Computer Graphics

Computer graphics is part of a larger field that is visual computing, this includes
- Computer graphics and computer vision.
- Image capture and image display.

Computer graphics includes:
- User interface
- 3D renders
- AR/VR
- Special effects, including image editing

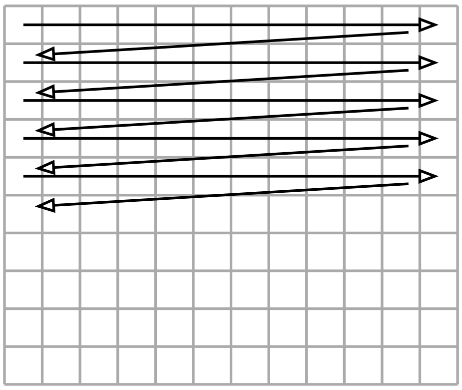
Image as a 2D Array

From a discrete perspecitive, an image is a **2D array of pixels**. The memory is not two-dimensional, how do we store it in memory? (how to linearise an image?)

Row Major

The first row is stored before the second row. The index of a pixel

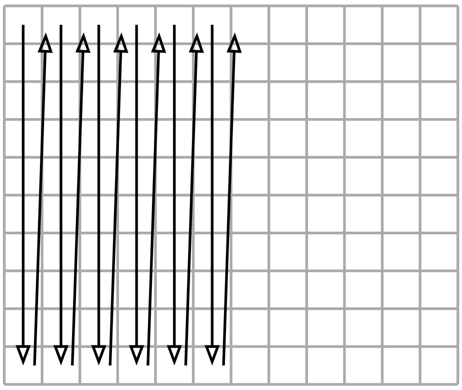
$$i(x, y) = x + y \cdot n_{\text{cols}}$$



Column Major

The first column is stored before the second column. The index of a pixel

$$i(x, y) = x \cdot n_{\text{rows}} + y$$



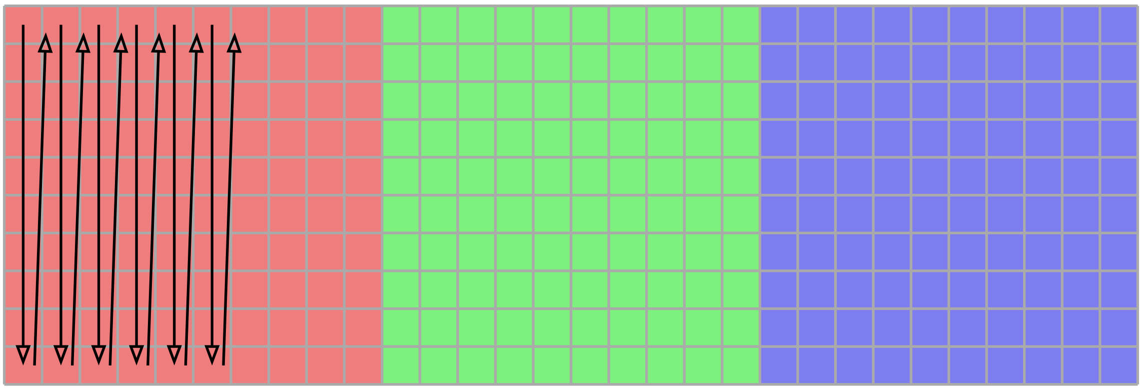
CRT TVs draw images from top to bottom, which is why most APIs represent images in row major.

RGB Representations

- **Interleaved, row major** stores all colours of a pixel next to each other, before moving on to the next pixel.

$$i(x, y, c) = 3x + 3y \cdot n_{\text{cols}} + c$$

- **Planar, column major** lays the R, G and B image next to each other, then use column major on the combined image.



$$i(x, y, c) = x \cdot n_{\text{rows}} + y + cxy$$

The general formula is $i(x, y, c) = xs_x + ys_y + cs_c$, where s_x is the number of pixels for each step in the x direction.

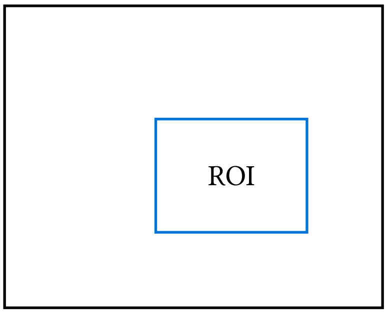
Padded Image

An algorithm that operates on neighbouring pixels need a padded image so every pixel in the image has neighbouring pixels.

The pixels in the region of interest are given by

$$i(x, y, c) = i_{\text{first pixel}} + xs_x + ys_y + cs_c$$

Where $i_{\text{first pixel}}$ is the index of the top left pixel of the ROI.



Pixels

Pixel is short for **picture element**.

Each pixel consist of 3 values R, G, B describing the colour.
- 0-255 for each colour, because it is convenient to use 1 byte per colour.
- It is also the number of colours we need to have no visible artifacts.

Colour Banding

> **Definition**
> **Colour banding** is when there are not enough bits to reprsent colour.

This is very visible to our eyes due to the mach band/chevreul illution where our eyes enhances the contrast of an edge, making the band more visible than it is.

> **Definition**
> **Dithering** adds noise to reduce banding: randomly add a value to the pixel value, the probability determined by the colour of the pixel.

Image as a Continuous Function

From a continuous perspective, an image is a continuous 2D function. This allows mathematical functions to run on the image.

> **Definition**
> A **pixel** is a point with no dimension.

We don't have continuous memory in computers, so we need to
- **Sample** the points, and
- **Quantise** the level of allowed values.

Rendering

Depth Perception

Our eyes will use anything to infer the depth of a scene, including:
- Occlusion: where one thing covers another because it is in front of the other thing.
- Relative size
- Distance to the horizon: the closer to the horizon the further away it is.
- Infer the shape of an object from shading.
- Red objects look closer than blue objects: chromatic abberation focus it further back in retina.
- Atmosphere/focus
- Perspective, this is easier when there are parallel lines.

In CG we want to use all the cues we can give.

Raytracing

We identify points on the image surface and calculate illumation for each pixel.
1. Shoot many rays from the camera.
2. Then calculate the colour of the pixel from where the ray hits, which is the closest intersection point (of the ray) to the camera.

Raytracing can easily handle reflection, refraction, shadows and motion blur, but is computationally expensive.

Finding Intersection

$$\text{ray} : \boldsymbol{r} = \boldsymbol{o} + s\boldsymbol{d}$$
$$\text{plane} : \boldsymbol{r} \cdot \boldsymbol{n} + a = 0$$

After solving

$$s = -\frac{a + \boldsymbol{n} \cdot \boldsymbol{o}}{\boldsymbol{n} \cdot \boldsymbol{d}}$$

To find an intersection with a polygon
1. Find intersections with the plane
2. Check whether the point is inside the polygon, which is just 2D geometry.

In the real word, the rays comes from the object
- It is computationally inefficient as a lot of the rays don't hit the eye.
- It can be mathematically proven that the result is the same regardless if you trace from the source to the eye or the other way round.

3D Object Intersection

Sphere

- Ray: $\boldsymbol{r} = \boldsymbol{o} + s\hat{\boldsymbol{d}}$
- Sphere: $(\boldsymbol{r} - \boldsymbol{c})^2 = r^2$

$$\left(\boldsymbol{o} - s\hat{\boldsymbol{d}} - \boldsymbol{c}\right)^2 - r^2 = 0$$
$$\hat{\boldsymbol{d}}^2 s^2 - 2s\hat{\boldsymbol{d}} \cdot (\boldsymbol{o} - \boldsymbol{c}) - r^2 = 0$$

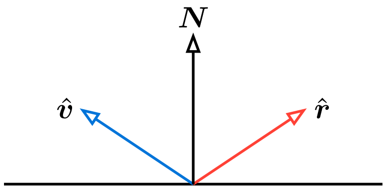
This is the quadratic equation, solve for s.
- If there are 2 real solutions, then cloose the closer intersection.

- If there are 0 real solutions, then there is no intersection.

You can also find intersections with a cylinder, cone and torus, it's easier if the objet is **axis aligned**.

Shading

1. Calculate the normal to the object at intersection.
2. Continue to look for a light source.
   - If the reflected ray intersect with another object, then the surface is not illuminated by the source. This is called a **shadow ray**.
3. If a surface is reflective, spawn a new ray to find the pixel's colour given by reflection.

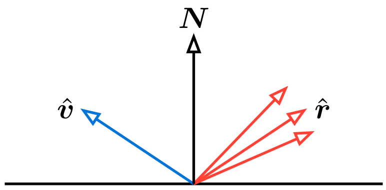


$$\hat{\boldsymbol{v}} \cdot \boldsymbol{N} = -\hat{\boldsymbol{r}} \cdot \boldsymbol{N}$$
$$\hat{\boldsymbol{r}} = -\hat{\boldsymbol{v}} + 2\boldsymbol{N}(\hat{\boldsymbol{v}} \cdot \boldsymbol{N})$$

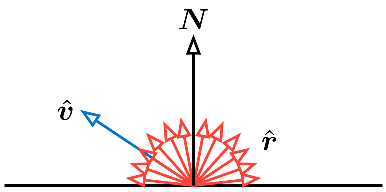
If we have a material that is both reflective and transparent, e.g. glass, then light will be refracted.
- 80% colour comes from reflection.
- 20% colour comes from refraction.

Types of Reflection

1. **Perfect reflection** as shown above.
2. **Imperfect specular reflection** (where the surface is not perfectly flat)



3. **Diffuse reflection** (where the structure is so complex light scatters in random direction)



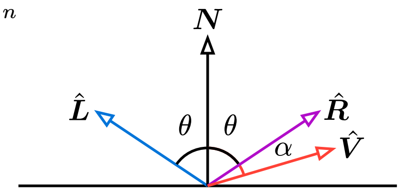
E.g. **plastic** has specular reflection on the light's colour, and diffuse reflection on the plastic's colour. Different wavelengths of light may reflect/scatter differently.

Phong's Imperfection Model

Definitions

Keyword	Definition
$\hat{\boldsymbol{L}}$	Normalised vector in direction of light source.
$\widehat{\boldsymbol{N}}$	Normal vector of the plane.
I_l	The intensity of light source.
k_d	The proportion of light diffusely reflected by the surface.
I	The intensity of the light being reflected.

$$I = I_l k_s \cos^n \alpha = I_l k_s \left(\hat{\boldsymbol{R}} \cdot \hat{\boldsymbol{L}}\right)^n$$



n determines how spread out the reflected light is - it is the **roughness factor**.

Overall Shading Equation

$$I = I_a k_d + \sum_i I_i k_d \hat{\boldsymbol{L}} \cdot \widehat{\boldsymbol{N}} + \sum_i I_i k_s \left(\hat{\boldsymbol{R}} \cdot \hat{\boldsymbol{V}}\right)^n$$

The $I_a k_d$ term gives the **ambient** shading. The next two terms gives the diffuse and specularity.

Sampling

So far we assume each ray passes through the centre of a pixel. This can lead to

- Jagged edges.
- Small objects being missed.
- Thin objects being split into pieces.

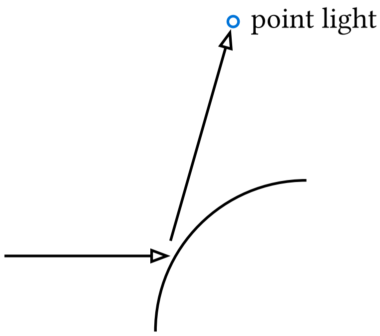
Antialiasing

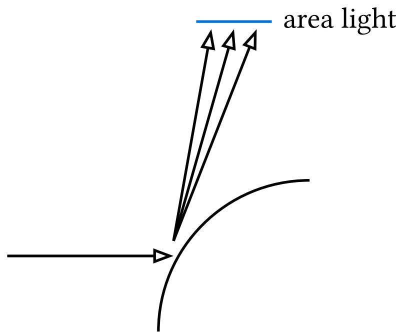
Definition
Artifacts are also known as **aliasing**.

- **Single point sampling** just samples the point at the centre.
- **Super sampling** samples points in a regular grid/gaussian pattern.
  - **Random sampling** sample random points in the pixel, as our eyes are less sensitive to noise than artifact.
  - **Poisson disc sampling** reject rays less than distance ε from each other, but it takes n^2 comparisons to check for disc overlaps.
  - **Jitter sampling** is a type of stratified sampling: take a random grid, then random sample in each grid.
- **Adaptive super sampling** samples the four corners:
  - If the variance is high, do a super sample.
  - Otherwise, don't sample the pixel.

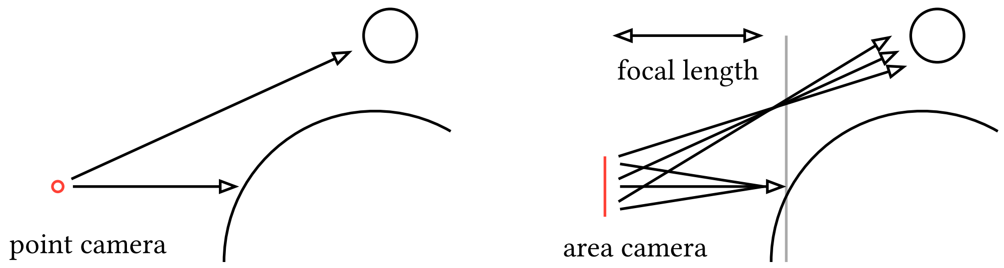
Distributed Sampling

- **Super sampling** - distributing samples in a pixel $\Longrightarrow$ **antialiasing**.
- **Area light** - distributing light on a plane $\Longrightarrow$ **soft shadows**.





- **Camera as an area** - distribute the camera position $\Longrightarrow$ **depth of field** effects.



And distributing sampling through time creates **motion blur**.

Rasterisation

Ray tracing gives very high quality results, but is computationally expensive.

Real-time applications use rasterisation:

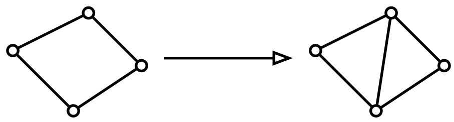
1. Model surfaces as polyhedrons.
2. Apply transformations to project the plane on screen.
3. Fill pixels with colours of the nearest visible polygon.

Most modern games use 90% rasterisation combined with 10% ray tracing.

Definition
Polyhedral surfaces are made of connected polygons surfaces.

We can approximate curved surfaces with polygons - the triangle is the simplest polygon as it vertices must be planar. GPUs are optimised to draw triangles.

We can split a polygon surface to triangles.



- **Postscript**: 2D transformations.
- **OpenGL**: 3D transformations.

Transformations as Matrices

Transformation	Matrix
Scale by factor m	$\begin{pmatrix} m & 0 \\ 0 & m \end{pmatrix}$
Rotate by angle θ	$\begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$
Shear parallel to x by factor a	$\begin{pmatrix} 1 & a \\ 0 & 1 \end{pmatrix}$

Homogenous Coordinates

We could not represent transformation as matricies, unless we define the homogenous coordinates.

$$(homogenous) = (conventional)$$

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

There is an infinite number of homogenous coordinates that maps to the same point.

Transformation	Matrix
Scale by factor m	$\begin{pmatrix} m & 0 & 0 \\ 0 & m & 0 \\ 0 & 0 & 1 \end{pmatrix}$
Translate by (x, y)	$\begin{pmatrix} 1 & 0 & x \\ 0 & 1 & y \\ 0 & 0 & 1 \end{pmatrix}$

Multiple transformations can be concatenated to make a more efficient transformation.

Note that in general, transformations are **not commutative**.

Scale/Rotation About a Point

To scale by factor of m about point (x_0, y_0)

1. Translate by $(-x_0, -y_0)$
2. Scale by factor of m
3. Translate by (x_0, y_0)

Similar for rotation.

3D Homogenous Coordinates

It is a simple extension of the 2D homogenous coordinates.

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

Example: Placing a Cylinder in 3D Space

The program defines a cylinder as one with radius 1, height 2, oriented in direction of $(0, 0, 1)$, and centred at origin. We need to apply transformations to get it to the correct place.

We usually do in order

1. Scale
2. Rotate
3. Translate

So each step does not interfere with the previous steps. (what's the word for that?)

Rotation is the only nontrivial step, it is easier to do it in reverse order

1. Find the rotation R_1 about the y axis so the desired shape is oriented in direction $(0, y, z)$
2. Find the rotation R_2 about the x axis so the desired shape is oriented in direction $(0, 0, 1)$

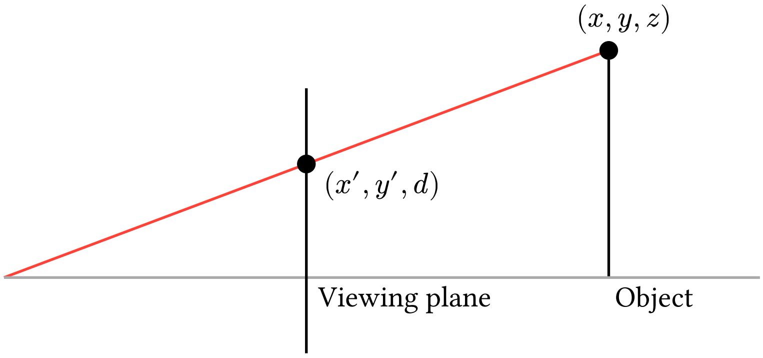
$$\text{The combined transformation} = T \times (R_1)^{-1} \times (R_2)^{-1} \times S$$

With transformations an object can be modelled once, and multiple instances can be placed.

2D Projection

Parallel projection	Perspective projection
Used in CAD.	Thing's get smaller as they get further away.
Less realistic.	More realistic.

Project to Viewing Plane



By similar triangles.

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

The furthere things are the smaller they look, because it is divided by larger z.

We also want $z' = 1/z$ to use in the z-buffer algorithm.

$$\begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1/d \\ 0 & 0 & 1/d & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ 1/d \\ z/d \end{pmatrix}$$

Which gives conventional coordinates $(x \cdot d/z, y \cdot d/z, 1/z)$.

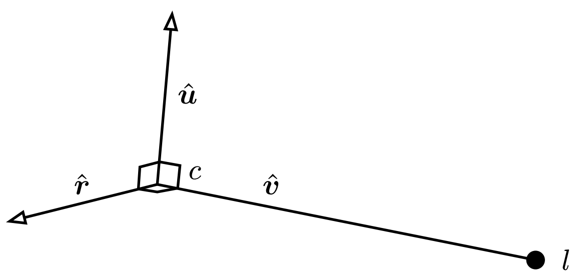
Viewing Coordinates

Instead of projecting all objects to an arbitrary plane, it is easier to transform all aobjects to a viewing coordinates system.

> **Note**
> OpenGL uses **right-handed coordinates**, which we will be using. (also note y is up)

We want to place the camera
- Centred at c
- Directed at l
- Up in direction of $\boldsymbol{u}$

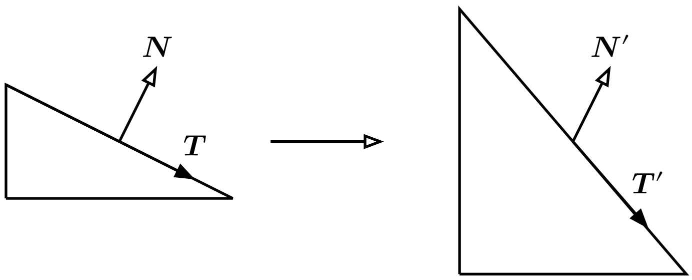
$$\hat{\boldsymbol{v}} = \frac{l - c}{|l - c|}$$
$$\hat{\boldsymbol{r}} = \frac{\boldsymbol{u} \times \hat{\boldsymbol{v}}}{|\boldsymbol{u} \times \hat{\boldsymbol{v}}|}$$
$$\hat{\boldsymbol{u}} = \hat{\boldsymbol{v}} \times \hat{\boldsymbol{r}}$$



We need to use this formula as $\boldsymbol{u}$ is given by the user, we cannot be sure that $\boldsymbol{u} \perp \hat{\boldsymbol{v}}$.

Transformaing Normal Vectors

Transformation by non-orthogonal matrix does not preserve angle, this breaks normals.



We want $\boldsymbol{N} \cdot \boldsymbol{T} = 0$ after transformation M. Then we need to transform $\boldsymbol{N}$ by matrix G.

$$T' = MT$$
$$N' = GN$$

For two vectors A and B, $A \cdot B = A^T B$ in matrix multiplication.

Note
Let A and B be matrices, and M^T the transpose operator.

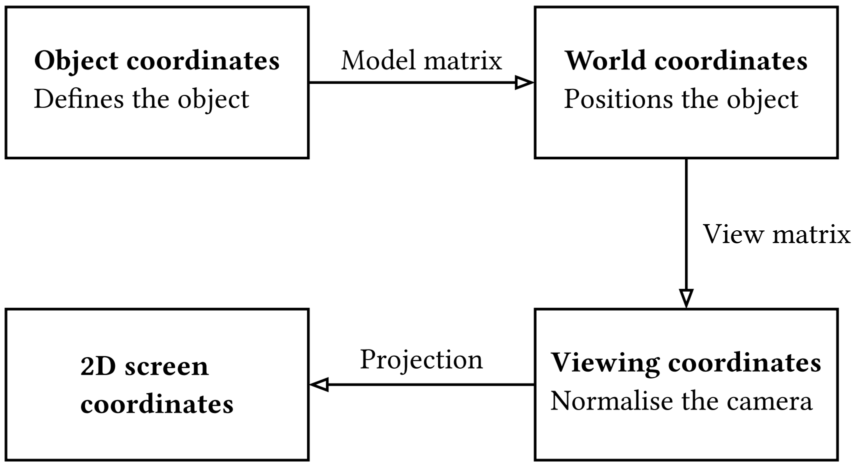
$$(AB)^T = B^T A^T$$

(Why?)

$$(GN) \cdot (MT) = (GN)^T (MT)$$
$$N^T G^T MT = 0$$

Because $N^T T = 0$, then $G^T M = I$, and $G = \left(M^{-1}\right)^T$.

The overall process to display an object on screen.



Scene Graph
To attach object B to object A.
1. Apply scale to A.
2. Apply scale, rotation and translation to move B to where it will attach to A.
3. Apply rotation and translation to both A and B.

To attach object C to B, add extra steps between 1 and 2 to attach it to B, all other transformations applied to B should also be applied to C.

We can build a **scene graph** by attachments, traversing the scene graph draws the scene.

Rasterisation Algorithm

The algorithm goes as:

1. Set the model, view, projection (MVP) transforms.
2. For all triangles, transform the vertices with MVP.
3. If the triangle is in **view frustum**, clip triangle to screen boarder.
4. For each **fragment** (pixel in the triangle), interpolate the attributes of the fragment between pixels (e.g. colour and normal).
5. If the fragment is closer to the camera than pixels drawn so far, update the screen pixels with fragment colour.

Interpolation

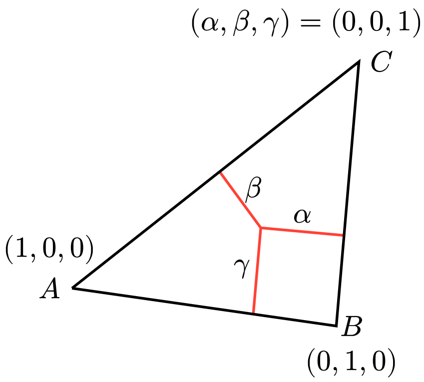
Homogenous barycentric coordinates are used for interpolating attributes.

Where α, β and γ are the distances of point P from the line $\overline{BC}$, $\overline{AC}$ and $\overline{AB}$.

We want this distance to be normalised between 0 and 1, so

$$\alpha = \frac{(P - A) \cdot \overrightarrow{AB}}{(C - A) \cdot \overrightarrow{AB}}$$

Similar for β and γ. If α, β and γ are all between 0 and 1, then the point is in the triangle.



Note
We don't need to write this algorithm because it is already included in the GPU.

Occlusion

The **z-buffer** algorithm initialises a **colour buffer** and a **depth buffer** (which describes how far objects are away from the camera).

1. Initialise the colour buffer to background colour, the depth buffer to as far as possible.
2. For every fragment in every triangle, calculate z for current fragment (pixel in triangle).
3. If $z < \text{depth}(x, y)$ and $z > z_{\text{min}}$, then set $\text{depth}(x, y) = z$ and $\text{colour}(x, y) = \text{fragment colour}$.

The z-buffer stores distances with finite precision, we store $1/z$ instead of z so infinite distance can be easily represented.

Note
Z-fighting happens when two planes have the same depth.

The GPU

Definition
The **GPU** is optimised for floating point operations on large arrays of data.

It is optimised for **parallel operations** such as

- Ray tracing
- Rasterisation
- Shading
- Video encoding

GPUs have thousands of **SIMD processors**, and a much higher memory access speed than the CPU.

GPU APIs

OpenGL	Vulkan
An older API.	Has lower overhead than OpenGL.
A good starting point for doing grapahics.	Reduces CPU load.
	Allows for finer controls.
	But very verbose, it is used in performance critical code like game engines.

There are **general purpose GPU** APIs, as OpenGL and Vulkan are only for graphics.
- CUDA is only supported by Nvidia.
- OpenCL is cross platform.

Then there are graphics API that run on the browser:
- WebGL is a wrapper for OpenGL.
- WebGPU is a wrapper for Vulkan.

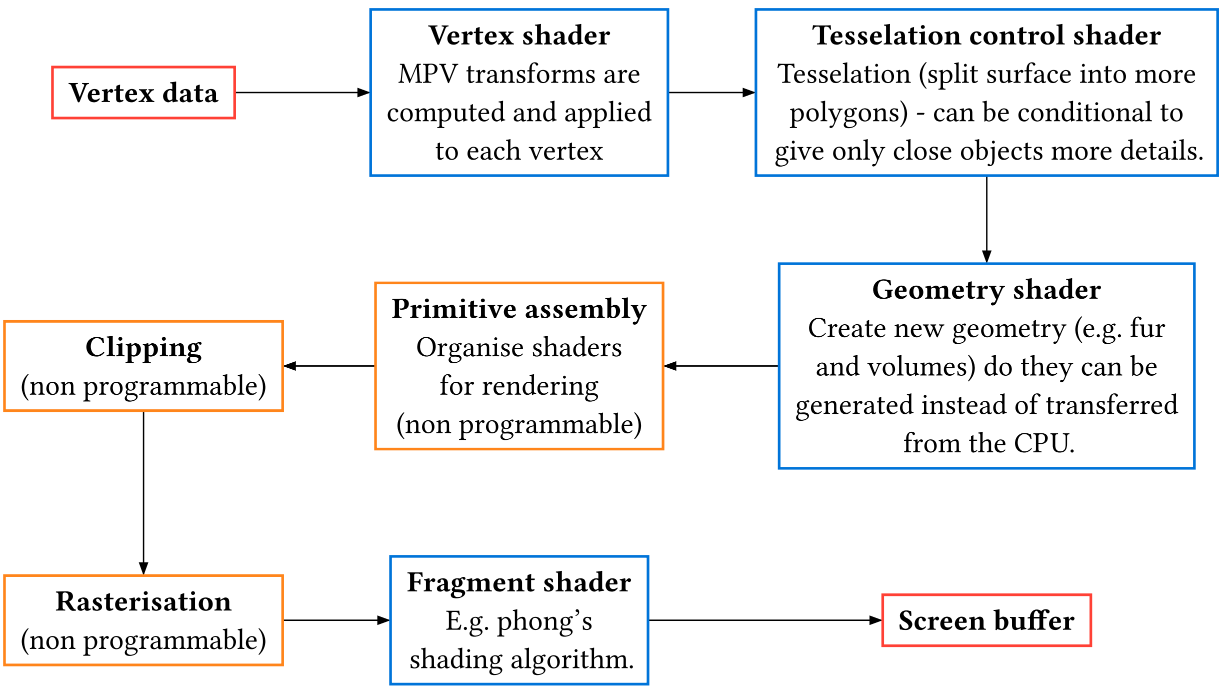
OpenGL Programming

We need to write both CPU and GPU code:
1. GL functions create OpenGL objects.
2. Data is copied between CPU and GPU

> **Definition**
> GPU code are called **shaders**, and are written in **GLSL**.

OpenGL Rendering

The rendering pipeline of OpenGL:



Vertex Data

Vertex are defined as defined by their positions and normals, primitives are defined by vertices.

Vertex attributes.

Index	Position	Normal
1	0,0,0	0,0,-1
2	1,0,0	0,0,-1
3	0,1,0	0,0,-1

Primitives (triangles)

Indicies
1,2,3

> **Note**
> Two primitives with different normals cannot share vertices.

> **Note**
> Primitives are **one-sided** by default, it is see through if the normal is in the wrong direciton.

GLSL

Shaders are code that runs on the GPU, it is executed for each fragment and vertex.

```
#version 330 // OpenGL driver version
in vec3 position;
in vec3 normal;
out vec3 frag_normal;
uniform mat4 mpv_matrix;

void main() {
  frag_normal = normal;
  gl_position = mpv_matrix * vec4(position, 1.0); // position is vec3,
                                                  // MPV is 4x4
}
```

Aggregate Types

The aggregated types have format `([empty for float]|d|i|u|b)vec(2|3|4)` corresponding to `float`, `double`, `int`, `uint` and `bool` vectors.

> **Note**
> Operating on **aggregate types** such as `vec4` is almost as fast as opering to a `float` due to parallelisation.

Arrays exist:

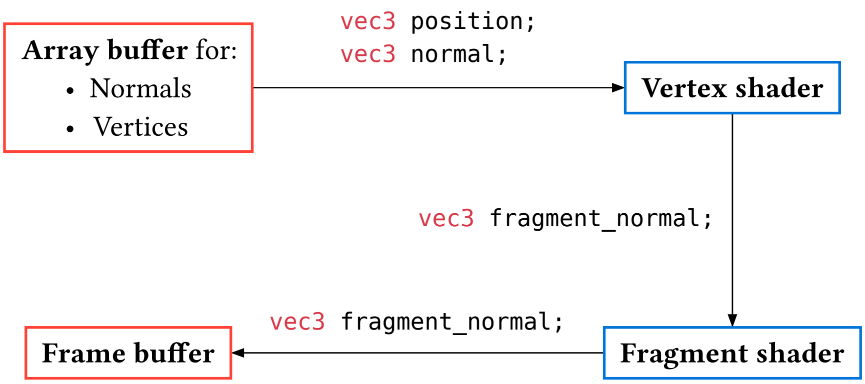
```
float lut[5] = float[5](1.0, 2.0, 3.0, 4.0, 5.0);

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

Storage Qualifiers

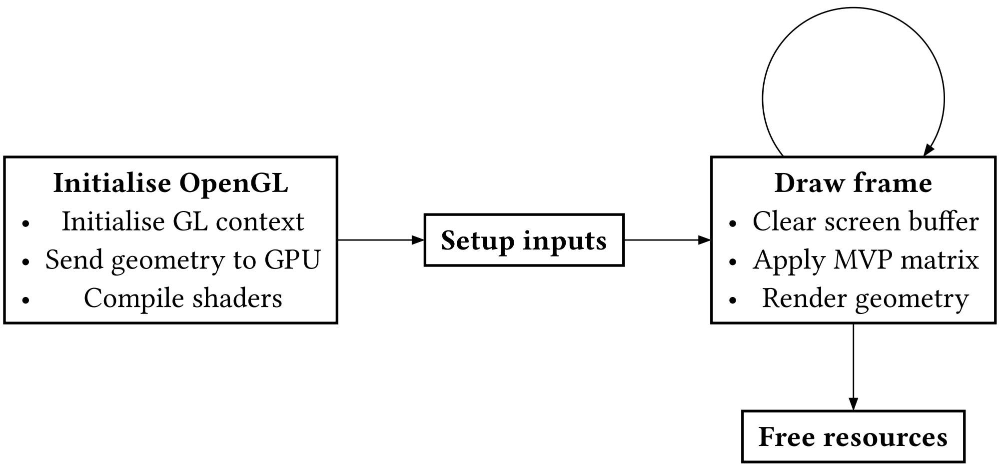
Modifier	Description
`const`	Fixed at compile time.
`in`	Input to the shader.

Modifier	Description
`out`	Output of the shader.
`uniform`	Parameter passed from application (Java), constant for geometry.
`buffer`	GPU memory buffer with R/W access.
`shared`	???



Note
Recursion is not allowed in GLSL.

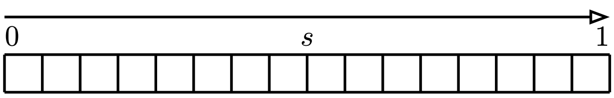
Flow of GL Applicaiton



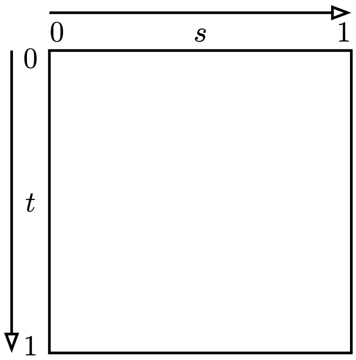
Textures

Textures is a way to get details without increasing geometry. Each point on the texture have coordinates normalised between 0 and 1.

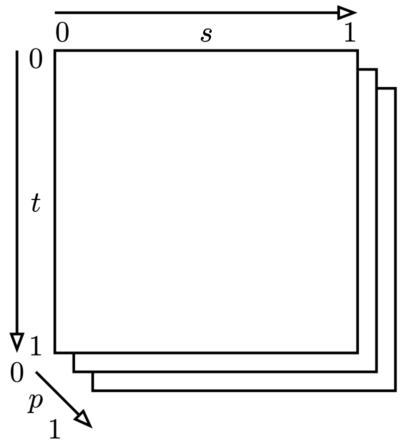
1D Textures (s)



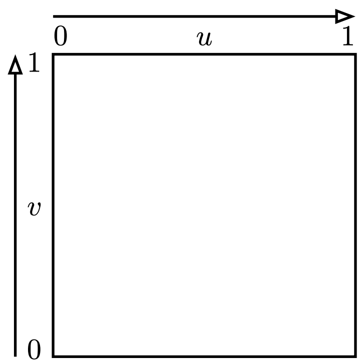
2D Textures (s, t)



3D Textures (s, t, p)



OpenGL uses a UV-map, a (u, v) coordinate is defined for each vertex so by interpolation, every surface point gets a texture value.



Upscaling and Downscaling

Upscaling algorithms:

- **Nearest neighbour** creates blocky artifacts.
- **Bilinear interpolation** creates blurry artifacts.

Downsampling algorithm:

- **Area avreaging** averages the **texels** (texture elements) the pixel covers. This is a slow operation.
- **Mipmap** stores texture in multiple resolution so that calculation don't need to be done every time.

  A full scale texture image, half scaled, quarter scaled, etc., images are stored. This only requires $1/3$ extra storage.

Texture Tiling

We want to store small fragments of the texture, and allow it to wrap around in coordinates.

$$T(2.1, 0) = T(1.1, 0) = T(0.1, 0)$$

> **Note**
> We can store all UV maps of a surface in a single image so it is easier to manage.

Other Mappings

Bump mapping	Use a texture to deflect hte normal, which affect the shading.
Displacement mapping	Also changes the shape (outline) of an object.
Environment mapping	Environment texture have infinite distance to the source of reflection.

Note
A **cube map** is used for **sky boxes**, each face captures the environment in that direction.

A texture can contain attributes, e.g. how to interpolate between mipmap levels.

OpenGL Buffers

Render buffers are where we render screen pixels.

Buffer type	Buffer names
Colour	`gl_front`, `gl_back`
Stereo (for different image in each eye)	`gl_front_left`, `gl_back_left`, `gl_front_right`, `gl_back_right`
Depth	The z buffer
Stencil	Block rendering for specific pixels

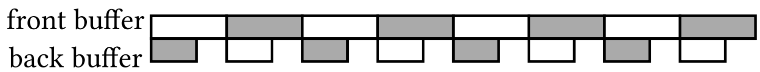
Note
To simulate **perfect specular reflection**:
1. Render the world without the reflection.
2. Move the camera to the surface, then put what it sees on the reflective surface.

Front and Back Buffers

- The **back buffer** is the one that the GPU draws to.
- The **front buffer** is displayed to the screen.

Note
The GPU can draw to the front buffer, it clear the buffer before drawing to it, this cause flickers.

When we are done drawing, call the **swap** command to swap the buffers.



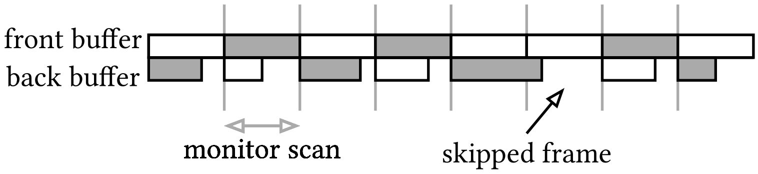
Notice there are gaps between times we can draw to the back buffer, this is time wasted. We can remove them by using a 3rd buffer.

V-Sync

Since pixels are copied to the screen row by row, if **swap** is called while this is still happening, **tearing artifacts** can occur.

Definition
Tearing artifacts is where upper screen shows the current frame but the lower screen shows the old frame.

To remove this, the GPU have to wait for the current frame (buffer) to finish copying to the screen before calling swap.



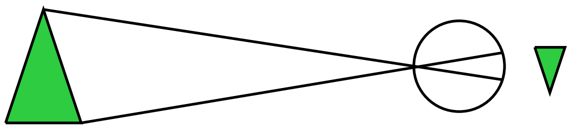
Variable Refersh Rate

VRR allows the GPU to control the timing of the frames, this allows.

- Saving power when the screen is static.
- Reduce lag for real time graphics.

Human Vision

The lens of our eyes acts like a camera to project an ubside down image onto the retina.



Definitions

Keyword	Definition
Retina	The inner surface of eyeball with the photoreceptors.
Forea	Where our eye gives the highest resolution.
Cornea & lens	Focus light onto the retina.
Pupil	Shrink and expands to control the amount of light entering the eye.

Colour Representation

Phong's model only has 3 channels

- But we need the absorption spectra of a surface for a fully accuracte reflection.
- This is expensive and requires hundreds of channels.

Although it is true that

$$L(\lambda) = I(\lambda)R(\lambda)$$
$$\text{reflected light} = \text{illumination reflectance}$$

There are 3 types of cone cells: S, M, L responsible for colour vision.

For a particular light, the cone response for S is

$$R_s = \int_{380}^{730} I(\lambda)R(\lambda)d\lambda$$

where the visible light has a wavelength range of $380 - 730$.

The percepted colour is entirely characterised by R_s, R_m and R_l. It is possible for two different light specturms to appear the same if

$$\begin{aligned} R_s &= R_{s'} \\ R_m &= R_{m'} \\ R_l &= R_{l'} \end{aligned}$$

CIE-XYZ Colour Space

Any colour can be matched using three linear independent reference colour.

CIE-XYZ is a standard colour space:
- R, G, B values can be expressed as a funciton of the spectrum.
- Does not require negative contribution (some colour space requires negative red light to accomodate for all wavelengths).

CIE Chromaticity Diagram

> **Note**
> Search the image up yourself.

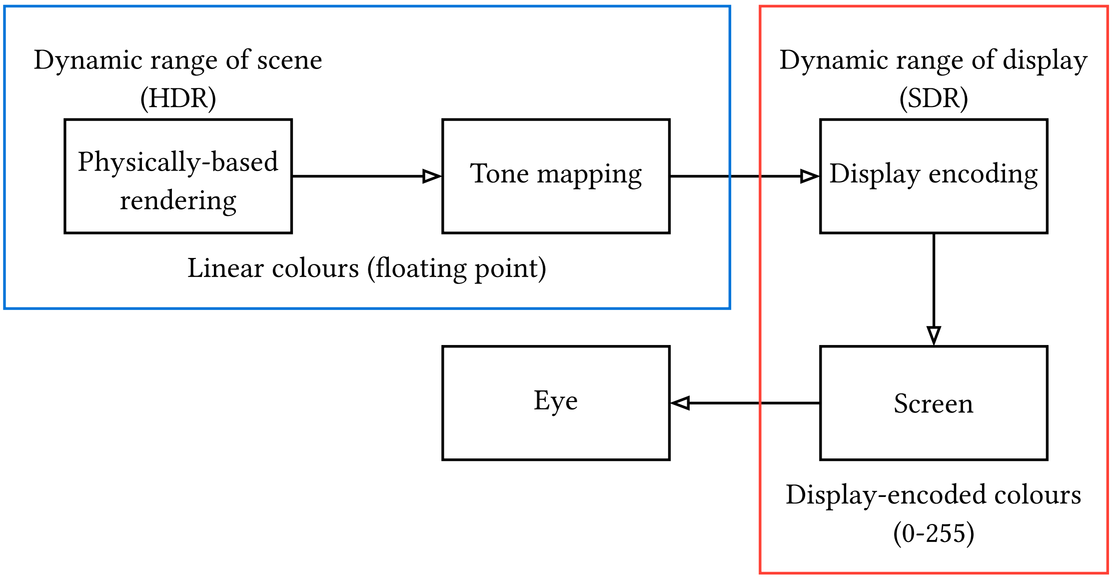
The diagram ignores luminosity, it shows:
- Pure colours on the edge of the diagram.
- All colours are mixture of pure colours.
- No colours are possible outside of the diagram.

Luminosity is the ability to distinguish black from white (green + red). All possible colours forms a 3D vaolume in an XYZ space.

Dynamic Range

A image with standard dynamic range encodes only colours the screen can display.

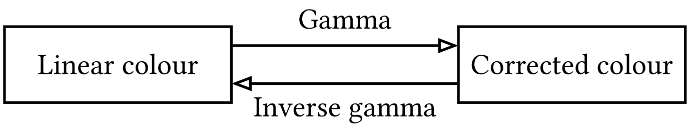
Since we now have different kinds of screens (OLED, laser, etc.), HDR tries to encode all visible wavelengths, so the screens can display the image more accurately with the encoded information.



Gamma Correction

Gamma correction is used to encode **luminance** and **tri-stimulus colour**, so the pixels give a scale of brightness level that is *perceptionally uniform*. So only 8 bits is needed to remove banding artifacts (instead of 12).

$$V_{\text{out}} = a \cdot (V_{\text{in}})^{\gamma}$$

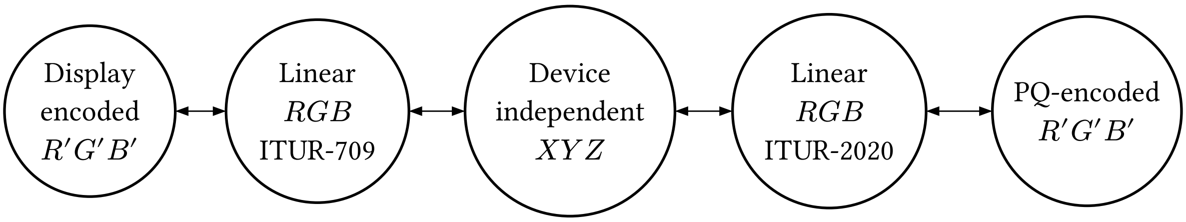


Luma is the gamma corrected greyscale brightness. Let $R'G'B'$ be the gamma corrected RGB.

$$\text{luma} = 0.21R' + 0.71G' + 0.07B' \text{ by experimentation}$$

SDR to HDR Conversion

HDR can represent wider colours than SDR as each pixel has more bit.



$$\begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = M_{\text{RGB to XYZ}} \cdot \begin{pmatrix} R \\ G \\ B \end{pmatrix}_{\text{R709}}$$

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix}_{\text{R2020}} = M_{\text{XYZ to R2020}} \cdot M_{\text{R709 to XYZ}} \cdot \begin{pmatrix} R \\ G \\ B \end{pmatrix}_{\text{R709}}$$

- Let L be an $n \times 1$ matrix represent the sample intensities of n different wavelengths.
- Let S_{XYZ} be a matrix that transforms L to XYZ colour space.

$$\begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = (S_{\text{XYZ}})^T \cdot L$$

Let P_{RGB} be an $n \times 3$ matrix that transforms RGB colours to L. $M_{\text{RGB to XYZ}}$ is a 3×3 matrix.

$$M_{\text{RGB to XYZ}} = (S_{\text{XYZ}})^T \cdot P_{\text{RGB}}$$

Alternative Colour Spaces

Colour space	Description
RGB	Represents colour in the fewest number of bits, can be linear or display encoded, scene referred or display encoded.
CMY	Inverse of RGB, used for printing because ink absorbs light.
CMYK	CMY with black ink, because CMY gives dark grey instead of black, and printing black text using CMY causes **colour fringement** due to misalighment.
HSV	Hue, saturation and value (brightness). All pure colours and white have value 1.
HLS	Similar to HSV, all pure colours have value 0.5, white have value 1 (because white is brighter than pure colours).

The CIE u'v' Colour Space

Grouping indistinguishable colours in the CIE-XYZ colours space shows some parts of the colour space are less distinguishable than others: the groups are larger where they are less distinguishable.

The CIE u'v' colour space transforms CIE-XYZ so each groups have equal size.

Tone Mapping

The human eye can adapt to different light levels, tone mapping is used for:

- Reduce dynamic range.

- Customise the look.
- Simulate human vision.
- Adapt image to conditions.
- Make renders more realistic.
- Map scene to display encoded colours.

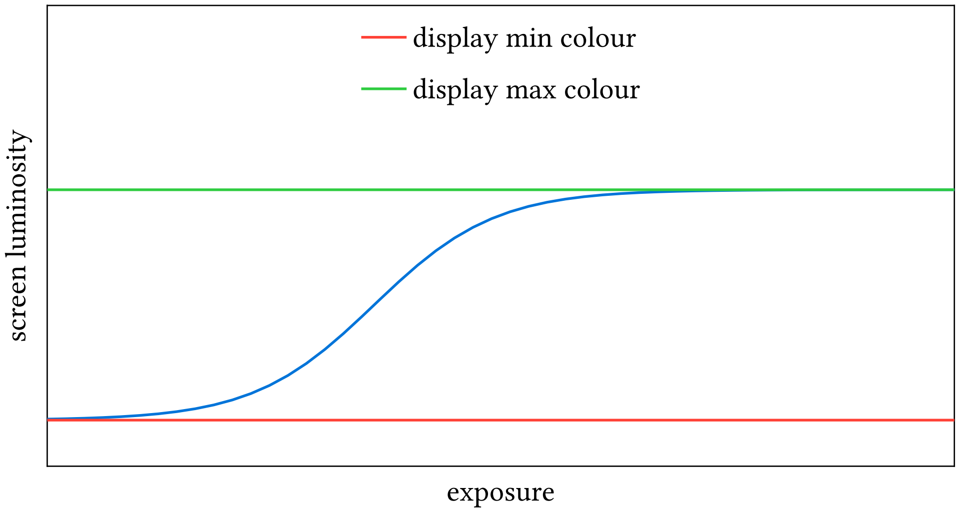
Exposure changes scene white S_{white}

$$R_d = \frac{R_s}{S_{\text{white}}}$$

Where R_s and R_d are scene and display encoded colour. (assume $R_d = 1$ is the most white the display can show)

Tone Curve

This is the **sigmoidal tone curve**. The goal of the tone curve is to be steep where it is the most visible: colours with very high and very low luminance is discarded as they are less distinguishable.



This mimics the analogue film, and is fast to compute.

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

- L_m sets the medium colour of the image.
- a shifts the curve to left or right.
- b is the steepness of the curve.

Contrast can be increased by increasing b, but lowers dynamic range.

`END Introduction to Graphics`