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

Material is copyright © Neil A Dodgson, Peter Robinson & Rafał Mantiuk, 1996-2017, except where otherwise noted.

All other copyright material is made available under the University’s licence. All rights reserved.
Introduction to Computer Graphics
Peter Robinson & Rafał Mantiuk
www.cl.cam.ac.uk/~pr & ~rkm38

Eight lectures & two practical tasks for Part IA CST
Two supervisions suggested
Two exam questions on Paper 3
What are Computer Graphics & Image Processing?

- Scene description
- Digital image
- Image capture
- Image processing
- Image display
- Image analysis & computer vision
- Computer graphics
Why bother with CG & IP?

✿ All visual computer output depends on CG
  ◆ printed output (laser/ink jet/phototypesetter)
  ◆ monitor (CRT/LCD/plasma/DMD)
  ◆ all visual computer output consists of real images generated by the computer from some internal digital image

✿ Much other visual imagery depends on CG & IP
  ◆ TV & movie special effects & post-production
  ◆ most books, magazines, catalogues, brochures, junk mail, newspapers, packaging, posters, flyers
Course Structure

✧ **Background**
  - What is an image? Human vision. Resolution and quantisation. Storage of images in memory. [1 lecture]

✧ **Rendering**

✧ **Graphics pipeline**
  - Polygonal mesh models. Transformations using matrices in 2D and 3D. Homogeneous coordinates. Projection: orthographic and perspective. [1 lecture]

✧ **Graphics hardware and modern OpenGL**
  - Vertex processing. Rasterisation. Fragment processing. Working with meshes and textures. [2 lectures]

✧ **Technology**
  - Colour spaces. Output devices: brief overview of display and printer technologies. [1 lecture]
Course books

- **Fundamentals of Computer Graphics**
  - Shirley & Marschner

- **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)
Computer Graphics & Image Processing

Background
- What is an image?
- Human vision
- Resolution and quantisation
- Storage of images in memory

Rendering
- Graphics pipeline
- Graphics hardware and modern OpenGL
- Colour
What is required for vision?

- **illumination**
  - some source of light

- **objects**
  - which reflect (or transmit) the light

- **eyes**
  - to capture the light as an image

- **direct viewing**
- **transmission**
- **reflection**
The spectrum

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

the short wavelength end of the spectrum is violet
the long wavelength end of the spectrum is red

violet, blue, green, yellow, red

The spectrum of electromagnetic radiation includes a range of wavelengths, from ultraviolet (UV) to infrared (IR), with visible light occupying a small portion in the middle.

- **Violet** (shortest wavelength, highest energy)
- **Blue**
- **Green**
- **Yellow**
- **Red** (longest wavelength, lowest energy)

Visible light is a small subset of the entire electromagnetic spectrum, which also includes X-rays, gamma rays, microwaves, radio waves, and more.
What is an image?

- two dimensional function
- value at any point is an intensity or colour
- not digital!
The workings of the human visual system

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

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

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

- **two classes**
  - rods
  - cones

- **cones come in three types**
  - sensitive to short, medium and long wavelengths
  - allow you to see in colour

- **the cones are concentrated in the macula, at the centre of the retina**

- **the fovea is a densely packed region in the centre of the macula**
  - contains the highest density of cones
  - provides the highest resolution vision
Colour signals sent to the brain

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

\[ \text{long} + \text{medium} + \text{short} = \text{luminance} \]

\[ \text{long} - \text{medium} = \text{red-green} \]

\[ \text{long} + \text{medium} - \text{short} = \text{yellow-blue} \]

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

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

✨ not all colours can be created in this way.
What is a *digital* image?

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

a digital image is a rectangular array of intensity values

each value is called a *pixel*

“picture element”

sampling resolution is normally measured in pixels per inch (ppi) or dots per inch (dpi)

- computer monitors have a resolution around 100 ppi
- laser and ink jet printers have resolutions between 300 and 1200 ppi
- typesetters have resolutions between 1000 and 3000 ppi
Image capture

✿ a variety of devices can be used

◆ scanners
  ■ line CCD (charge coupled device) in a flatbed scanner
  ■ spot detector in a drum scanner

◆ cameras
  ■ area CCD
  ■ CMOS camera chips

The image of the Heidelberg drum scanner and many other images in this section come from “Handbook of Print Media”, by Helmut Kipphan, Springer-Verlag, 2001.
Image capture example

A real image

A digital image
Sampling resolution

256×256  128×128  64×64  32×32
Quantisation

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

- 8 bits (256 levels)
- 7 bits (128 levels)
- 6 bits (64 levels)
- 5 bits (32 levels)
- 4 bits (16 levels)
- 3 bits (8 levels)
- 2 bits (4 levels)
- 1 bit (2 levels)
Storing images in memory

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

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

- most computers have a special piece of memory reserved for storage of the current image being displayed
- the frame buffer normally consists of dual-ported Dynamic RAM (DRAM)
  - sometimes referred to as Video RAM (VRAM)
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

✦ Technology
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
More perspective

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

  ![Diagram of a ray intersecting a plane]

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

  plane: \( P \cdot N + d = 0 \)

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

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

- sphere

![Diagram of a ray intersecting a sphere]

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

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

- cylinder, cone, torus

  - all similar to sphere
  - try them as an exercise

\[
\begin{align*}
a &= D \cdot D \\
b &= 2D \cdot (O - C) \\
c &= (O - C) \cdot (O - C) - r^2 \\
d &= \sqrt{b^2 - 4ac} \\
s_1 &= \frac{-b + d}{2a} \\
s_2 &= \frac{-b - d}{2a}
\end{align*}
\]
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).
because you are tracing rays from the intersection point to the light, you can check whether another object is between the intersection and the light and is hence casting a shadow

also need to watch for self-shadowing
Ray tracing: reflection

- if a surface is totally or partially reflective then new rays can be spawned to find the contribution to the pixel’s colour given by the reflection

  - this is perfect (mirror) reflection
Ray tracing: transparency & refraction

- objects can be totally or partially transparent
  - this allows objects behind the current one to be seen through it

- transparent objects can have refractive indices
  - bending the rays as they pass through the objects

- transparency + reflection means that a ray can split into two parts
Illumination and shading

Dürer’s method allows us to calculate what part of the scene is visible in any pixel

But what colour should it be?

Depends on:
  - lighting
  - shadows
  - properties of surface material
How do surfaces reflect light?

perfect specular reflection (mirror)

Imperfect specular reflection

diffuse reflection (Lambertian reflection)

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

Johann Lambert, 18th century German mathematician
Comments on reflection

- the surface can absorb some wavelengths of light
  - e.g. shiny gold or shiny copper

- specular reflection has “interesting” properties at glancing angles owing to occlusion of micro-facets by one another

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

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

- **observation:**
  - the colour of a flat surface will be uniform across it, dependent only on the colour & position of the object and the colour & position of the light sources
Diffuse shading calculation

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

- \( L \) is a normalised vector pointing in the direction of the light source.
- \( N \) is the normal to the surface.
- \( I_l \) is the intensity of the light source.
- \( k_d \) is the proportion of light which is diffusely reflected by the surface.
- \( I \) is the intensity of the light reflected by the surface.

Use this equation to calculate the colour of a pixel.
Diffuse shading: comments

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

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

- \( L \) is a normalised vector pointing in the direction of the light source.
- \( R \) is the vector of perfect reflection.
- \( N \) is the normal to the surface.
- \( V \) is a normalised vector pointing at the viewer.
- \( I_l \) is the intensity of the light source.
- \( k_s \) is the proportion of light which is specularly reflected by the surface.
- \( n \) is Phong’s *ad hoc* “roughness” coefficient.
- \( I \) is the intensity of the specularly reflected light.

Examples

diffuse reflection

specular reflection

100% 75% 50% 25% 0%
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 I

- regular grid
  - divide the pixel into a number of sub-pixels and shoot a ray through the centre of each
  - problem: can still lead to noticeable 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 $\varepsilon$ to one another
  - for $N$ rays this produces a better looking image than pure random sampling
  - very hard to implement properly

![Poisson disc and pure random sampling comparison]
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 sampling](image)

![Poisson disc sampling](image)

![pure random sampling](image)
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
Computer Graphics & Image Processing

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

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

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

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

- Curved surfaces
  - must be converted to polygons to be drawn
Surfaces in 3D: polygons

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

![Diagram showing a non-planar "polygon" with one vertex in front of the other three, which are all in the same plane.]

Rotate the polygon about the vertical axis. Should the result be this or this?
Splitting polygons into triangles

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

which is preferable?
2D transformations

- **scale**
- **rotate**
- **translate**
- **(shear)**

**why?**
- It is extremely useful to be able to transform predefined objects to an arbitrary location, orientation, and size.
- Any reasonable graphics package will include transforms:
  - 2D ➔ Postscript
  - 3D ➔ OpenGL
Basic 2D transformations

- **scale**
  - about origin
  - by factor $m$
  
  \[ x' = mx \]
  \[ y' = my \]

- **rotate**
  - about origin
  - by angle $\theta$
  
  \[ x' = x \cos \theta - y \sin \theta \]
  \[ y' = x \sin \theta + y \cos \theta \]

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

- **do nothing**
  - identity
  $$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

- **rotate**
  - about origin, angle $\theta$
  $$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

- **shear**
  - parallel to $x$ axis, factor $a$
  $$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & a \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$
Homogeneous 2D co-ordinates

- translations cannot be represented using simple 2D matrix multiplication on 2D vectors, so we switch to homogeneous co-ordinates
  
  \[(x, y, w) \equiv \left( \frac{x}{w}, \frac{y}{w} \right)\]

- an infinite number of homogeneous co-ordinates map to every 2D point

- \(w=0\) represents a point at infinity

- usually take the inverse transform to be:
  
  \[(x, y) \equiv (x, y, 1)\]
Matrices in homogeneous co-ordinates

- **scale**
  - about origin, factor \( m \)
  \[
  \begin{bmatrix}
  x' \\
  y' \\
  w'
  \end{bmatrix} =
  \begin{bmatrix}
  m & 0 & 0 \\
  0 & m & 0 \\
  0 & 0 & 1
  \end{bmatrix}
  \begin{bmatrix}
  x \\
  y \\
  w
  \end{bmatrix}
  \]

- **do nothing**
  - identity
  \[
  \begin{bmatrix}
  x' \\
  y' \\
  w'
  \end{bmatrix} =
  \begin{bmatrix}
  1 & 0 & 0 \\
  0 & 1 & 0 \\
  0 & 0 & 1
  \end{bmatrix}
  \begin{bmatrix}
  x \\
  y \\
  w
  \end{bmatrix}
  \]

- **rotate**
  - about origin, angle \( \theta \)
  \[
  \begin{bmatrix}
  x' \\
  y' \\
  w'
  \end{bmatrix} =
  \begin{bmatrix}
  \cos \theta & -\sin \theta & 0 \\
  \sin \theta & \cos \theta & 0 \\
  0 & 0 & 1
  \end{bmatrix}
  \begin{bmatrix}
  x \\
  y \\
  w
  \end{bmatrix}
  \]

- **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

\[
x' = x + wx_o \quad y' = y + wy_o \quad w' = w
\]

In conventional coordinates

\[
\frac{x'}{w'} = \frac{x}{w} + x_o \quad \frac{y'}{w'} = \frac{y}{w} + y_0
\]
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:

\[
\begin{bmatrix}
  x'' \\
  y'' \\
  w''
\end{bmatrix} = \begin{bmatrix} m & 0 & 0 \\
  0 & m & 0 \\
  0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x' \\
  y' \\
  w' \end{bmatrix} \quad \begin{bmatrix} x' \\
  y' \\
  w' \end{bmatrix} = \begin{bmatrix} 1 & a & 0 \\
  0 & 1 & 0 \\
  0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\
  y \\
  w \end{bmatrix}
\]

\[
\begin{bmatrix}
  x'' \\
  y'' \\
  w''
\end{bmatrix} = \begin{bmatrix} m & 0 & 0 \\
  0 & m & 0 \\
  0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & a & 0 \\
  0 & 1 & 0 \\
  0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\
  y \\
  w \end{bmatrix} = \begin{bmatrix} m & ma & 0 \\
  0 & m & 0 \\
  0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\
  y \\
  w \end{bmatrix}
\]
Transformation are not commutative

_be careful of the order in which you concatenate transformations_

- **rotate by 45°**
- **scale by 2 along x axis**

*rotate then scale*

\[
\begin{bmatrix}
\frac{2}{\sqrt{2}} & -\frac{2}{\sqrt{2}} & 0 \\
\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 \\
0 & 0 & 1
\end{bmatrix}
\]

**scale**

\[
\begin{bmatrix}
2 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1
\end{bmatrix}
\]

- **scale then rotate**
- **rotate**

\[
\begin{bmatrix}
\frac{2}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & 0 \\
\frac{2}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 \\
0 & 0 & 1
\end{bmatrix}
\]

\[
\begin{bmatrix}
\frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & 0 \\
\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 \\
0 & 0 & 1
\end{bmatrix}
\]
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 $(x_o,y_o)$

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

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

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

Exercise: show how to perform rotation about an arbitrary point
3D transformations

3D homogeneous co-ordinates

\[(x, y, z, w) \rightarrow \left(\frac{x}{w}, \frac{y}{w}, \frac{z}{w}\right)\]

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**
  \[
  \begin{bmatrix}
  1 & 0 & 0 & 0 \\
  0 & \cos \theta & -\sin \theta & 0 \\
  0 & \sin \theta & \cos \theta & 0 \\
  0 & 0 & 0 & 1 \\
  \end{bmatrix}
  \]

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

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

- **scale**
  \[
  \begin{bmatrix}
  m_x & 0 & 0 & 0 \\
  0 & m_y & 0 & 0 \\
  0 & 0 & m_z & 0 \\
  0 & 0 & 0 & 1 \\
  \end{bmatrix}
  \]
3D transformations are not commutative

90° rotation about z-axis

90° rotation about x-axis

90° rotation about z-axis

90° rotation about x-axis

opposite faces
Model transformation I

- 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

\[
\begin{align*}
S &= \begin{bmatrix}
2 & 0 & 0 & 0 \\
0 & 1.5 & 0 & 0 \\
0 & 0 & 2 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix} \\
T &= \begin{bmatrix}
1 & 0 & 0 & 1.5 \\
0 & 1 & 0 & 3 \\
0 & 0 & 1 & 4 \\
0 & 0 & 0 & 1
\end{bmatrix}
\end{align*}
\]

- scale from size (2,2,2) to size (4,3,4)
- translate centre of cylinder from (0,0,0) to halfway between (1,2,3) and (2,4,5)
Model transformation 3

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

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

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

- original axis: \(y\)-axis = (0,1,0)
Model transformation 4

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

- zero the \(z\)-coordinate by rotating about the \(x\)-axis

\[
R_1 = \begin{bmatrix}
1 & 0 & 0 & 0 & 0 \\
0 & \cos \theta & -\sin \theta & 0 & 0 \\
0 & \sin \theta & \cos \theta & 0 & 0 \\
0 & 0 & 0 & 0 & 1
\end{bmatrix}
\]

\[
\theta = -\arcsin \frac{2}{\sqrt{2^2 + 2^2}}
\]
Model transformation 5

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

$$R_2 = \begin{bmatrix}
\cos \varphi & -\sin \varphi & 0 & 0 \\
\sin \varphi & \cos \varphi & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}$$

$$\varphi = \arcsin \frac{1}{\sqrt{1^2 + \sqrt{8}^2}}$$
the overall transformation is:

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

\[
\begin{bmatrix}
x'
y'
z'
w'
\end{bmatrix} = T \times R_1^{-1} \times R_2^{-1} \times S \times \begin{bmatrix}
x y z \\
w
\end{bmatrix}
\]
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 (27°, 156°)
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

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

\[
x' = x \frac{d}{z}
\]

\[
y' = y \frac{d}{z}
\]

\[
z' = \frac{1}{z}
\]

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

Remember:

\[
\begin{bmatrix}
  x \\
  y \\
  z \\
  w
\end{bmatrix}
\rightarrow
\begin{bmatrix}
  x/w \\
  y/w \\
  z/w
\end{bmatrix}
\]
Perspective projection with an arbitrary camera

- We have assumed that:
  - Screen centre at \((0,0,d)\)
  - Screen parallel to \(xy\)-plane
  - \(z\)-axis into screen
  - \(y\)-axis up and \(x\)-axis to the right
  - Eye (camera) at origin \((0,0,0)\)

- For an arbitrary camera we can either:
  - Work out equations for projecting objects about an arbitrary point onto an arbitrary plane
  - Transform all objects into our standard co-ordinate system (viewing co-ordinates) and use the above assumptions
A variety of transformations

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

钯 the problem:

◆ to transform an arbitrary co-ordinate system to the default viewing co-ordinate system

钯 camera specification in world co-ordinates

◆ eye (camera) at \((e_x, e_y, e_z)\)
◆ look point (centre of screen) at \((l_x, l_y, l_z)\)
◆ up along vector \((u_x, u_y, u_z)\)
  ■ perpendicular to \(el\)
Viewing transformation 2

- translate eye point, \((e_x,e_y,e_z)\), to origin, \((0,0,0)\)
  
  \[
  T = \begin{bmatrix}
  1 & 0 & 0 & -e_x \\
  0 & 1 & 0 & -e_y \\
  0 & 0 & 1 & -e_z \\
  0 & 0 & 0 & 1
  \end{bmatrix}
  \]

- scale so that eye point to look point distance, \(|\vec{el}|\), is distance from origin to screen centre, \(d\)

  \[
  |\vec{el}| = \sqrt{(l_x - e_x)^2 + (l_y - e_y)^2 + (l_z - e_z)^2}
  \]

  \[
  S = \begin{bmatrix}
  \frac{d}{|\vec{el}|} & 0 & 0 & 0 \\
  0 & \frac{d}{|\vec{el}|} & 0 & 0 \\
  0 & 0 & \frac{d}{|\vec{el}|} & 0 \\
  0 & 0 & 0 & 1
  \end{bmatrix}
  \]
Viewing transformation 3

◆ need to align line \( \overline{el} \) with \( z \)-axis
  - first transform \( e \) and \( l \) into new co-ordinate system
    \[
    e'' = S \times T \times e = 0 \quad l'' = S \times T \times l
    \]
  - then rotate \( e''l'' \) into \( yz \)-plane, rotating about \( y \)-axis
    \[
    R_1 = \begin{bmatrix}
    \cos \theta & 0 & -\sin \theta & 0 \\
    0 & 1 & 0 & 0 \\
    \sin \theta & 0 & \cos \theta & 0 \\
    0 & 0 & 0 & 1
    \end{bmatrix}
    \]
    \[
    \theta = \arccos \frac{l''_z}{\sqrt{l''_x^2 + l''_z^2}}
    \]
    \[
    \left( \begin{array}{c}
    0, l''_y, \sqrt{l''_x^2 + l''_z^2} \\
    \end{array} \right)
    \]
Viewing transformation 4

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

\[ l''' = R_1 \times l'' \]

\[
R_2 = \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & \cos \varphi & -\sin \varphi & 0 \\
0 & \sin \varphi & \cos \varphi & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

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

\[
(0,0,\sqrt{l'''_y^2 + l'''_z^2}) = (0,0,d)
\]

\[
(0,l'''_y, l'''_z)
\]
Viewing transformation 5

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

\[ \mathbf{u'''} = \mathbf{R}_2 \times \mathbf{R}_1 \times \mathbf{u} \]

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

\[
\psi = \arccos \frac{\mathbf{u'''}}{\sqrt{u'''^2 + u'''^2}}
\]

why don’t we need to multiply \( \mathbf{u} \) by \( \mathbf{S} \) or \( \mathbf{T} \)?

\( \mathbf{u} \) is a vector rather than a point, vectors do not get translated

scaling \( \mathbf{u} \) by a uniform scaling matrix would make no difference to the direction in which it points
we can now transform any point in world co-ordinates to the equivalent point in viewing co-ordinate

\[
\begin{bmatrix}
    x' \\
    y' \\
    z' \\
    w'
\end{bmatrix}
= R_3 \times R_2 \times R_1 \times S \times T \times
\begin{bmatrix}
    x \\
    y \\
    z \\
    w
\end{bmatrix}
\]

in particular: \( e \rightarrow (0,0,0) \quad l \rightarrow (0,0,d) \)

the matrices depend only on \( e, l, \) and \( u, \) so they can be pre-multiplied together

\[
M = R_3 \times R_2 \times R_1 \times S \times T
\]
Illumination & shading

- Drawing polygons with uniform colours gives poor results
- Interpolate colours across polygons
Illumination & shading

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

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

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

Flat vs Gouraud shading

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

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

- similar to Gouraud shading, but calculate the specular component in addition to the diffuse component
- therefore need to interpolate the normal across the polygon in order to be able to calculate the reflection vector

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

\[(x_1', y_1'), z_1, (r_1, g_1, b_1), N_1\]
\[(x_2', y_2'), z_2, (r_2, g_2, b_2), N_2\]
\[(x_3', y_3'), z_3, (r_3, g_3, b_3), N_3\]
Introduction to Computer Graphics

- Background
- Rendering
- Graphics pipeline
- Graphics hardware and modern OpenGL
  - GPU & APIs
  - OpenGL Rendering pipeline
  - Example OpenGL code
  - GLSL
  - Transformations & vertex shaders
  - Raster buffers
  - Textures
- Colour
What is a GPU?

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

Intell 8-core Core i7 Haswell-E
2,600,000,000 transistors

Nvidia GeForce GTX Titan X
8,000,000,000 transistors
What does a GPU do

- Performs all low-level tasks & a lot of high-level tasks
  - Clipping, rasterisation, hidden surface removal, …
    - Essentially draws millions of triangles very efficiently
  - Procedural shading, texturing, animation, simulation, …
  - Video rendering, de- and encoding, deinterlacing, …
  - Physics engines

- Full programmability at several pipeline stages
  - 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
  - >>400 GB/s memory access
    - This is much higher bandwidth than CPU
    - But peak performance can be expected for very specific operations
## GPU APIs

**(Application Programming Interfaces)**

<table>
<thead>
<tr>
<th>OpenGL</th>
<th>DirectX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multi-platform</td>
<td>Microsoft Windows / Xbox</td>
</tr>
<tr>
<td>Open standard API</td>
<td>Proprietary API</td>
</tr>
<tr>
<td>Focus on general 3D applications</td>
<td>Focus on games</td>
</tr>
<tr>
<td></td>
<td>Application manages resources</td>
</tr>
<tr>
<td></td>
<td>Open GL driver manages the resources</td>
</tr>
</tbody>
</table>

- Similar functionality and performance
One more API

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

- OpenGL and DirectX are not meant to be used for general purpose computing
  - Example: physical simulation, 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 CPU
  - Supported by AMD, Intel and Nvidia, Qualcomm, Apple, …
GPU and mobile devices

- **OpenGL ES 1.0-3.2**
  - Stripped version of OpenGL
  - Removed functionality that is not strictly necessary on mobile devices

- **Devices**
  - iOS: 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
  - Chrome and Firefox (2017)
- Most modern browsers support WebGL
- Potentially could be used to create 3D games in a browser
  - and replace Adobe Flash

http://zygotebody.com/
OpenGL in Java

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

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

- Proprietary library IRIS GL by SGI
- OpenGL 1.0 (1992)
- OpenGL 1.2 (1998)
- OpenGL 2.0 (2004)
  - GLSL
  - Non-power-of-two (NPOT) textures
- OpenGL 3.0 (2008)
  - Major overhaul of the API
  - Many features from previous versions depreciated
- OpenGL 3.2 (2009)
  - Core and Compatibility profiles
- OpenGL 4.0 (2010)
  - Geometry shaders
- OpenGL 4.5 (2014)
- OpenGL 4.6 (2017)
  - SPIR-V shaders
OpenGL rendering pipeline
# OpenGL programming model

<table>
<thead>
<tr>
<th>CPU code</th>
<th>GPU code</th>
</tr>
</thead>
<tbody>
<tr>
<td>- gl* functions that</td>
<td>- Fragment shaders</td>
</tr>
<tr>
<td>- Create OpenGL objects</td>
<td>- Vertex shaders</td>
</tr>
<tr>
<td>- Copy data CPU&lt;-&gt;GPU</td>
<td>- and other shaders</td>
</tr>
<tr>
<td>- Modify OpenGL state</td>
<td>- Written in GLSL</td>
</tr>
<tr>
<td>- Enqueue operations</td>
<td>- Similar to C</td>
</tr>
<tr>
<td>- Synchronize CPU &amp; GPU</td>
<td>- From OpenGL 4.6 could be written in other language and compiled to</td>
</tr>
<tr>
<td>- C99 library</td>
<td>SPIR-V</td>
</tr>
<tr>
<td>- Wrappers in most programming language</td>
<td></td>
</tr>
</tbody>
</table>
OpenGL rendering pipeline

- Vertex data
- Vertex shader
- Tessellation control shader
- Tessellation evaluation shader
- Rasterization
- Clipping
- Primitive assembly
- Geometry shader
- Fragment shader
- Screen buffer
- Programmable stages
- Fixed stages
OpenGL rendering pipeline

Vertex data → Vertex shader → Tessellation control shader → Tessellation evaluation shader

Rasterization → Primitive assembly → Geometry shader

Fragment shader

Processing of vertices, normals, uv texture coordinates.

Programmable stages

Fixed stages
OpenGL rendering pipeline

- **Vertex data**
- **Vertex shader**
- **Tessellation control shader**
- **Tessellation evaluation shader**
- **Geometry shader**
- **Clipping**
- **Rasterization**
- **Fragment shader**
- **Screen buffer**

[Optional] Create new primitives by tessellating existing primitives (patches).

Programmable stages

Fixed stages
OpenGL rendering pipeline


Vertex data

Vertex shader

Tessellation control shader

Tessellation evaluation shader

Geometry shader

Clipping

Rasterization

Primitives assembly


Fragment shader

Screen buffer

Programmable stages

Fixed stages

fur

shadow volumes
OpenGL rendering pipeline

Vertex data → Vertex shader

Tessellation

control shader → evaluation shader

Tessellation

Primitive assembly

Geometry shader

Clipping

Rasterization

Fragment shader

Screen buffer

Programmable stages

Fixed stages

Organizes vertices into primitives and prepares them for rendering.
OpenGL rendering pipeline

1. **Vertex data**
2. **Vertex shader**
3. **Tessellation control shader**
4. **Tessellation evaluation shader**
5. **Geometry shader**
6. **Clipping**
7. **Rasterization**
8. **Primitive assembly**
9. **Geometry shader**
10. **Fragment shader**
11. **Screen buffer**

Remove or modify vertices so that they all lie within the viewport (view frustum).

**Programmable stages**

**Fixed stages**
Generates fragments (pixels) to be drawn for each primitive. Interpolates vertex attributes.

scanlines
OpenGL rendering pipeline

- **Vertex data**
- **Rasterization**
- **Fragment shader**
- **Screen buffer**

**Programmable stages**

- **Vertex shader**
- **Tessellation control shader**
- **Tessellation evaluation shader**
- **Geometry shader**

**Fixed stages**

- **Clipping**
- **Rasterization**

**Screen buffer**

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

Physically accurate materials

Non-Photorealistic-Rendering shader

Programmable stages

Fixed stages
Managing buffers/objects in OpenGL

- Generating names
  - “name” is like a reference in Java
  - `glGen*` functions create names WITHOUT allocating the actual object
  - From OpenGL 4.5: `glCreate*` functions create names AND allocate actual object

- Binding objects
  - `glBind*` functions
  - Performs two operations
    - Allocates memory for a particular object (if it does not exist)
    - Makes this object active in the current OpenGL Context
  - Functions operating on OpenGL objects will change the currently bound (or active) object
Managing buffers/objects in OpenGL

- **Unbinding objects**
  - Passing “0” instead of “name” unbinds the active object
  - `glBind( ..., 0 )`

- **Deleting object**
  - `glDelete*` functions
  - Deletes both the object and its name
Geometry objects in OpenGL (OO view)
OpenGL as a state-machine

If OpenGL was OO API:

VertexArray va = new VertexArray();

ArrayBuffer vertices = new ArrayBuffer( my_data );

va.add( vertices );

But it is not, and you must do:

int va = glGenVertexArrays();

glBindVertexArray(va); // va becomes “active” VertexArray

int vertices = glGenBuffers();

glBindBuffer(GL_ARRAY_BUFFER, vertices); // This adds vertices to currently bound VertexArray
OpenGL example code - overview
Let us draw some triangles

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

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

Initialize OpenGL

Set up inputs

Draw a frame

Free resources
A more complete example

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

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

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

- The OpenGL functions and constants can be found in the LWJGL packages:
  - org.lwjgl.opengl.GL11
  - org.lwjgl.opengl.GL15
  - org.lwjgl.opengl.GL20
  - org.lwjgl.opengl.GL30
  - ...

- For simplicity, package names are omitted in all examples shown in these slides
Let us draw some triangles

Initialize OpenGL

Set up inputs

Draw a frame

Free resources

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

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

// Step 2: Clear the buffer
glClearColor(1.0f, 1.0f, 1.0f, 1.0f); // Set the background colour to dark grey
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
// Step 3: Draw our VertexArray as triangles

glBindVertexArray(vertexArrayObj); // Bind the existing VertexArray object

glDrawElements(GL_TRIANGLES, no_of_triangles, GL_UNSIGNED_INT, 0); // Draw it as triangles

glBindVertexArray(0); // Remove the binding

// Step 4: Swap the draw and back buffers to display the rendered image

glfwSwapBuffers(window);

glfwPollEvents();
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

```glsl
#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()
{
    // Typically 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

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

- Size can be checked with “length()”

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

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

Example: `const float pi = 3.14;`
Shader inputs and outputs

**Vertex shader**
- in vec3 position
- in vec3 normal
- out vec3 frag_normal
- Vertex attribute interpolation

**Fragment shader**
- in vec3 frag_normal
- out vec3 colour
- [optional] layout(location=?) in GLSL

**ArrayBuffer (vertices)**
- in vec3 position
- in vec3 normal

**ArrayBuffer (normals)**
- out vec3 frag_normal

**FrameBuffer (pixels)**
- out vec3 colour

**GLSL functions**
- glGetAttribLocation
- glBindBuffer
- glVertexAttribPointer
- glEnableVertexAttribArray
- glBindFragDataLocation

(see the next slide)
How to specify input to a vertex shader?

```c
// Get the locations of the "position" vertex attribute variable in our shader
int position_loc = glGetAttribLocation(shaders_handle, "position");

// If the vertex attribute found
if (position_loc != -1) {
    // Activate the ArrayBuffer that should be accessed in the shader
    glBindBuffer(GL_ARRAY_BUFFER, vertex_handle);
    // Specifies where the data for "position" variable can be accessed
    glVertexAttribPointer(position_loc, 3, GL_FLOAT, false, 0, 0);
    // Enable that vertex attribute variable
    glEnableVertexAttribArray(position_loc);
}
```
Passing uniform(s) to a shader

- **In shader:**
  
  ```
  uniform mat4 mvp_matrix; // model-view-projection matrix
  ```

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

Name of the method depends on the data type. For example, `glUniformMatrix3fv` for `Vector3f`
GLSL Operators

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

- Assignment:
  float a=0;
  a += 2.0; // Equivalent to a = a + 2.0

See the quick reference guide at: [https://www.opengl.org/documentation/glsl/](https://www.opengl.org/documentation/glsl/)

159
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/](https://www.opengl.org/documentation/glsl/)
GLSL flow control

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

for( int i = 0; i<10; i++ ) {
    
    ...
    }

while( n < 10 ) {
    
    ...
    }

switch( int_value ) {
    case n:
        // statements
        break;
    case m:
        // statements
        break;
    default:
        
        do {
            ...
            
        } while ( n < 10 )
        
    }

}
Transformations (Vertex shaders)
Model, View, Projection matrices

Object coordinates

Object centred at the origin

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

Model matrix

World coordinates
Model, View, Projection matrices

World coordinates

View matrix

To position all objects relative to the camera

View (camera) coordinates

Camera at the origin, pointing at -z
Model, View, Projection matrices

The default OpenGL coordinate system is right-handed.

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

x and y must be in the range -1 and 1.
All together

Screen coordinates
$x_s/w_s$ and $y_s/w_s$ must be between -1 and 1

Projection, view and model matrices

3D world vertex coordinates

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

- Transformation by a nonorthogonal matrix does not preserve angles

- Since:
  \[ N \cdot T = 0 \]
  \[ N' \cdot T' = (GN) \cdot (MT) = 0 \]

- We can find that: \( G = (M^{-1})^T \)
  - Derivation shown on the visualizer
Raster buffers (colour, depth, stencil)
Render buffers in OpenGL

Colour:
- GL_FRONT
- GL_BACK

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.

Four components:
- RGBA

Typically 8 bits per component
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
  - `glfwSwapInterval(1);`
  - `glSwapBuffers()` waits until the last row is copied to the display.
No V-Sync vs. V-Sync

No V-Sync

GPU

Frame 1

Frame 2

Frame 3

Frame 4

Display

Scan 1

Scan 2

Scan 3

Scan 4

Time [ms]

0

16

32

48

Tear

Tear

Tear

Tear

V-Sync

GPU

Frame 1

Frame 2

Frame 3

Display

Scan 1

Scan 2

Scan 3

Scan 4

Lag

Stutter (same frame displayed)

Time [ms]

0

16

32

48
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
Z-Buffer - algorithm

- Initialize the depth buffer and image buffer for all pixels
  \[ \text{color}(x, y) = \text{Background\_Color}, \]
  \[ \text{depth}(x, y) = \text{z\_far} \quad // \quad \text{position of the far clipping plane} \]
- For every triangle in a scene
  - For every fragment \((x, y)\) representing this triangle
    - Calculate \(z\) for current \((x, y)\)
    - \(\text{if} \ (z < \text{depth}(x, y))\)
      - \(\text{depth} (x, y) = z\)
      - \(\text{color} (x, y) = \text{Polygon\_Color} (x, y)\)
View frustum

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

Z-buffer must store all these values
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.

1D

2D

3D

CUBE_MAP

Used for environment mapping
Texture mapping

1. Define your texture function (image) $T(u,v)$
2. $(u,v)$ 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

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

Nearest neighbour

Pick the nearest texel: D

Bilinear interpolation

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

nearest-neighbour

bilinear
Up-sampling

- if one pixel in the texture map covers several pixels in the final image, you get visible artefacts.

- only practical way to prevent this is to ensure that texture map is of sufficiently high resolution that it does not happen.
Down-sampling

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

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

without area averaging

with area averaging
Texture tiling

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

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

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

- Stores texture data
- Mipmap
  - min_filter : int
  - max_filter : int
  - wrap_s : int
  - wrap_t : int

- Texture
  - min_filter : int
  - max_filter : int
  - wrap_s : int
  - wrap_t : int

- Texture1D
  - width : int

- Texture2D
  - width : int
  - height : int

- Texture3D
  - width : int
  - height : int
  - depth : int

- TextureUnit
  - index : int

- Sampler
  - min_filter : int
  - mag_filter : int
  - wrap_s : int
  - wrap_t : int

- SamplerUnit
  - index : int

- Defines how the texels are looked up in Textures
- Hardware units that performs sampling

- Hardware unit for reading texture in fragment shader
Setting up a texture

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

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

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

//Setup filtering, i.e. how OpenGL will interpolate the pixels when scaling up or down

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_NEAREST);

//Setup wrap mode, i.e. how OpenGL will handle pixels outside of the expected range

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
Fragment shader

#version 330
uniform sampler2D texture_diffuse;
in vec2 frag_TextureCoord;

out vec4 out_Color;

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

Rendering

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

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

.glBindTexture(GL_TEXTURE_2D, 0);

Frame Buffer Objects
Frame Buffer Objects (FBOs)

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

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

- Post-processing, tone-mapping, blooming, etc.
- Reflections (in water), animated textures (e.g. TV screen)
- When the result of rendering is not shown (e.g. saved to disk)
Create FBO, attach a Texture (colour) and a RenderBuffer (depth)

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

int myFBO = glGenFramebuffers();
glBindFramebuffer(GL_FRAMEBUFFER, myFBO);
//Attach 2D texture to this FBO
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, color_tex, 0);
```
FBO: Code example 2/3

```c
int depth_rb = glGenRenderbuffers();
glBindRenderbuffer(GL_RENDERBUFFER, depth_rb);
glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT24, 256, 256);

//Attach depth buffer to FBO
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_RENDERBUFFER, depth_rb);
```
FBO: Code example 3/3

- **Render**

  ```gl
  glBindFramebuffer(GL_FRAMEBUFFER, myFBO);
  glClearColor(0.0, 0.0, 0.0, 0.0);
  glClearDepth(1.0f);
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  
  // Render
  
  glBindFramebuffer(GL_FRAMEBUFFER, 0);
  ```
References


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

- Google search: „man gl.....”
Colour and colour spaces
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

  \[ \text{colour} = \text{perception}(\text{spectral_emission}) \]

- For reflective displays / objects

  \[ \text{colour} = \text{perception}(\text{illumination} \ast \text{reflectance}) \]
Black body radiation

- Electromagnetic radiation emitted by a perfect absorber at a given temperature
- Graphite is a good approximation of a black body
Correlated colour temperature

- The temperature of a black body radiator that produces light most closely matching the particular source.

- Examples:
  - Typical north-sky light: 7500 K
  - Typical average daylight: 6500 K
  - Domestic tungsten lamp (100 to 200 W): 2800 K
  - Domestic tungsten lamp (40 to 60 W): 2700 K
  - Sunlight at sunset: 2000 K

- Useful to describe colour of the illumination (source of light).
Standard illuminant D65

- Mid-day sun in Western Europe / Northern Europe
- Colour temperature approx. 6500 K
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

Why not red?
Reflected light

\[ L(\lambda) = I(\lambda) \cdot R(\lambda) \]

- Reflected light = illumination * reflectance

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

- Cones are the photoreceptors 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
Perceived light

- **cone response** = \( \text{sum}(\text{sensitivity} \times \text{reflected light}) \)

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:**

\[
\begin{align*}
\text{II} &= [L_1, M_1, S_1] \\
\text{II} &= [L_2, M_2, S_2]
\end{align*}
\]
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

\[ [L_1, M_1, S_1] \]
\[ \parallel \]
\[ [L_2, M_2, S_2] \]
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 monochromatic 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)

- **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
    - One primary is roughly proportionally to light intensity
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
Luminance – photometric quantity

- Luminance – perceived brightness of light, adjusted for the sensitivity of the visual system to wavelengths

Luminance

\[ L_v = \int_0^\infty L(\lambda) \cdot V(\lambda) d\lambda \]

ISO Unit: cd/m²
CIE chromaticity diagram

- **chromaticity values** are defined in terms of $x$, $y$, $z$
  
  \[
  x = \frac{X}{X+Y+Z}, \quad y = \frac{Y}{X+Y+Z}, \quad z = \frac{Z}{X+Y+Z} \quad \therefore \quad x + y + z = 1
  \]

  - ignores luminance
  - can be plotted as a 2D function
  - pure colours (single wavelength) lie along the outer curve
  - all other colours are a mix of pure colours and hence lie inside the curve
  - points outside the curve do not exist as colours
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
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 so that the (Euclidean) colour differences are approximately perceptually uniform
    - CIE Lab, CIE Luv
**RGB space**

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

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

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

- HDR / physical rendering
  - Linear colour
  - Tone mapping
    - Gamma-corrected colour
  - Digital signal

- Pixel values
  - 0 to 255
  - Gamma corrected
  - RGB values
    - Luma
- Physical values
  - 0.0001 to 10000
  - Linear Radiance Luminance
  - Emitted light
Linear vs. gamma-corrected values
Basic display model: gamma correction

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

\[ V_{out} = \alpha \cdot V_{in}^{\gamma} \]

For color images: \( R = \alpha \cdot (R')^{\gamma} \) and the same for green and blue
Why is gamma needed?

“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 luminance are different quantities despite similar formulas
sRGB colour space

- “RGB” colour space is not a standard. Colours may differ depending on the choice of the primaries
- “sRGB” is a standard colour space, which most displays try to mimic (standard for HDTV)

<table>
<thead>
<tr>
<th>Chromaticity</th>
<th>Red</th>
<th>Green</th>
<th>Blue</th>
<th>White point</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0.6400</td>
<td>0.3000</td>
<td>0.1500</td>
<td>0.3127</td>
</tr>
<tr>
<td>y</td>
<td>0.3300</td>
<td>0.6000</td>
<td>0.0600</td>
<td>0.3290</td>
</tr>
<tr>
<td>z</td>
<td>0.0300</td>
<td>0.1000</td>
<td>0.7900</td>
<td>0.3583</td>
</tr>
</tbody>
</table>

- The chromacities above are also known as Rec. 709
sRGB colour space

- Two step XYZ to sRGB transformation:
  - Step 1: Linear color transform

\[
\begin{bmatrix}
R_{\text{linear}} \\
G_{\text{linear}} \\
B_{\text{linear}}
\end{bmatrix} = \begin{bmatrix}
3.2406 & -1.5372 & -0.4986 \\
-0.9689 & 1.8758 & 0.0415 \\
0.0557 & -0.2040 & 1.0570
\end{bmatrix} \begin{bmatrix}
X \\
Y \\
Z
\end{bmatrix}
\]

- Step 2: Non-linearity

\[
C_{\text{srgb}} = \begin{cases}
12.92C_{\text{linear}}, & C_{\text{linear}} \leq 0.0031308 \\
(1 + a)C_{\text{linear}}^{1/2.4} - a, & C_{\text{linear}} > 0.0031308
\end{cases}
\]

\[a = 0.055\]
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 the 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**
\[ L^* = \begin{cases} \left(\frac{29}{3}\right)^3 \frac{Y}{Y_n}, & Y/Y_n \leq \left(\frac{6}{29}\right)^3 \\ 116(Y/Y_n)^{1/3} - 16, & Y/Y_n > \left(\frac{6}{29}\right)^3 \end{cases} \]

**Chromacity coordinates**
\[ u^* = 13L^* \cdot (u' - u'_n) \]
\[ v^* = 13L^* \cdot (v' - v'_n) \]

- Hue and chroma

\[ C_w^* = \sqrt{(u^*)^2 + (v^*)^2} \]
\[ h_{uw} = \arctan2(v^*, u^*) \]
CIE $L^*a^*b^*$ colour space

- Another approximately perceptually uniform colour space

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

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

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

$$f(t) = \begin{cases} \frac{3t}{\delta^3} + \frac{4}{29} & \text{if } t > \delta^3 \\ \frac{6}{29} & \text{otherwise} \end{cases}$$

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

- Chroma and hue

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

Trichromatic values of the white point, e.g.

$$X_n = 95.047,$$

$$Y_n = 100.000,$$

$$Z_n = 108.883$$
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
Colour - references

- Chapters „Light” and „Colour” in

- Textbook on colour appearance