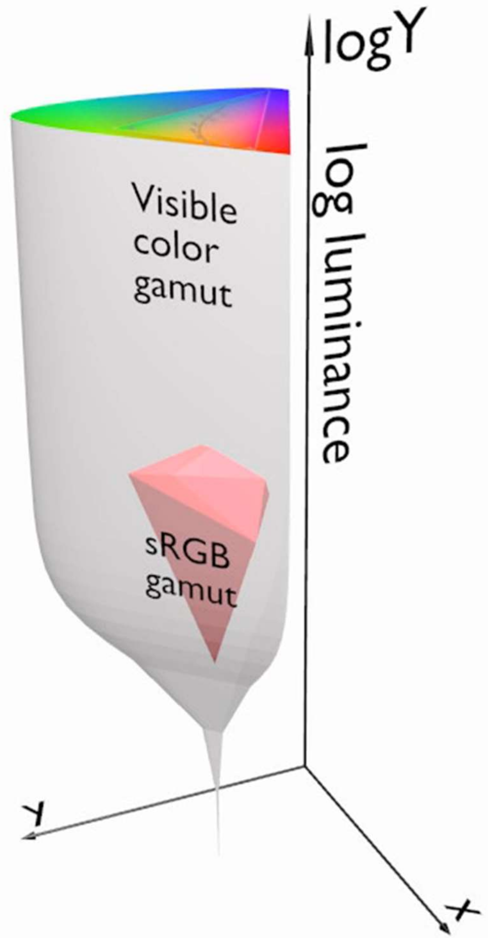
Visible vs. displayable colours

- All physically possible and visible colours form a solid in XYZ space
- Each display device can reproduce a subspace of that space
- A chromacity diagram is a slice taken from a 3D solid in XYZ space
- Colour Gamut – the solid in a colour space
  - Usually defined in XYZ to be device-independent

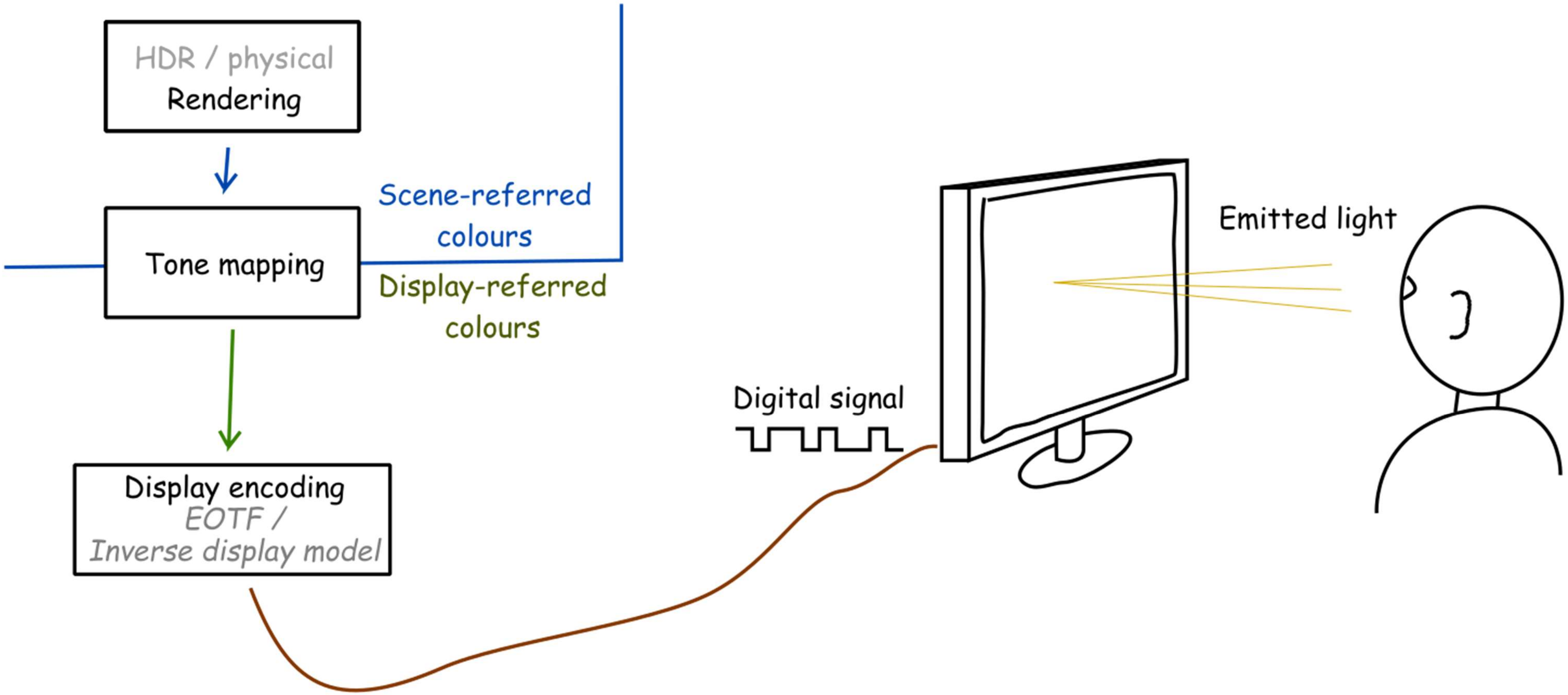


Standard vs. High Dynamic Range

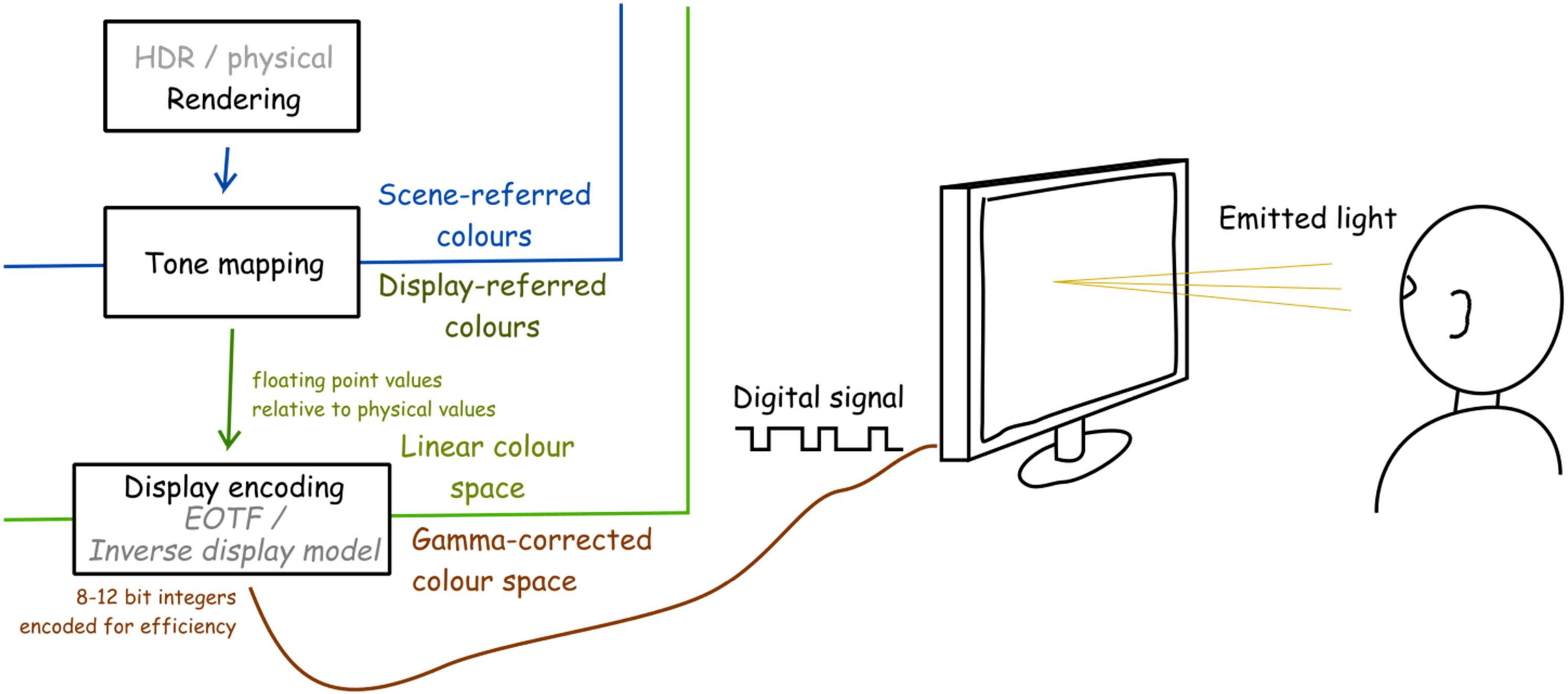
- **HDR** cameras/formats/displays attempt capture/represent/reproduce (almost) all visible colours
  - They represent scene colours and therefore we often call this representation *scene-referred*
- **SDR** cameras/formats/devices attempt to capture/represent/reproduce only colours of a standard sRGB colour gamut, mimicking the capabilities of CRTs monitors
  - They represent display colours and therefore we often call this representation *display-referred*



From rendering to display



From rendering to display



Display encoding for SDR: gamma correction

- Gamma correction is often used to encode luminance or tri-stimulus color values (RGB) in imaging systems (displays, printers, cameras, etc.)

Gain

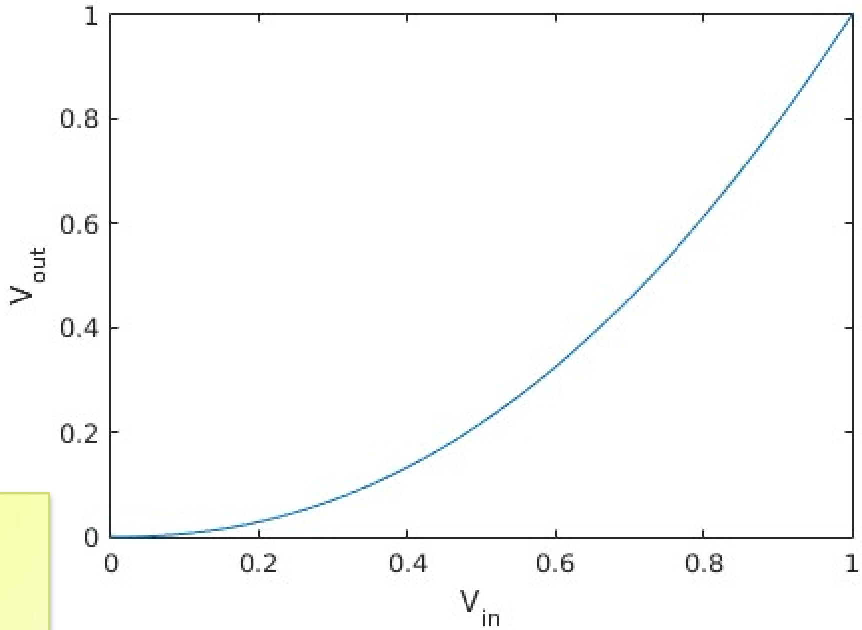
Gamma (usually =2.2)

$$\mathrm{V_{out}} = \mathrm{a} \cdot V_{in}^{\gamma}$$

(relative) Luminance
Physical signal

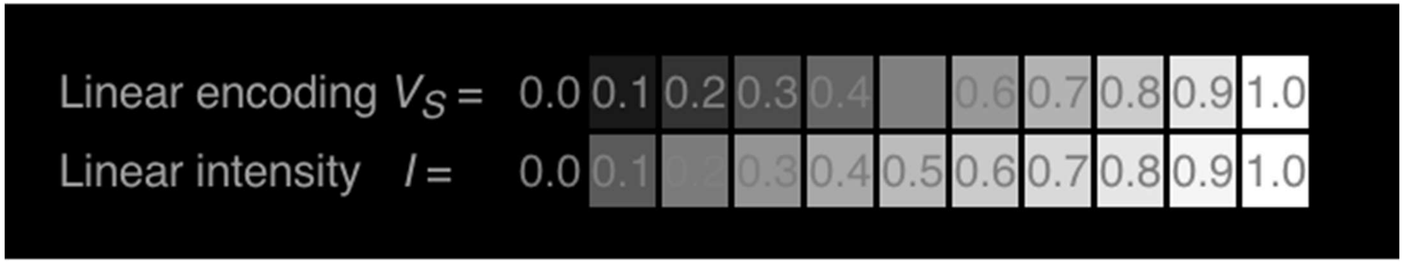
Luma
Digital signal (0-1)

Inverse: $\mathrm{V_{in}} = \left(\frac{1}{a} \cdot V_{out}\right)^{\frac{1}{\gamma}}$



Colour: the same equation applied to red, green and blue colour channels.

Why is gamma needed?



<- Pixel value (luma)

<- Luminance

- *Gamma-corrected* pixel values give a scale of brightness levels that is more perceptually uniform
- At least 12 bits (instead of 8) would be needed to encode each color channel without gamma correction
- And accidentally it was also the response of the CRT gun

Luma – gray-scale pixel value

- **Luma** - pixel brightness in *gamma corrected* units
$$L' = 0.2126R' + 0.7152G' + 0.0722B'$$
  - R', G' and B' are *gamma-corrected* colour values
  - Prime symbol denotes *gamma corrected*
  - Used in image/video coding

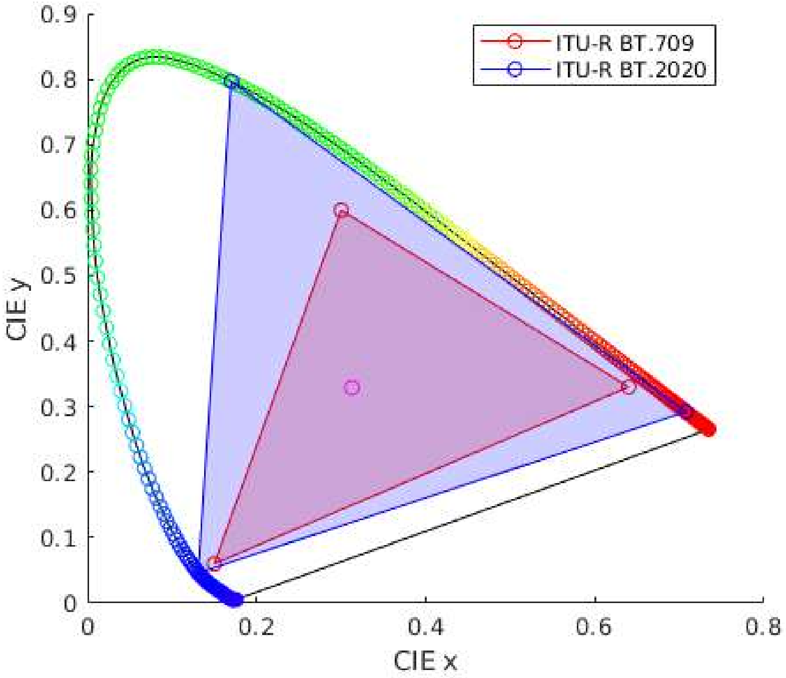
- Note that relative **luminance** if often approximated with
$$L = 0.2126R + 0.7152G + 0.0722B$$
$$= 0.2126(R')^{\gamma}+0.7152(G')^{\gamma}+0.0722(B')^{\gamma}$$
- R, G, and B are *linear* colour values
- Luma and luminace are different quantities despite similar formulas

Standards for display encoding

Display type	Colour space	EOTF	Bit depth
Standard Dynamic Range	ITU-R 709	2.2 gamma / sRGB	8 to 10
High Dynamic Range	ITU-R 2020	ITU-R 2100 (PQ/HLG)	10 to 12

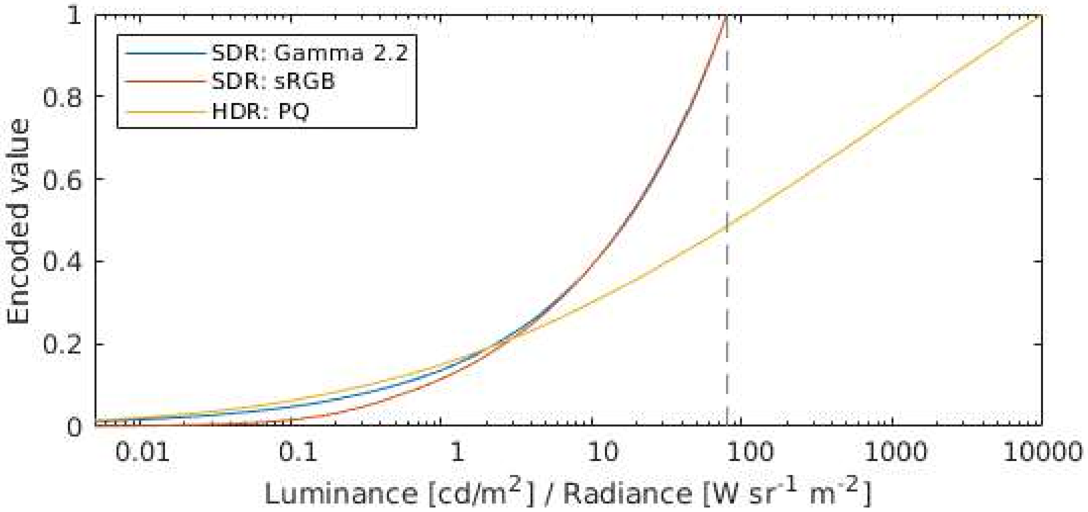
Colour space

What is the colour of "pure" red, green and blue



Electro-Optical Transfer Function

How to efficiently encode each primary colour



How to transform between RGB colour spaces?

display-encoded		linear				linear		display-encoded
R'G'B' sRGB	→ ←	RGB ITU-R 709	→ ←	XYZ	→ ←	RGB ITU-R 2020	→ ←	R'G'B' PQ-encoded
SDR				Device -independent		HDR		

- From ITU-R 709 RGB to XYZ:

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} 0.4124 & 0.3576 & 0.1805 \\ 0.2126 & 0.7152 & 0.0722 \\ 0.0193 & 0.1192 & 0.9505 \end{bmatrix}_{R709toXYZ} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix}_{R709}$$

Relative XYZ of the red primary

Relative XYZ of the green primary

Relative XYZ of the blue primary

Relative RGB (0-1) in the R709 space

How to transform between RGB colour spaces?

- From ITU-R **709** RGB to ITU-R **2020** RGB:

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix}_{R2020} = M_{XYZtoR2020} \cdot M_{R709toXYZ} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix}_{R709}$$

- From ITU-R **2020** RGB to ITU-R **709** RGB:

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix}_{R709} = M_{XYZtoR709} \cdot M_{R2020toXYZ} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix}_{R2020}$$

- Where:

$$M_{R709toXYZ} = \begin{bmatrix} 0.4124 & 0.3576 & 0.1805 \\ 0.2126 & 0.7152 & 0.0722 \\ 0.0193 & 0.1192 & 0.9505 \end{bmatrix} \text{ and } M_{XYZtoR709} = M_{R709toXYZ}^{-1}$$

$$M_{R2020toXYZ} = \begin{bmatrix} 0.6370 & 0.1446 & 0.1689 \\ 0.2627 & 0.6780 & 0.0593 \\ 0.0000 & 0.0281 & 1.0610 \end{bmatrix} \text{ and } M_{XYZtoR2020} = M_{R2020toXYZ}^{-1}$$

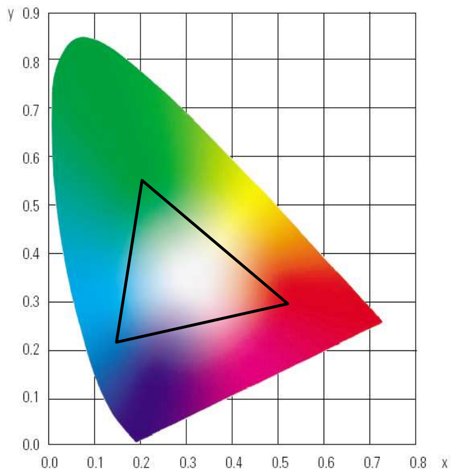
Representing colour

- We need a mechanism which allows us to represent colour in the computer by some set of numbers
  - A) preferably a small set of numbers which can be quantised to a fairly **small number of bits** each
    - Linear and gamma corrected RGB, sRGB
  - B) a set of numbers that are **easy to interpret**
    - Munsell's *artists'* scheme
    - HSV, HLS
  - C) a set of numbers in a 3D space so that the (Euclidean) distance in that space corresponds to approximately **perceptually uniform** colour differences
    - CIE Lab, CIE Luv

RGB spaces

- Most display devices that output light mix red, green and blue lights to make colour
  - televisions, CRT monitors, LCD screens

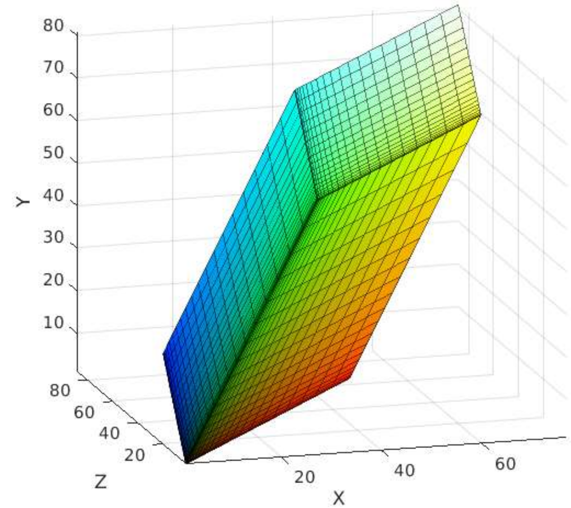
- RGB colour space
  - Can be **linear** (RGB) or **display-encoded** (R'G'B')
  - Can be **scene-referred** (HDR) or **display-referred** (SDR)
- There are multiple RGB colour spaces
  - ITU-R 709 (sRGB), ITU-R 2020, Adobe RGB, DCI-P3
    - Each using different primary colours
  - And different OETFs (gamma, PQ, etc.)
- Nominally, *RGB* space is a cube



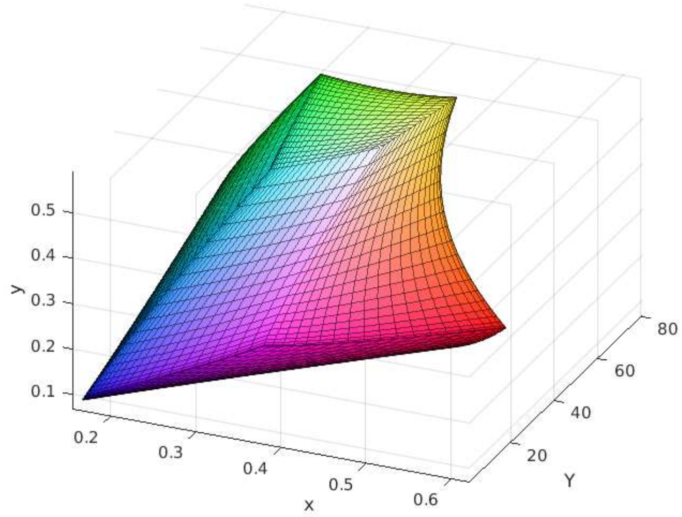
RGB in CIE *XYZ* space

- Linear RGB colour values can be transformed into CIE XYZ
  - by matrix multiplication
  - because it is a rigid transformation
    the colour gamut in CIE XYZ is a rotate and skewed cube
- Transformation into Yxy
  - is non-linear (non-rigid)
  - colour gamut is more complicated

RGB gamut in XYZ colour space

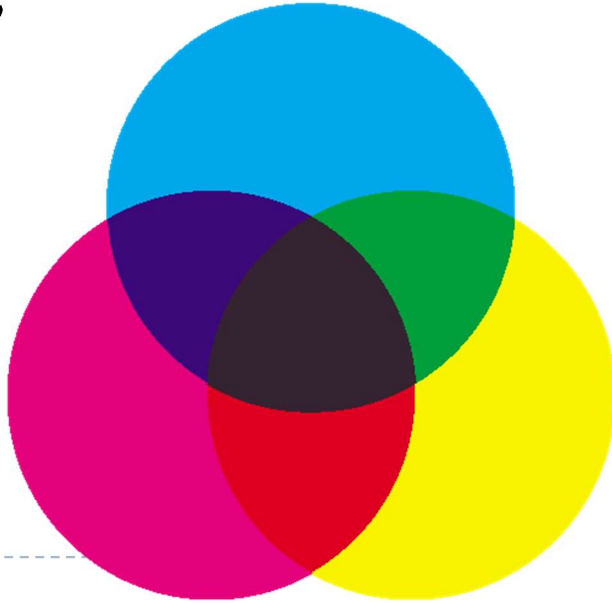


RGB gamut in Yxy colour space

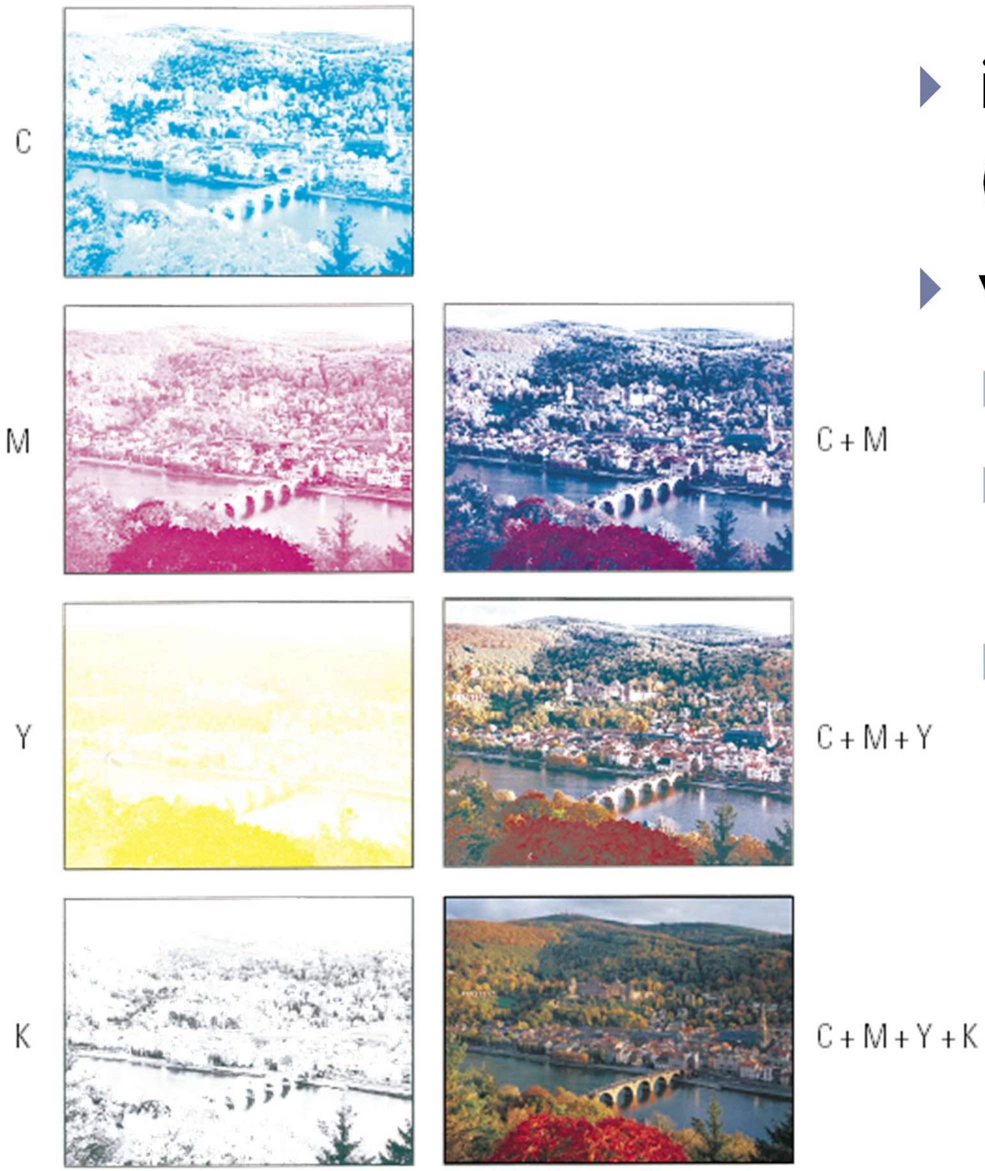


CMY space

- 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



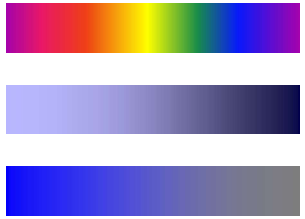
CMYK space

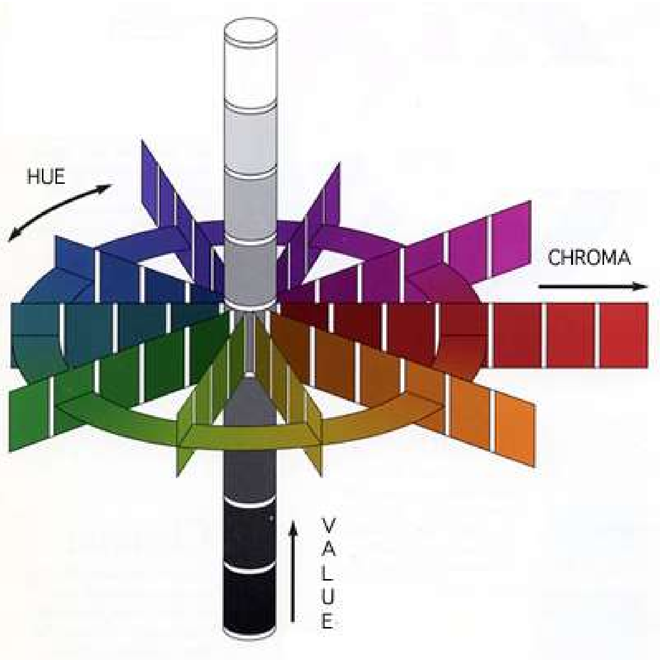


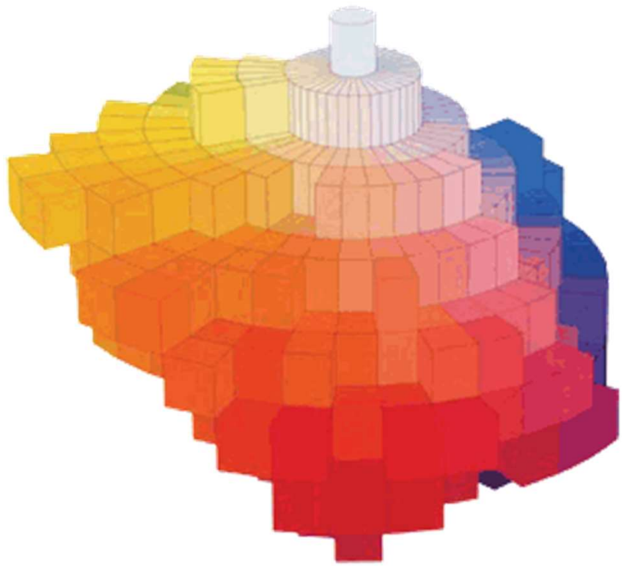
- in real printing we use black (key) as well as *CMY*
- why use black?
  - inks are not perfect absorbers
  - mixing $C + M + Y$ gives a muddy grey, not black
  - lots of text is printed in black: trying to align C, M and Y perfectly for black text would be a nightmare

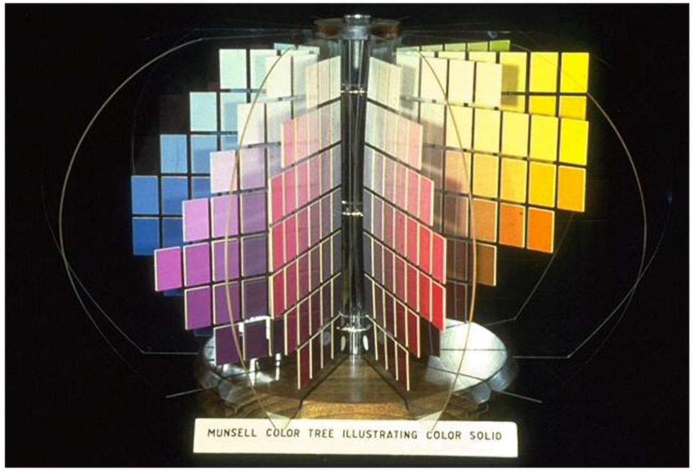
Munsell's colour classification system

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



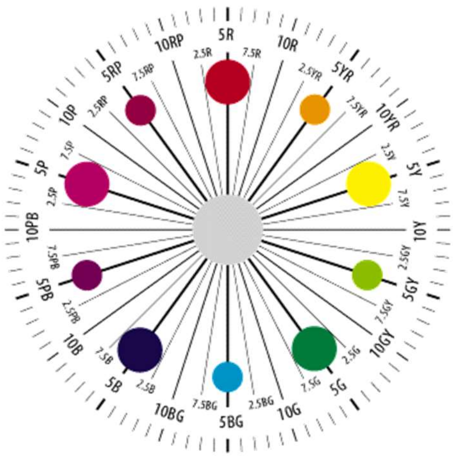


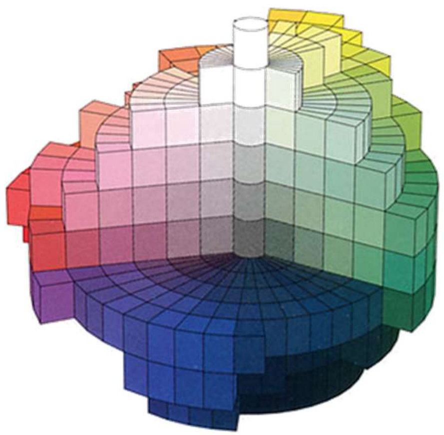


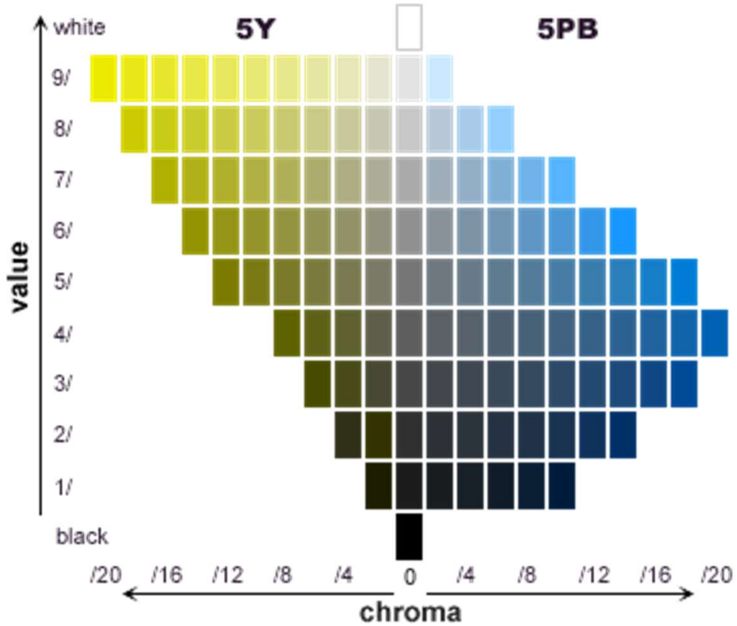


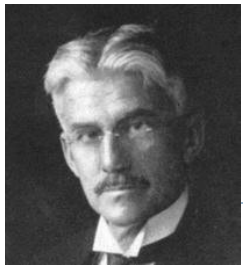
Munsell's colour classification system

- 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









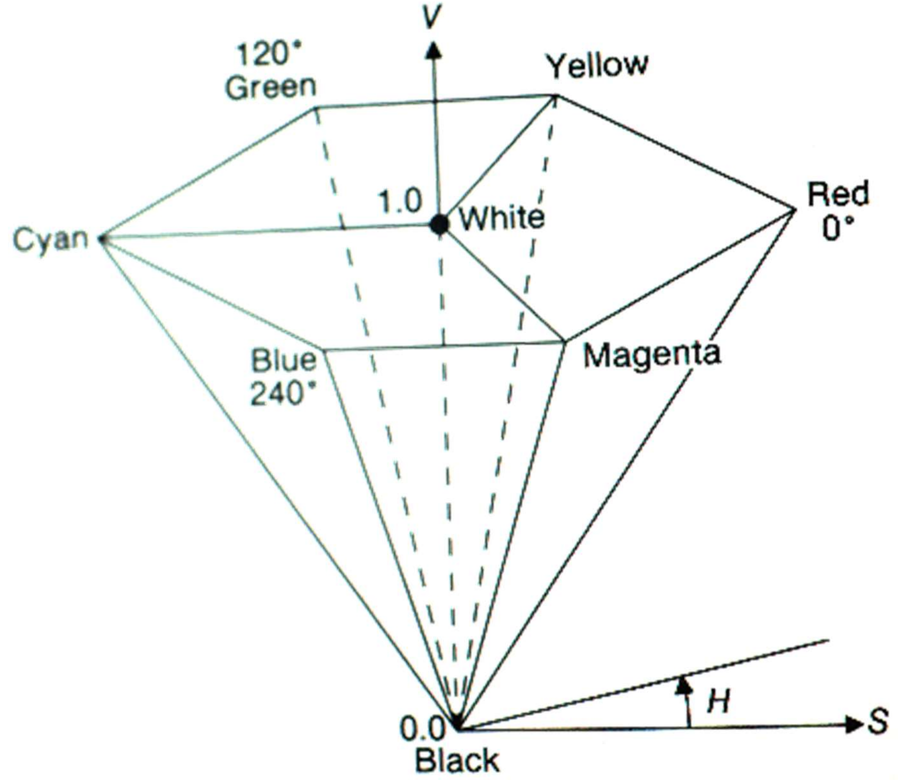
invented by Albert H. Munsell, an American artist, in 1905 in an attempt to systematically classify colours

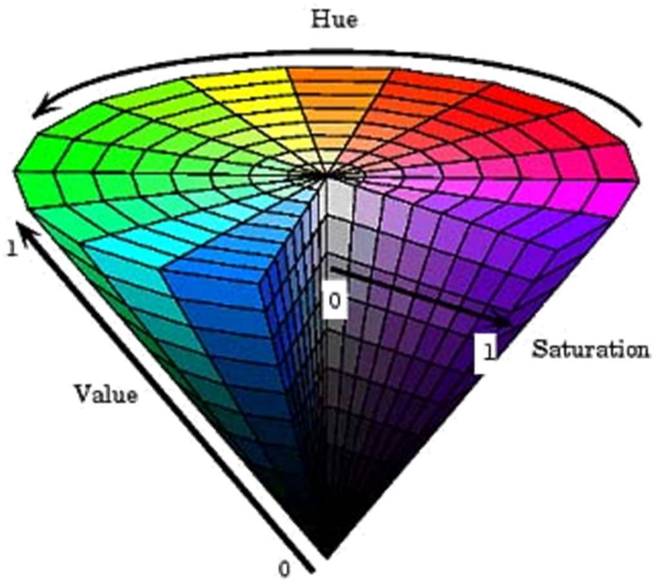
Colour spaces for user-interfaces

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

HSV: hue saturation value

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

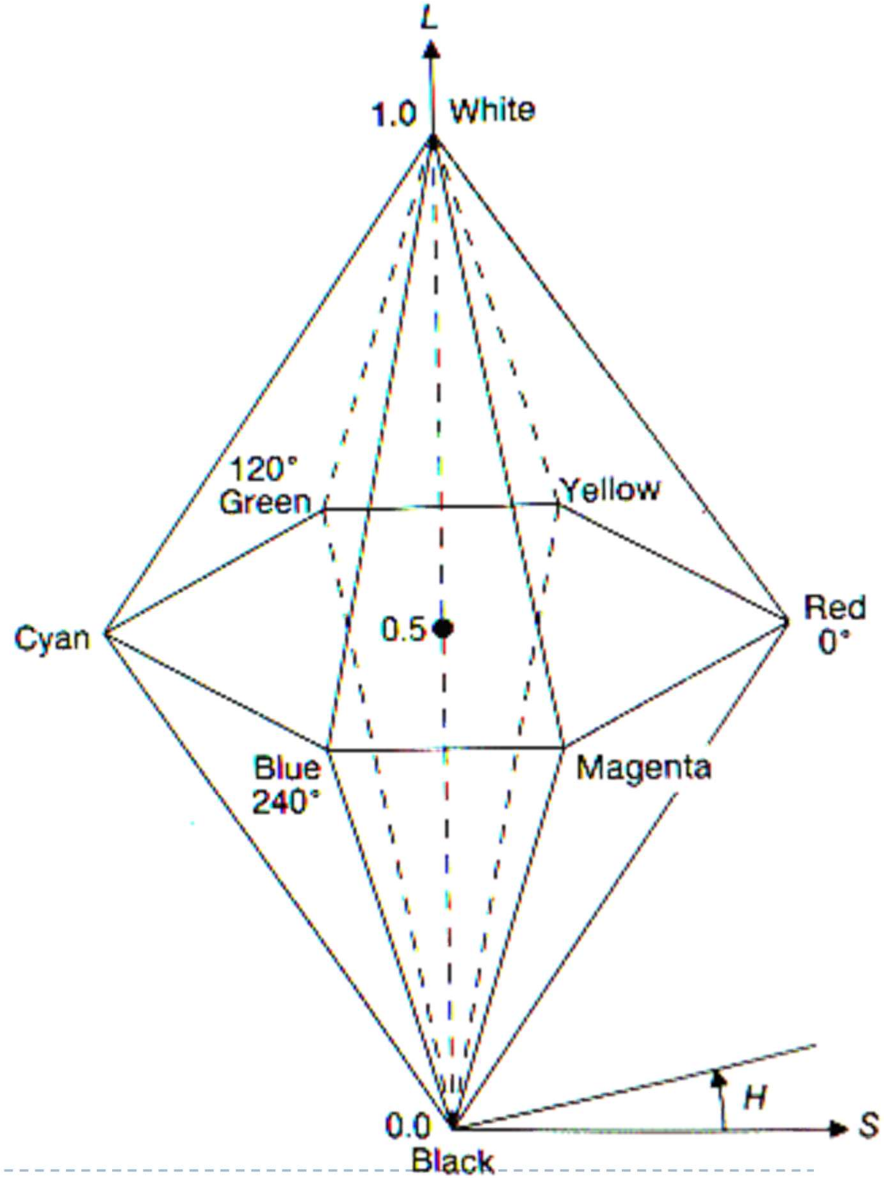




- designed by Alvy Ray Smith in 1978
- algorithm to convert *HSV* to *RGB* and back can be found in Foley et al., Figs 13.33 and 13.34

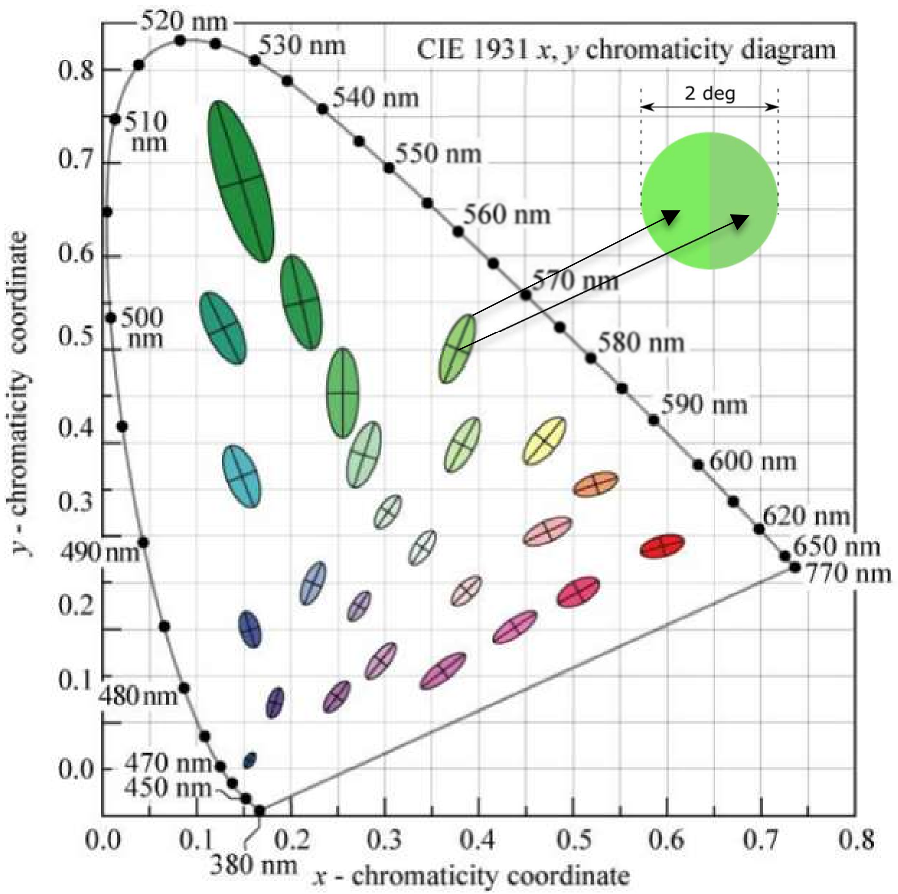
HLS: hue lightness saturation

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

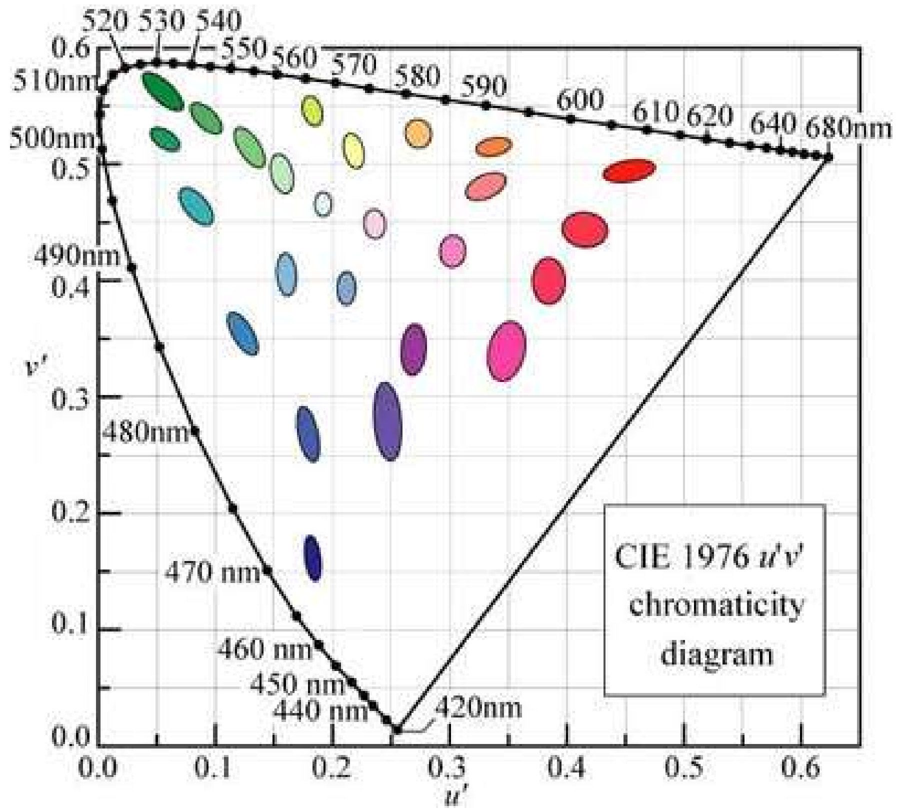


Perceptually uniformity

- MacAdam ellipses & visually indistinguishable colours



In CIE xy chromatic coordinates



In CIE u'v' chromatic coordinates

CIE L*u*v* and u'v'

- Approximately perceptually uniform
- u'v' chromacity

$$u' = \frac{4X}{X+15Y+3Z} = \frac{4x}{-2x+12y+3}$$
$$v' = \frac{9Y}{X+15Y+3Z} = \frac{9y}{-2x+12y+3}$$

- CIE LUV

Lightness

$$L^* = \begin{cases} \left(\frac{29}{3}\right)^3 Y/Y_n, & Y/Y_n \le \left(\frac{6}{29}\right)^3 \\ 116(Y/Y_n)^{1/3} - 16, & Y/Y_n > \left(\frac{6}{29}\right)^3 \end{cases}$$

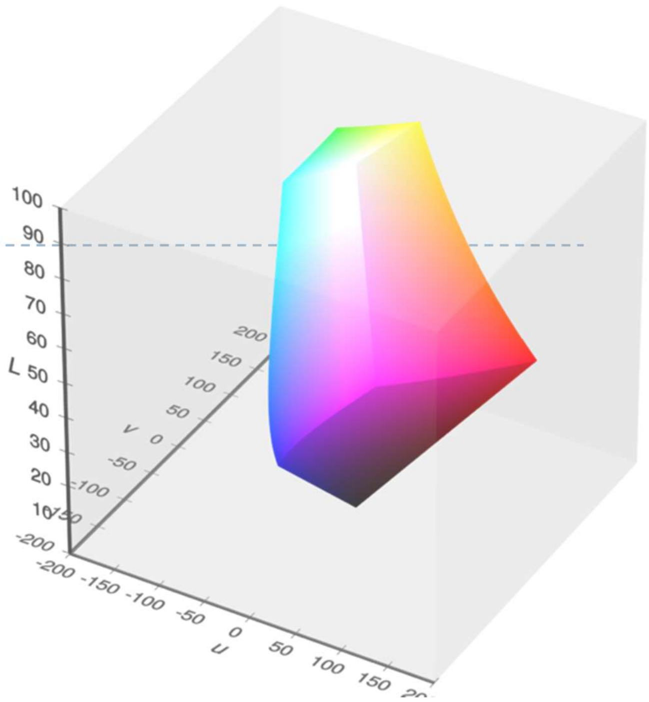
Chromacity coordinates

$$u^* = 13L^* \cdot (u' - u'_n)$$
$$v^* = 13L^* \cdot (v' - v'_n)$$

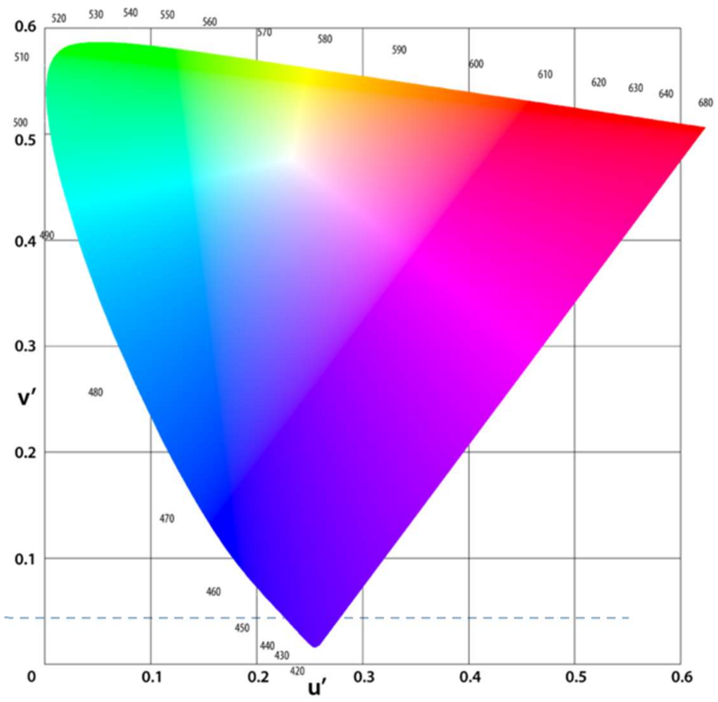
Colours less distinguishable when dark

- Hue and chroma

$$C^*_{uv} = \sqrt{(u^*)^2 + (v^*)^2}$$
$$h_{uv} = \mathrm{atan2}(v^*, u^*),$$



sRGB in CIE L*u*v*



CIE L*a*b* colour space

- Another approximately perceptually uniform colour space

$$L^\star = 116 f\left(\frac{Y}{Y_n}\right) - 16$$

$$a^\star = 500\left(f\left(\frac{X}{X_n}\right) - f\left(\frac{Y}{Y_n}\right)\right)$$

$$b^\star = 200\left(f\left(\frac{Y}{Y_n}\right) - f\left(\frac{Z}{Z_n}\right)\right)$$

Trichromatic values of the white point, e.g.

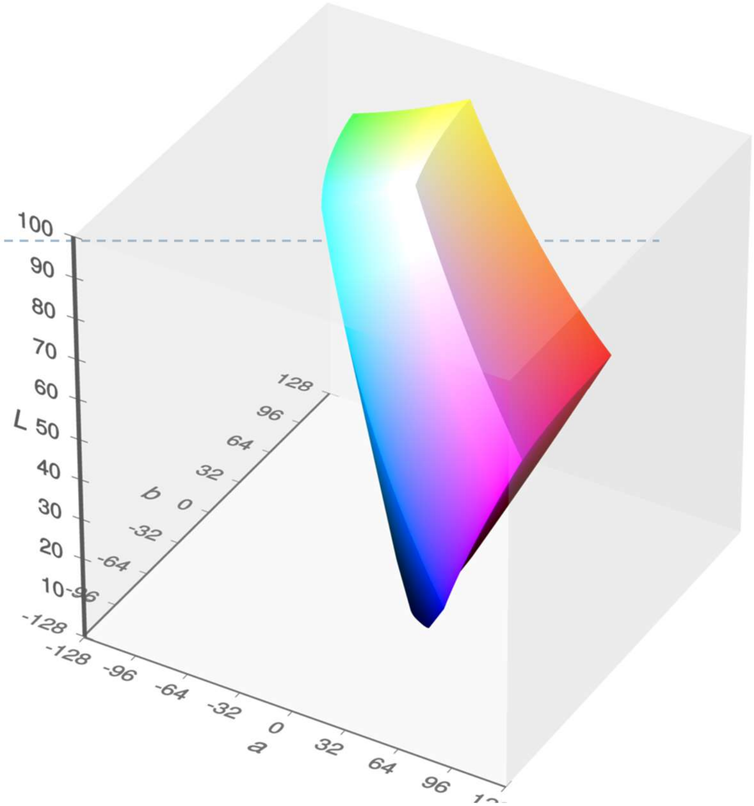
$$X_n = 95.047,\quad Y_n = 100.000,\quad Z_n = 108.883$$

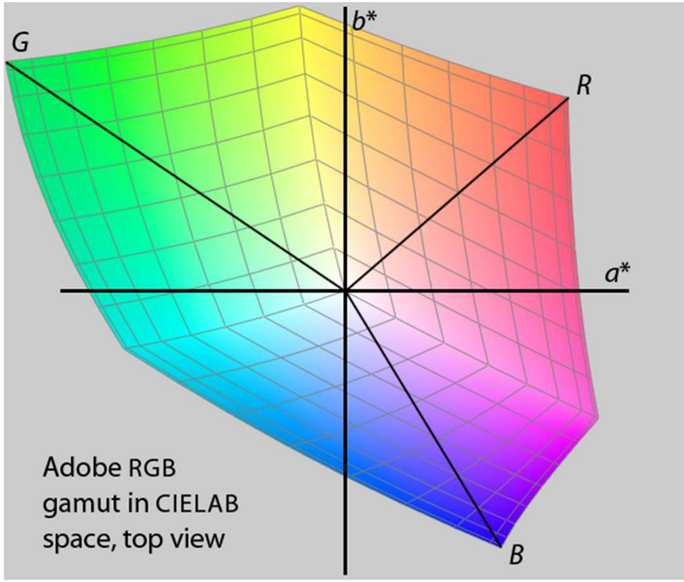
$$f(t) = \begin{cases} \sqrt[3]{t} & \text{if } t > \delta^3 \\ \frac{t}{3\delta^2} + \frac{4}{29} & \text{otherwise} \end{cases}$$

$$\delta = \frac{6}{29}$$

- Chroma and hue

$$C^\star = \sqrt{a^{\star 2} + b^{\star 2}}, \qquad h^\circ = \arctan\left(\frac{b^\star}{a^\star}\right)$$





Lab space

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

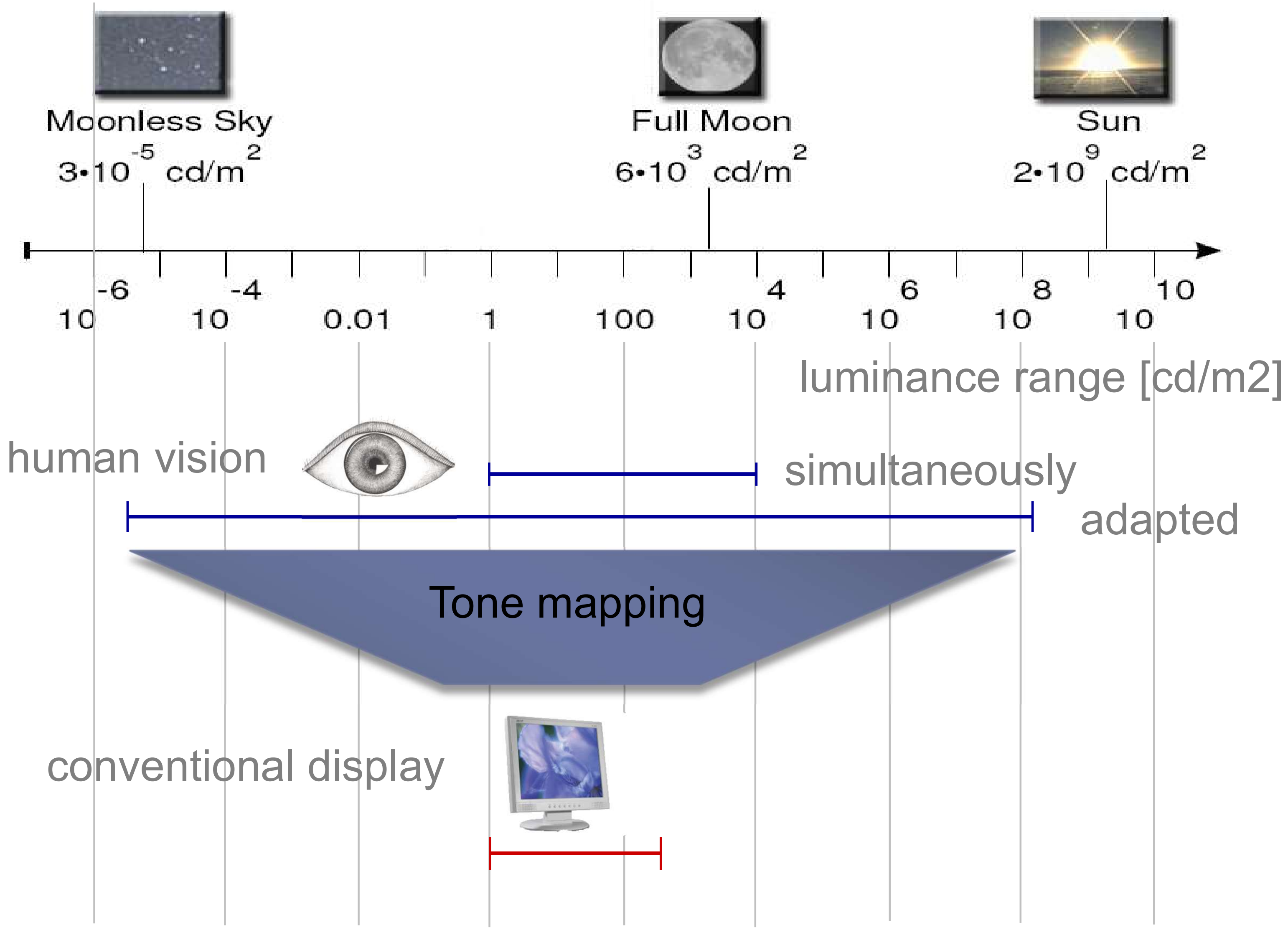
Recap: Linear and display-encoded colour

- Linear colour spaces
  - Examples: CIE XYZ, LMS cone responses, linear RGB
  - Typically floating point numbers
  - Directly related to the measurements of light (radiance and luminance)
  - Perceptually non-uniform
  - Transformation between linear colour spaces can be expressed as a matrix multiplication
- Display-encoded and non-linear colour spaces
  - Examples: display-encoded (gamma-corrected, gamma-encoded) RGB, HVS, HLS, PQ-encoded RGB
  - Typically integers, 8-12 bits per colour channel
  - Intended for efficient encoding, easier interpretation of colour, perceptual uniformity

Colour - references

- Chapters „Light” and „Colour” in
  - Shirley, P. & Marschner, S., *Fundamentals of Computer Graphics*
- Textbook on colour appearance
  - Fairchild, M. D. (2005). *Color Appearance Models* (second.). John Wiley & Sons.

Tone-mapping problem



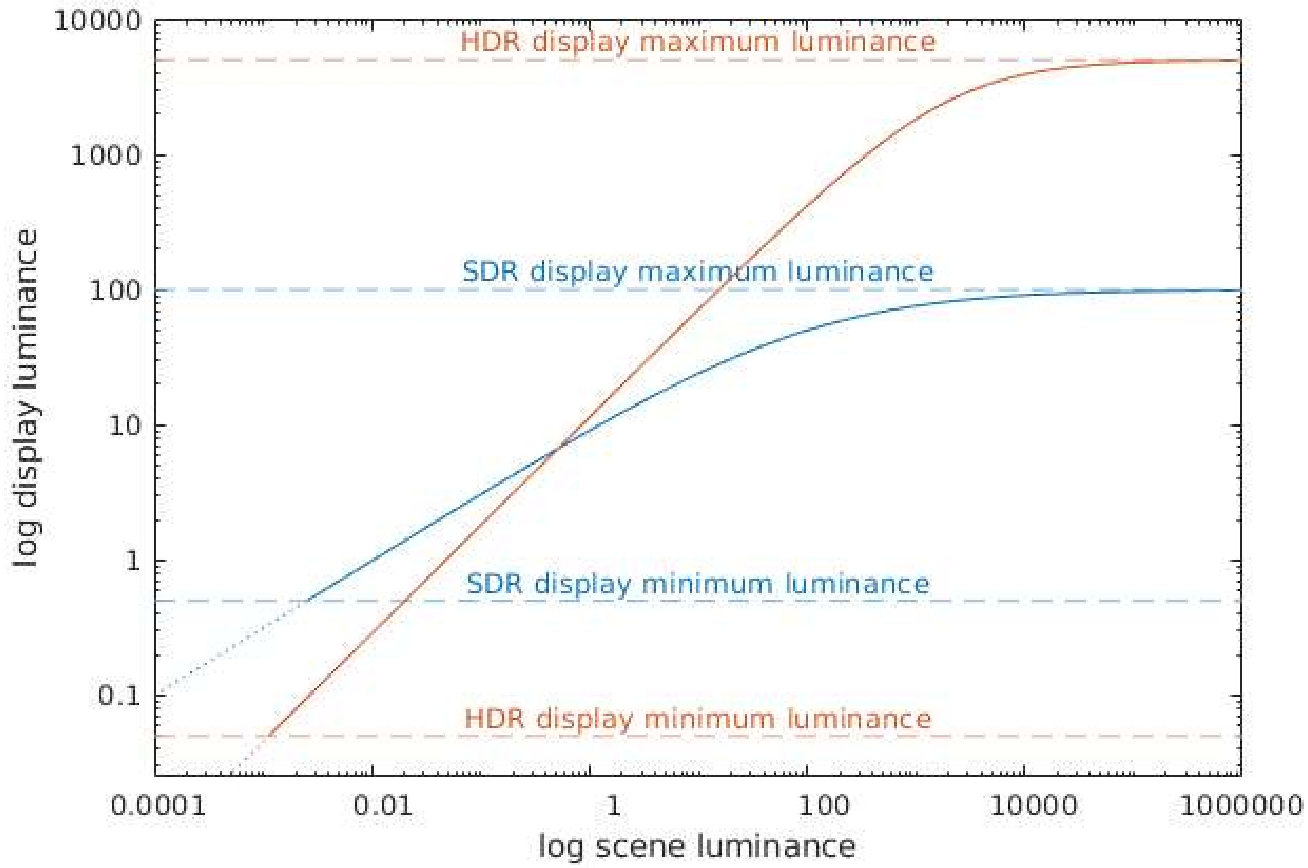
Why do we need tone mapping?

- To **reduce dynamic range**
- To **customize the look** (colour grading)
- To **simulate human vision** (for example night vision)
- To **simulate a camera** (for example motion blur)
- To **adapt** displayed images to a display and **viewing conditions**
- To make rendered images look **more realistic**
- To map from **scene- to display-referred** colours

- Different tone mapping operators achieve different combination of these goals

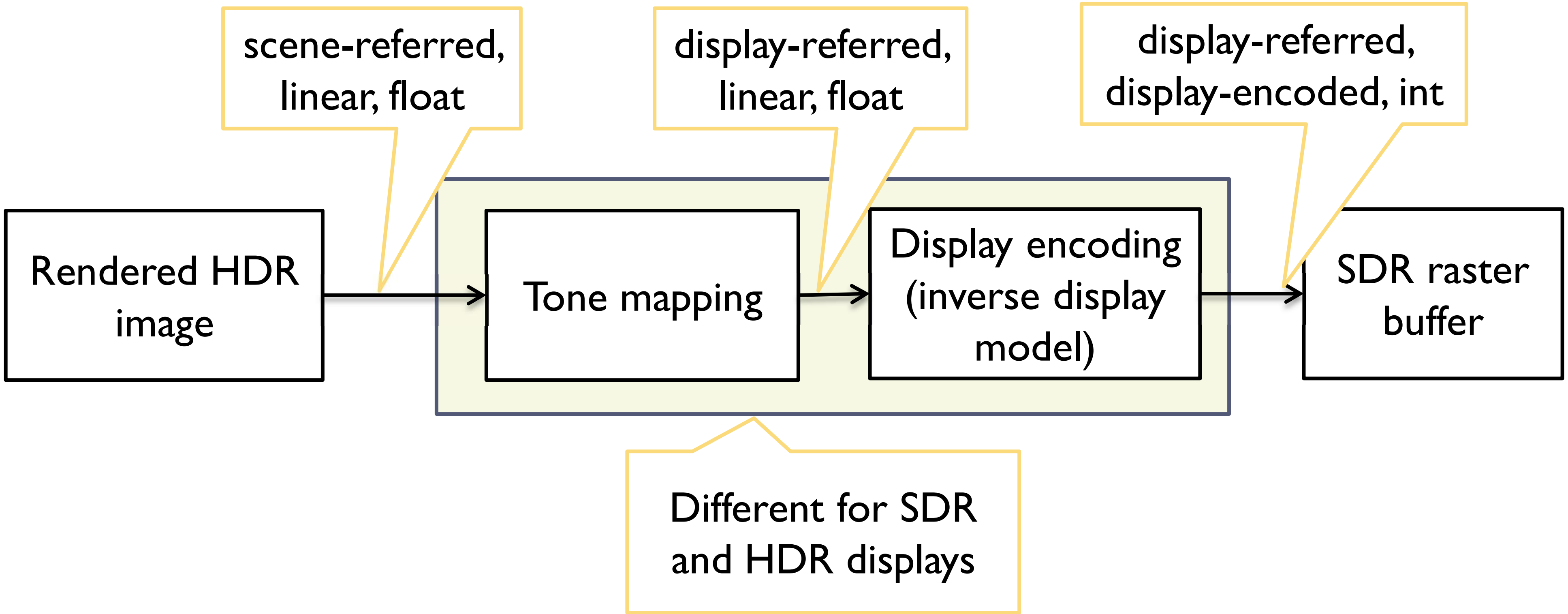
From scene- to display-referred colours

- The primary purpose of tone mapping is to transform an image from *scene-referred* to *display-referred* colours



Tone mapping and display encoding

- Tone mapping is often combined with display encoding



- Display encoding can model the display and account for
  - Display contrast (dynamic range), brightness and ambient light levels

Basic tone-mapping and display coding

- The simplest form of tone-mapping is the exposure/brightness adjustment:

$$R_d = \frac{R_s}{L_{white}}$$

Display-referred relative red value [0;1]

Scene-referred

Scene-referred luminance of white

  - R for red, the same for green and blue
  - No contrast compression, only for a moderate dynamic range
- The simplest form of display coding is the "gamma"

$$R' = (R_d)^{\frac{1}{\gamma}}$$

Prime (') denotes a gamma-corrected value

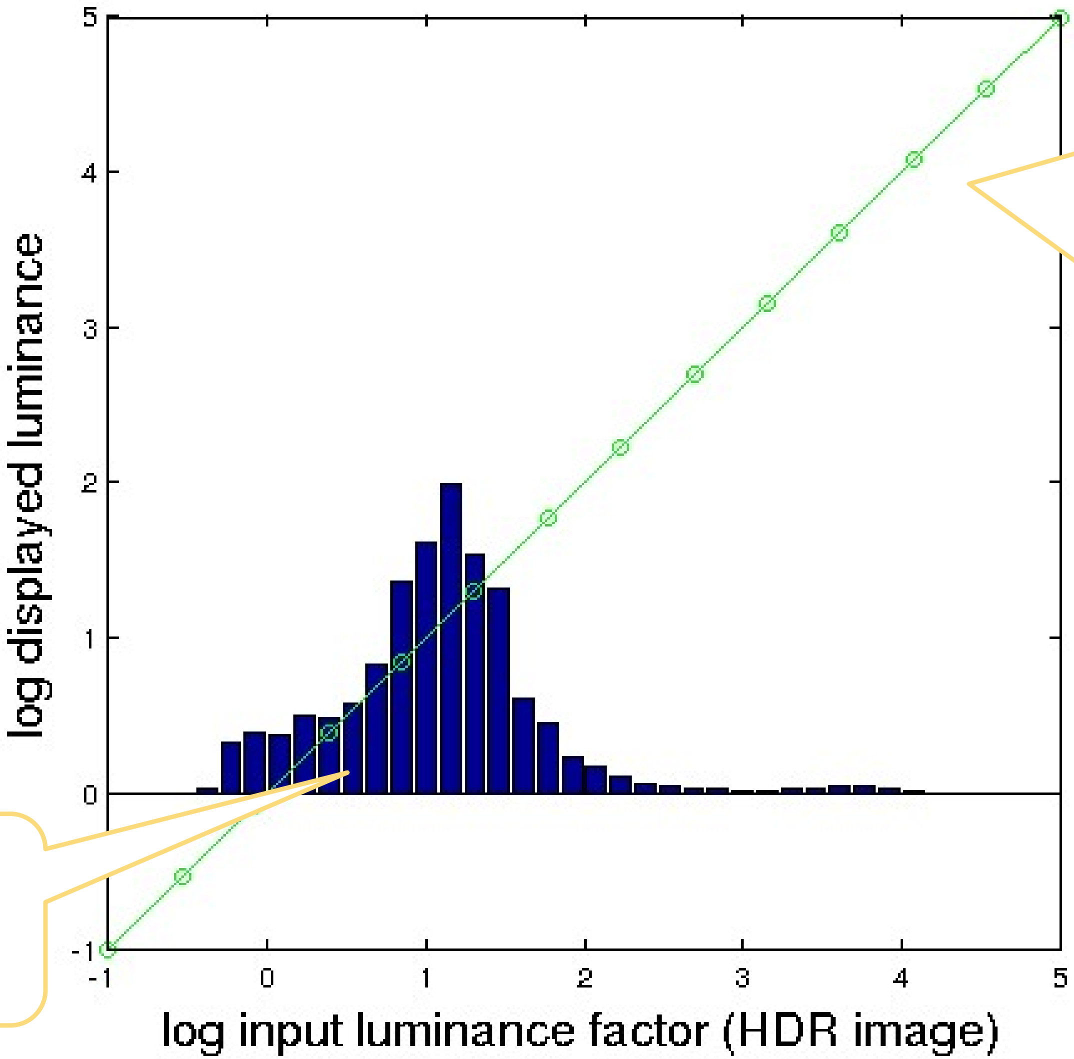
Typically γ=2.2

  - For SDR displays only

sRGB textures and display coding

- OpenGL offers sRGB textures to automate RGB to/from sRGB conversion
  - sRGB textures store data in gamma-corrected space
  - sRGB colour values are converted to (linear) RGB colour values on texture look-up (and filtering)
    - Inverse display coding
  - RGB to sRGB conversion when writing to sRGB texture
    - with glEnable(GL_FRAMEBUFFER_SRGB)
    - Forward display coding

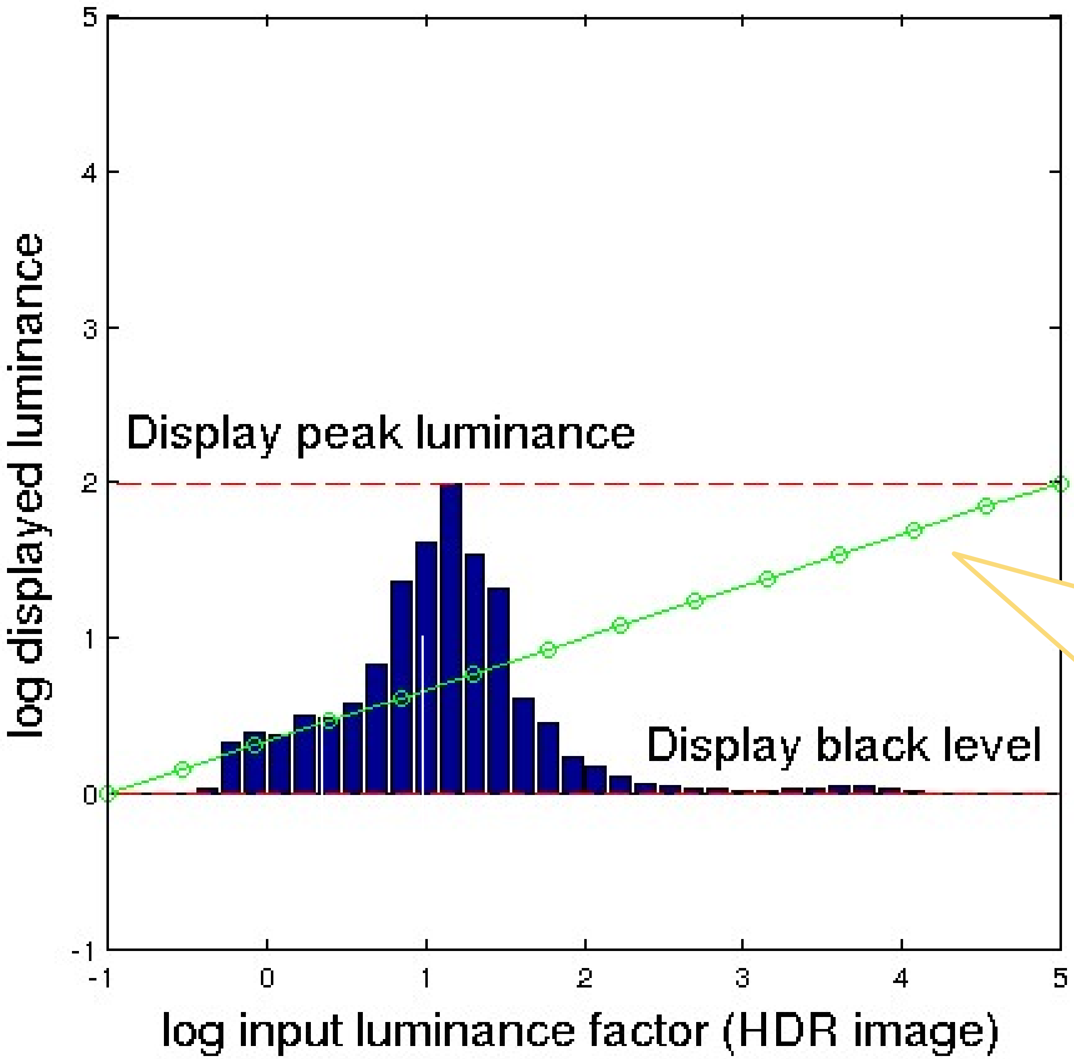
Tone-curve



The „best" tone-mapping is the one which does not do anything, i.e. slope of the tone-mapping curves is equal to 1.

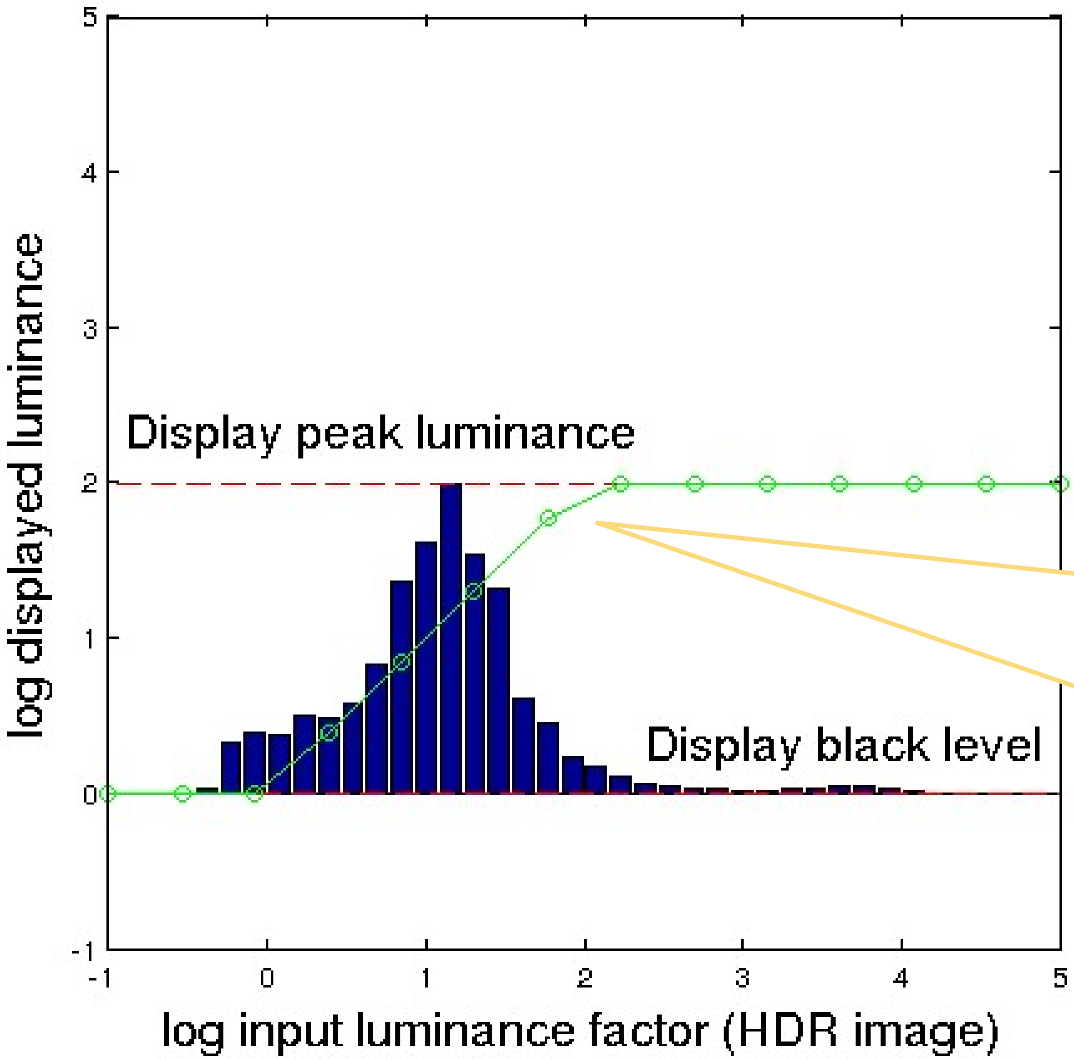
Image histogram

Tone-curve



But in practice contrast (slope) must be limited due to display limitations.

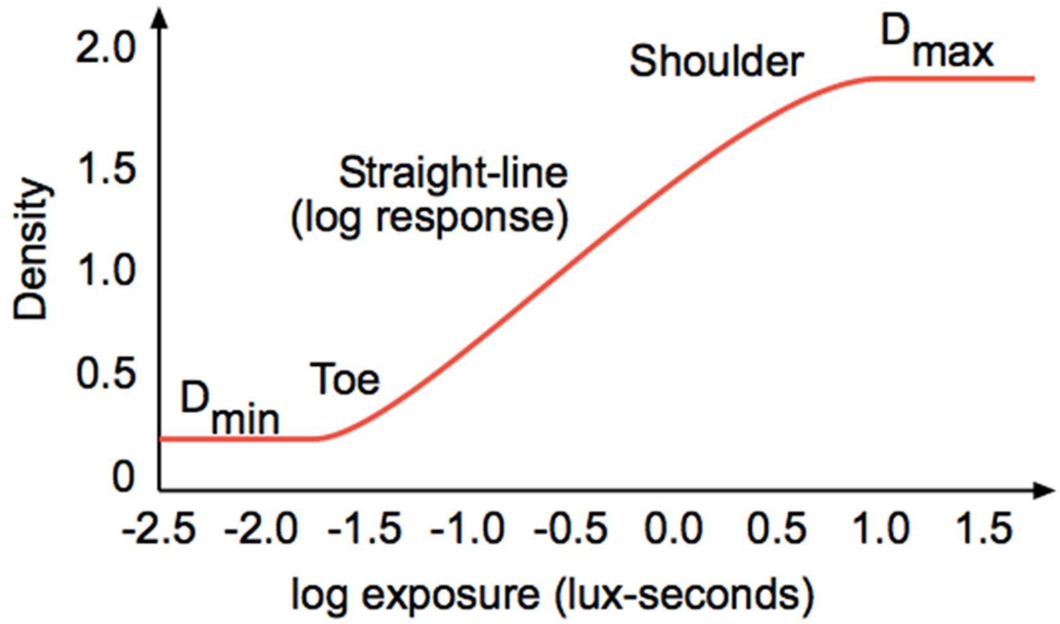
Tone-curve



Global tone-mapping is a compromise between clipping and contrast compression.

Sigmoidal tone-curves

- Very common in digital cameras
  - Mimic the response of analog film
  - Analog film has been engineered over many years to produce good tone-reproductio[n]
- Fast to compute



Sigmoidal tone mapping

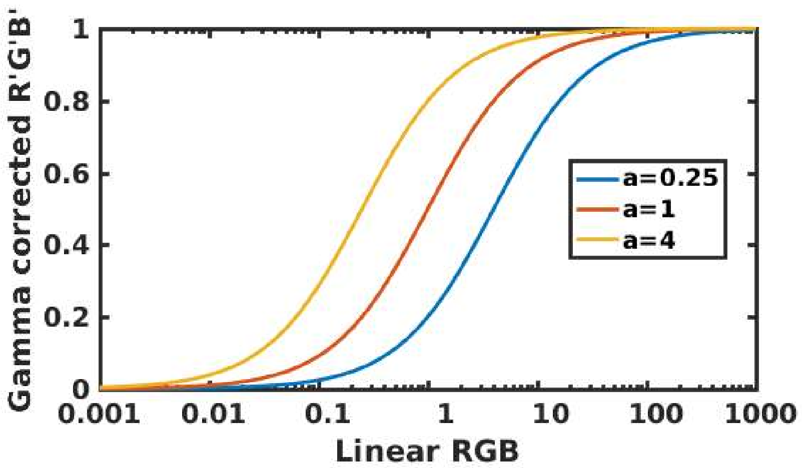
- Simple formula for a sigmoidal tone-curve:

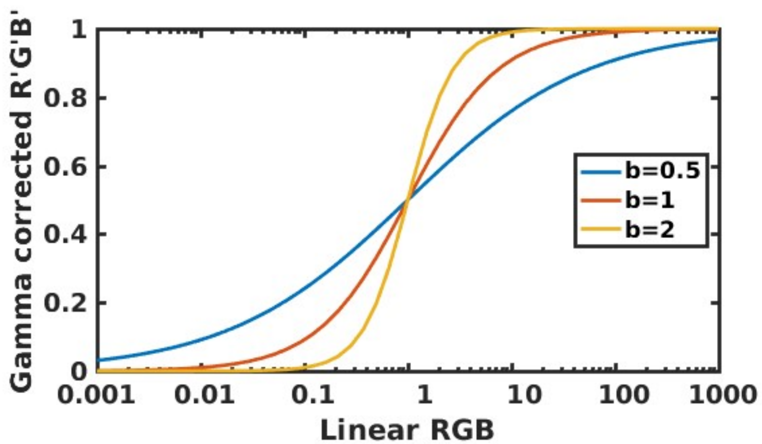
$$R'(x,y) = \frac{R(x,y)^b}{\left(\frac{L_m}{a}\right)^b + R(x,y)^b}$$

where L_m is the geometric mean (or mean of logarithms):

$$L_m = exp\left(\frac{1}{N}\sum_{(x,y)} \ln(L(x,y))\right)$$

and $L(x,y)$ is the luminance of the pixel (x,y).





Sigmoidal tone mapping example

a=0.25

a=1

a=4

b=0.5 b=1 b=2