CO317 - Graphics   (60005)

Lecture 1 - Projections and Transformations

Two Dimensional Graphics

At the lowest level, in every operating system, graphics processing operates on the pixels in a window with primitives, such as;

- `SetPixel(int x, int y, int colour);`
- `DrawLine(int xs, int ys, int xf, int yf);`

However, we'd like to be able to draw scenes from a three-dimensional world and have it appear in two-dimensional graphics primitives.

World Coordinate System

In order to achieve independence when drawing objects, we define a world coordinate system. For example, let our world be defined in meters, we can then allow a pixel to represent a millimetre. A viewing area is a window, and is defined as part of our 3D world. **Clipping** occurs when we attempt to draw outside (dashed) of the window;

```
1  SetWindow(30, 10, 70, 50)
2  DrawLine(40, 3, 90, 30)
3  DrawLine(50, 60, 60, 40)
```
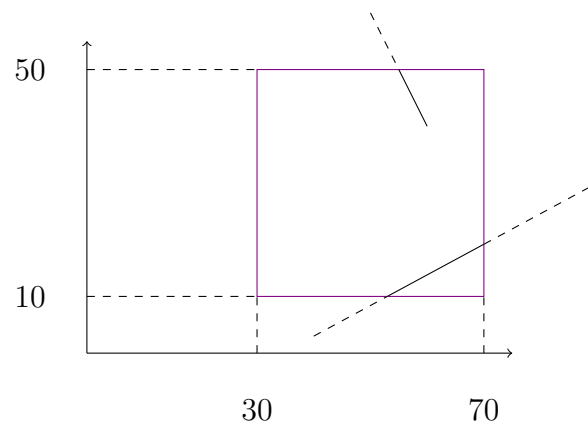

However, this isn't as trivial to do in 3D, as it cannot simply be left to the operating system. While we can represent 3D objects as a series of 2D commands, it's inefficient and expensive for the OS to perform the clipping (therefore we should do this manually).

Normalisation

A normalisation process is required to convert from device independent commands (where screen resolution isn't taken into account) to drawing commands using pixels. Consider a point in the world coordinate window (X_w, Y_w), and its corresponding result on the viewport (pixel coordinates; (X_v, Y_v));
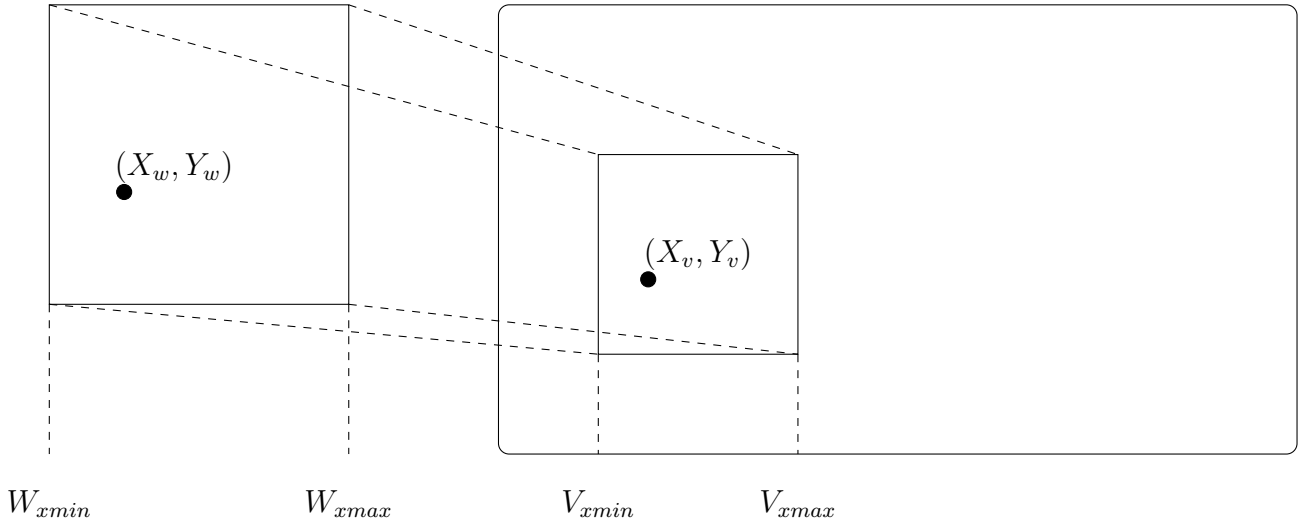
The expressions are similar for Y;

$$\frac{(X_w - W_{xmin})}{(W_{xmax} - W_{xmin})} = \frac{(X_v - V_{xmin})}{(V_{xmax} - V_{xmin})} \Rightarrow X_v = \frac{(X_w - W_{xmin})(V_{xmax} - V_{xmin})}{(W_{xmax} - W_{xmin})} + V_{xmin}$$

This gives us the resulting pair of linear equations (intuitively), where the constants found from the known values W_{xmin}, V_{xmax}, etc. are used to define the normalisation;

$$X_v = AX_w + B$$
$$Y_v = CY_w + D$$

Polygon Rendering

Most graphics applications deal with very simple objects - flat / planar polyhedra, referred to as **faces** or **facets**. These are graphic primitives, and can be used to approximate any shape. Consider the following tetrahedron, consisting of four vertices;
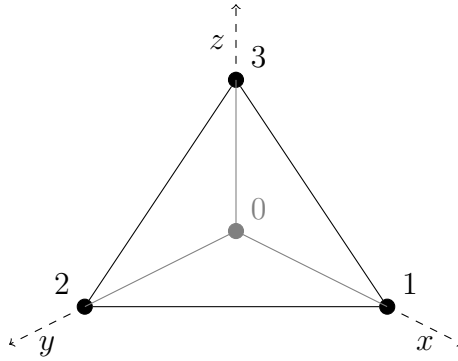
For this, we need a mixture of different data, including numerical data about the actual 3D coordinates of the vertices, as well as topological data regarding what vertices are connected to what. This can be represented in the following tables;

vertex data			face data	
index	location		index	vertices
0	$(0,0,0)$		0	0 1 3
1	$(1,0,0)$		1	0 2 1
2	$(0,1,0)$		2	0 3 2
3	$(0,0,1)$		3	1 2 3

This separation allows for the vertices to move without affecting the faces.
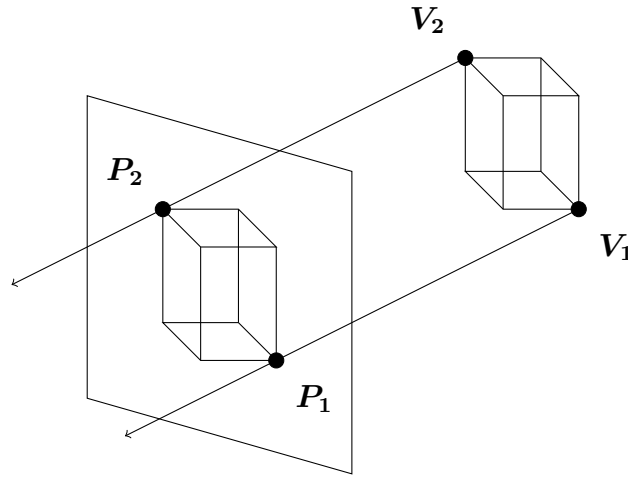
Projections

In order to draw a 3D wire frame, the points must first be converted into a 2D representation, via a **projection**, which can then be drawn with simple drawing primitives. Intuitively, we have an observer (a focal point) where all viewing rays converge. The observer is located between a projection surface P and an object V. While it's possible to project onto any surface, we only consider linear projections onto a flat surface.

Orthographic Projections

The simplest form of a projection is an **orthographic projection**. The assumptions made are that the viewpoint is located at $z = -\infty$, and the plane of projection is $z = 0$. With the viewing point being infinitely far away, the rays become parallel. This gives all projectors the same direction;

$$\boldsymbol{d} = \begin{bmatrix} 0 \\ 0 \\ -1 \end{bmatrix}$$

This gives the following, with each projection line having the equation $\boldsymbol{P} = \boldsymbol{V} + \mu\boldsymbol{d}$;

By substituting in the direction $\boldsymbol{d}$ we have determined, it gives the following Cartesian equations for each component;

$P_x = V_x + 0$
$P_y = V_y + 0$
$P_z = V_z - \mu$

However, since we have the projection plane $z = 0$, we also know that $P_z = 0$, therefore we don't need to solve for μ. From this, we can determine the projected location is the 3D x and y components of the vertex;
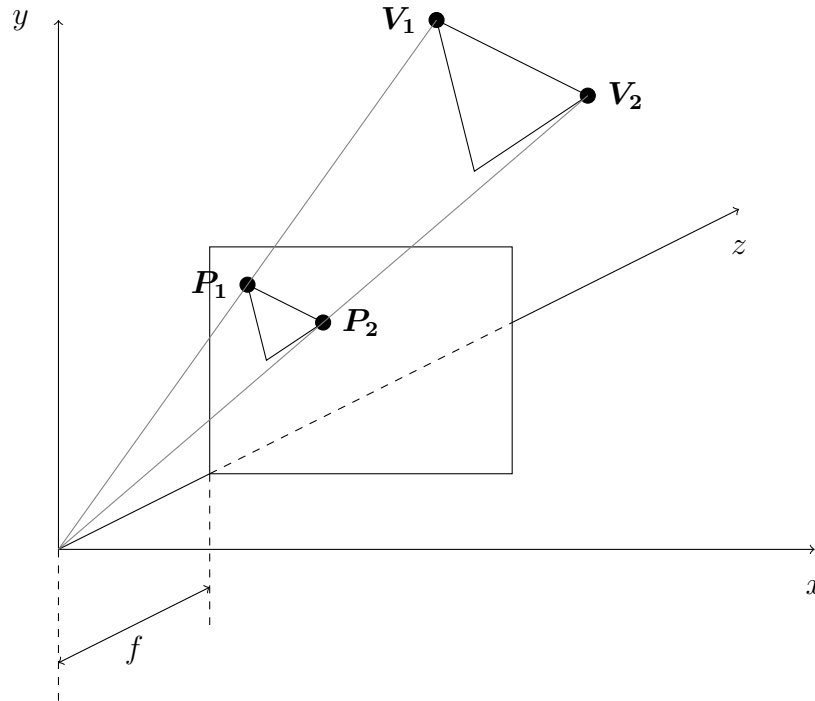
$$\boldsymbol{P} = \begin{bmatrix} V_x \\ V_y \\ 0 \end{bmatrix}$$

Viewing the wireframe for a cube directly from a face would look like the following;

Perspective Projection

While orthographic projections are fine when depth isn't a consideration (such as objects mostly being at the same distance from the viewer), it's insufficient for close work, where we want details to be realistic. The difference here is that we are no longer at an infinite distance (instead being at the origin), and the projection plane is $z = f$ (where f stands for focal length);
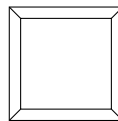

This gives us the following equation (since all projectors must go through the origin);

$$\boldsymbol{P} = \mu \boldsymbol{V}$$

We can work out the value of μ, let it be μ_p as follows;

$$
\begin{aligned}
P_z &= f && \text{by projection plane} \\
\mu_p &= \frac{P_z}{V_z} \\
&= \frac{f}{V_z} \\
P_x &= \mu_p V_x \\
&= \frac{f V_x}{V_z} \\
P_y &= \mu_p V_y \\
&= \frac{f V_y}{V_z}
\end{aligned}
$$

Viewing the wireframe for a cube directly from a face would look like the following (note the difference to the orthographic projection);

Transformations

Scenes are defined in a particular coordinate system, but we want to be able to draw a scene from any angle. To do so, it's easier to have the viewpoint at the origin, and the z-axis as the direction of view. As such, we need to be able to **transform** the coordinates of a scene.
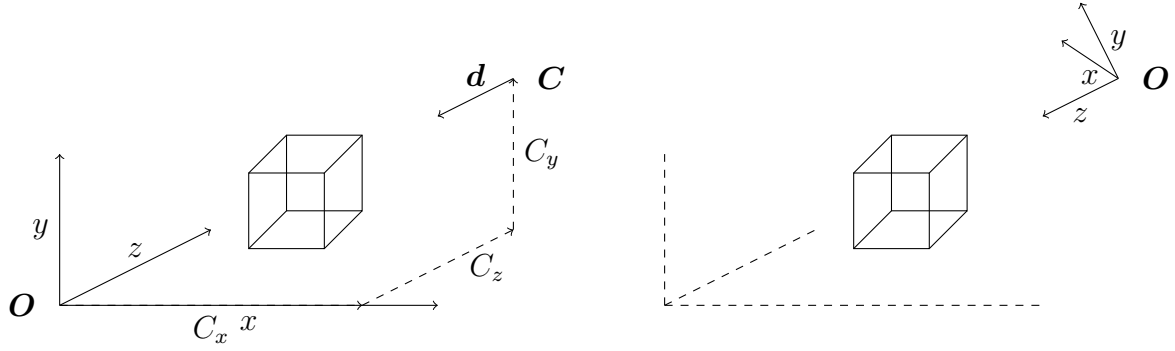
These are done by the application of transformation matrices. For example, a standard transformation to make an object twice as big from the origin;

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

Translation

However, being restricted to matrix operations with $\mathbb{R}^{3 \times 3}$ means that we cannot represent translations (for example, a shift of two units on the x-axis, such that $x' = x + 2$). The solution to this is to use 4D **homogenous coordinates**, where we assume the fourth dimension is fixed to 1.

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Frequently the last ordinate is 1, however in general it is a scale factor;

$$\underbrace{(p_x, p_y, p_z, s)}_{\text{homogenous}} \Leftrightarrow \underbrace{\left(\frac{p_x}{s}, \frac{p_y}{s}, \frac{p_z}{s} \right)}_{\text{Cartesian}}$$

Affine Transformations

Affine transformations preserve parallel lines. Most of the transformations we require are affine, with the most important being scaling, rotation, and translation;

- **scaling**
<div align="right">by (s_x, s_y, s_z)</div>

$$\begin{bmatrix} s_x & 0 & 0 & 1 \\ 0 & s_y & 0 & 1 \\ 0 & 0 & s_z & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix} = \begin{bmatrix} s_x p_x \\ s_y p_y \\ s_z p_z \\ 1 \end{bmatrix}$$

- **rotation**

   In order to define a rotation, we need both an axis and an angle, with the simplest rotations being about the Cartesian axes. The following matrices are used for rotations of θ about each of the axes;

$$\mathcal{R}_{\boldsymbol{x}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta & 0 \\ 0 & \sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

5

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

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

It's important to note that **rotations have a direction**. In this course, we use a left-handed coordinate system, where the rotation is anti-clockwise when looking along the axis of rotation (think about the origin being closer to you, and the axis going off to ∞ away from you).

- **translation** by (t_x, t_y, t_z)

$$\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix} = \begin{bmatrix} p_x + t_x \\ p_y + t_y \\ p_z + t_z \\ 1 \end{bmatrix}$$

However, perspective projections are an example of a non-affine transformation, as it doesn't preserve parallels. Intuitively, it's not invertible (singular), as we cannot convert from a photograph to a 3D model.

Note that we should be careful when we combine transformation. As matrix multiplication isn't commutative, we should read a sequence of matrices multiplied together from right to left, with the right-most matrix, before the vector, being the **first** transformation to be applied.

Lecture 2 - Transformations for Animation

Recall the transformation in the previous lecture, which took way too long to draw, moving the origin to the view point. It consists of three steps, with the latter two being used to align the z-axis with the view direction;

1. translation of the origin fairly trivial to do

$$\mathcal{A} = \begin{bmatrix} 1 & 0 & 0 & -C_x \\ 0 & 1 & 0 & -C_y \\ 0 & 0 & 1 & -C_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

2. rotation about y-axis

    Consider the following, looking at the $x - z$ plane;

This can be used to calculate the following (note that here we are using a right-handed system);

$$||\boldsymbol{v}|| = v$$
$$= \sqrt{d_x^2 + d_z^2}$$
$$\cos\theta = \frac{d_z}{v}$$
$$\sin\theta = \frac{d_x}{v}$$
$$\boldsymbol{B} = \begin{bmatrix} \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$
$$= \begin{bmatrix} \frac{d_z}{v} & 0 & -\frac{d_x}{v} & 0 \\ 0 & 1 & 0 & 0 \\ \frac{d_x}{v} & 0 & \frac{d_z}{v} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

3. rotation about x-axis

   This follows a similar process, note that we are now aligned along the $y - z$ plane (for clarity, the horizontal distance is v);

   Similarly, the matrix can be obtained as follows;

$$\cos\phi = \frac{v}{|\boldsymbol{d}|}$$
$$\sin\phi = \frac{d_y}{|\boldsymbol{d}|}$$
$$\boldsymbol{C} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi & 0 \\ 0 & \sin\phi & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$
$$= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{v}{|\boldsymbol{d}|} & -\frac{d_y}{|\boldsymbol{d}|} & 0 \\ 0 & \frac{d_y}{|\boldsymbol{d}|} & \frac{v}{|\boldsymbol{d}|} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

From this, we are able to combine the matrices into the following;

$$\mathcal{T} = \mathcal{CBA}$$

For every point $\boldsymbol{P}$ in the scene, we can obtain $\boldsymbol{P_t} = \mathcal{T}\boldsymbol{P}$, with the view in **canonical** form, allowing us to apply the standard perspective or orthographic projection.

Rotation About a General Line

Rotation of a scene around a general line can be done as a combination of transformations. The idea is similar, with the following three steps;

1. making the line of rotation one of the Cartesian axes

   This uses the matrices derived before, rotating the general line to be aligned with the z-axis.

2. perform the rotation

   Standard rotation around the z-axis defined previously.

3. restore line to original place

   Inversion of the initial matrices to revert rotation.

This gives us the following full matrix;

$$\mathcal{T} = \underbrace{\mathcal{A}^{-1}\mathcal{B}^{-1}\mathcal{C}^{-1}}_{3} \underbrace{\mathcal{R}_z}_{2} \underbrace{\mathcal{C}\mathcal{B}\mathcal{A}}_{1}$$

Projection Matrices

For the canonical / orthographic projection, the matrix simply drops the z component;

$$\mathcal{M}_O = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathcal{M}_O \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ 0 \\ 1 \end{bmatrix}$$

This is clearly non-invertible, as we are losing information about one of the axes. An effect of this is that we must do any effects in 3D **before** applying the projection matrix.

The perspective projection matrix can also be done in a similar way;

$$\mathcal{M}_p = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{1}{f} & 0 \end{bmatrix}$$

$$\mathcal{M}_p \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ \frac{z}{f} \end{bmatrix}$$

$$\rightarrow \begin{bmatrix} \frac{fx}{z} \\ \frac{fy}{z} \\ f \\ 1 \end{bmatrix} \qquad \text{using the fourth ordinate as a scale factor}$$

Note that homogenous coordinates and vectors fall into one of two types;

1. **position vectors**

These have a non-zero final ordinate $(s > 0)$ and can be normalised to Cartesian form. If two position vectors are added, we instead obtain the mid-point;

$$\begin{bmatrix} X_a \\ Y_a \\ Z_a \\ 1 \end{bmatrix} + \begin{bmatrix} X_b \\ Y_b \\ Z_b \\ 1 \end{bmatrix} = \begin{bmatrix} X_a + X_b \\ Y_a + Y_b \\ Z_a + Z_b \\ 2 \end{bmatrix} = \begin{bmatrix} \frac{X_a + X_b}{2} \\ \frac{Y_a + Y_b}{2} \\ \frac{Z_a + Z_b}{2} \\ 1 \end{bmatrix}$$

This has no real meaning in geometry, but is a useful observation.

2. **direction vectors**

These have a zero in the final ordinate, and have a direction and magnitude. If two direction vectors are added, we obtain a direction vector (following the normal addition rule);

$$\begin{bmatrix} x_i \\ y_i \\ z_i \\ 0 \end{bmatrix} + \begin{bmatrix} x_j \\ y_j \\ z_j \\ 0 \end{bmatrix} = \begin{bmatrix} x_i + x_j \\ y_i + y_j \\ z_i + z_j \\ 0 \end{bmatrix}$$

However, if a direction vector is added to a position vector, we obtain another position vector;

$$\begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} + \begin{bmatrix} x \\ y \\ z \\ 0 \end{bmatrix} = \begin{bmatrix} X + x \\ Y + y \\ Z + z \\ 1 \end{bmatrix}$$

Structure of a Transformation Matrix

Note that the bottom row of such a matrix is **always** 0 0 0 1. We can decompose the columns of a transformation matrix into three direction vectors and one position vector. The three direction vectors are the new axes and the position vector is the new origin.

$$\underbrace{\begin{bmatrix} q_x & r_x & s_x & C_x \\ q_y & r_y & s_y & C_y \\ q_z & r_z & s_z & C_z \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{\text{matrix}} \rightarrow \underbrace{\begin{bmatrix} q_x \\ q_y \\ q_z \\ 0 \end{bmatrix} \begin{bmatrix} r_x \\ r_y \\ r_z \\ 0 \end{bmatrix} \begin{bmatrix} s_x \\ s_y \\ s_z \\ 0 \end{bmatrix}}_{\text{direction vectors}} \underbrace{\begin{bmatrix} C_x \\ C_y \\ C_z \\ 1 \end{bmatrix}}_{\text{position vector}}$$

When applying this transformation to a direction vector, where the last ordinate is zero, no **translation** is applied. On the other hand, if it is a position vector, where the last ordinate is 1, all vectors will have the same displacement.

The following results can be proved by observing changes to the standard basis vectors, and the origin after applying the matrix;

- $\boldsymbol{q}$ transformed x-axis
- $\boldsymbol{r}$ transformed y-axis
- $\boldsymbol{s}$ transformed z-axis
- $\boldsymbol{C}$ transformed origin

Dot Product

We can consider the dot product as a projection;

$$\boldsymbol{P} \cdot \boldsymbol{u} = |\boldsymbol{P}||\boldsymbol{u}| \cos \theta$$

Visually, we can see the dot product as the following;

If $\boldsymbol{u}$ is along a co-ordinate axis, then $\boldsymbol{P} \cdot \boldsymbol{u}$ is the ordinate of $\boldsymbol{P}$ in the direction of $\boldsymbol{u}$.

Consider changing to the new axes $\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w}$, and origin $\boldsymbol{C}$. We can call the first co-ordinate of $\boldsymbol{P}$ in the new system $\boldsymbol{P}^t_x$;

$$
\begin{aligned}
\boldsymbol{P}^t_x &= (\boldsymbol{P} - \boldsymbol{C}) \cdot \boldsymbol{u} \\
&= \boldsymbol{P} \cdot \boldsymbol{u} - \boldsymbol{C} \cdot \boldsymbol{u}
\end{aligned}
$$

However, this can also be represented in matrix notation;

$$
\begin{bmatrix} P^t_x \\ P^t_y \\ P^t_z \\ 1 \end{bmatrix} = \begin{bmatrix} u_x & u_y & u_z & -\boldsymbol{C} \cdot \boldsymbol{u} \\ v_x & v_y & v_z & -\boldsymbol{C} \cdot \boldsymbol{v} \\ w_x & w_y & w_z & -\boldsymbol{C} \cdot \boldsymbol{w} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} P_x \\ P_y \\ P_z \\ 1 \end{bmatrix}
$$

Returning to the original problem, where we need to find a transformation matrix with a viewpoint $\boldsymbol{C}$ and direction $\boldsymbol{d}$. This can be done by first finding the vectors $\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w}$. Since $\boldsymbol{d}$ is the direction of the new axes, we can write (to get a unit vector);

$$
\boldsymbol{w} = \frac{\boldsymbol{d}}{|\boldsymbol{d}|}
$$

We also want to maintain an orthogonal basis, as well as all unit vectors. For the horizontal direction, we can write $\boldsymbol{u}$ in terms of some horizontal vector $\boldsymbol{p}$;

$$
\boldsymbol{u} = \frac{\boldsymbol{p}}{|\boldsymbol{p}|}
$$
$$
p_y = 0 \qquad\qquad\qquad \text{ensure horizontal, no vertical component}
$$

Similarly, we want some vertical vector $\boldsymbol{q}$ to write $\boldsymbol{v}$;

$$
\boldsymbol{v} = \frac{\boldsymbol{q}}{|\boldsymbol{q}|}
$$
$$
q_y = 1 \qquad\qquad\qquad \text{vertical, positive component}
$$

This gives us the following to solve;

$$
\boldsymbol{p} = \begin{bmatrix} p_x \\ 0 \\ p_z \end{bmatrix} \qquad\qquad\qquad \text{new horizontal}
$$

$$
\boldsymbol{q} = \begin{bmatrix} q_x \\ 1 \\ q_z \end{bmatrix} \qquad\qquad\qquad \text{new vertical}
$$

10

However, the view direction can also be written as the following (since we have the view direction, which happens to be the new z-axis as perpendicular to the remaining two vectors);

$$d = p \times q$$

Using this, we can write p in terms of q;

$$
\begin{aligned}
d &= \begin{bmatrix} d_x \\ d_y \\ d_z \end{bmatrix} \\
&= p \times q \\
&= \begin{vmatrix} i & j & k \\ p_x & 0 & p_z \\ q_x & 1 & q_z \end{vmatrix} \\
&= -p_z i + (p_z q_x - p_x q_z) j + p_x k \\
&= \begin{bmatrix} -p_z \\ p_z q_x - p_x q_z \\ p_x \end{bmatrix} \qquad\qquad \Rightarrow \\
d_x &= -p_z \\
d_y &= p_z q_x - p_x q_z \\
d_z &= p_x \qquad\qquad\qquad\qquad\qquad\quad \Rightarrow \\
p &= \begin{bmatrix} d_z \\ 0 \\ -d_x \end{bmatrix}
\end{aligned}
$$

However, since we know that the vectors p and q are orthogonal, we can say the following;

$$
\begin{aligned}
p \cdot q &= 0 \qquad\qquad\qquad\qquad\qquad\qquad \Rightarrow \\
p_x q_x + p_z q_z &= 0 \\
d_y &= p_z q_x - p_x q_z \qquad\qquad \text{from above}
\end{aligned}
$$

Both of these equations can be fully written in terms of d.

Lecture 3 - Clipping and 3D Geometry

Clipping is the process of eliminating portions of objects outside the **viewing frustum** (boundaries of the image plane projected in 3D, consisting of a near and far clipping plane). This is useful to avoid degeneracy (by not drawing objects behind the camera), as well as improving efficiency by not processing objects which won't be visible.

There are three points when we could clip;

- before perspective transform (3D space)   world co-ordinates
- after perspective transform   homogeneous co-ordinates
- after perspective division   screen space

The second option, in camera co-ordinates, is ideal. This is because the clipping planes are axis aligned, as we can discard anything further than the far plane or closer than the near plane.

Halfspace

We can define any infinite line (for simplicity, in 2D) as a test;

$$f(x, y) = 0 \text{ such as } x - y + 1 = 0$$
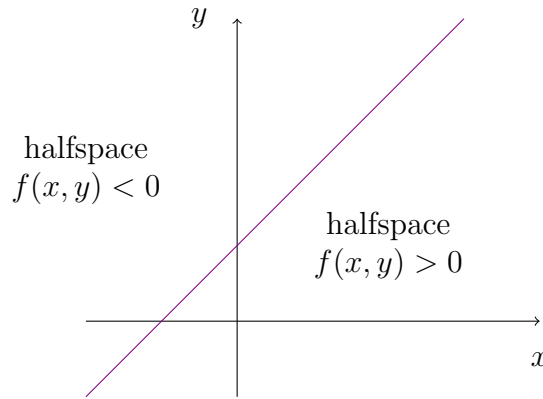
In 3D, the plane equation is $f(x, y, z) = Ax + By + Cz + D = 0$, which also divides space into two spaces, one where the test is positive, and one where it is negative.

Note that we can define $\boldsymbol{H}$ as the normal vector of our plane, and also normalise it to avoid infinite solutions by scaling;

$$\boldsymbol{H} = \begin{bmatrix} A \\ B \\ C \\ D \end{bmatrix}$$
$$A^2 + B^2 + C^2 = 1$$

The distance is quite easily calculated as follows;

$$d = \boldsymbol{H} \cdot \boldsymbol{P} = \boldsymbol{H}^\top \boldsymbol{P}$$

This is a **signed** distance, where a positive value would denote inside, and a negative value would denote outside.

Consider the **view frustum**, where we have 6 planes, and their normals oriented towards the interior of the frustum (where each plane has its own $\boldsymbol{H}$). If $\boldsymbol{H} \cdot \boldsymbol{P} < 0$ for **any** of the planes, then it is clipped, as it would be outside of the frustum.

Lecture 4 - Graphics Pipelines and APIs

All graphics systems work according to some fundamental principles, we can define two conceptually related graphics pipelines;

- **declarative**   tell system what we want to render, not how

  Relates to OOP. For example, we can define a sphere which knows about its environment, and its location etc, and can draw itself with graphics primitives (essentially how to tessellate itself). Virtual cameras, and scene descriptions (scene graphs) can also exist. Every object may know about each other.

- **imperative**   tell the system how to render something, but not what is being rendered

  In contrast, the pipeline here now takes in a sequence of drawing commands, such as drawing a vertex at a given position. As such, objects can be drawn independent from each other, allowing for a degree of parallelism.

It's important to note that we build declarative pipelines on top of an imperative model. For example, the sphere should know how to tessellate itself, and send the commands to the imperative backend.
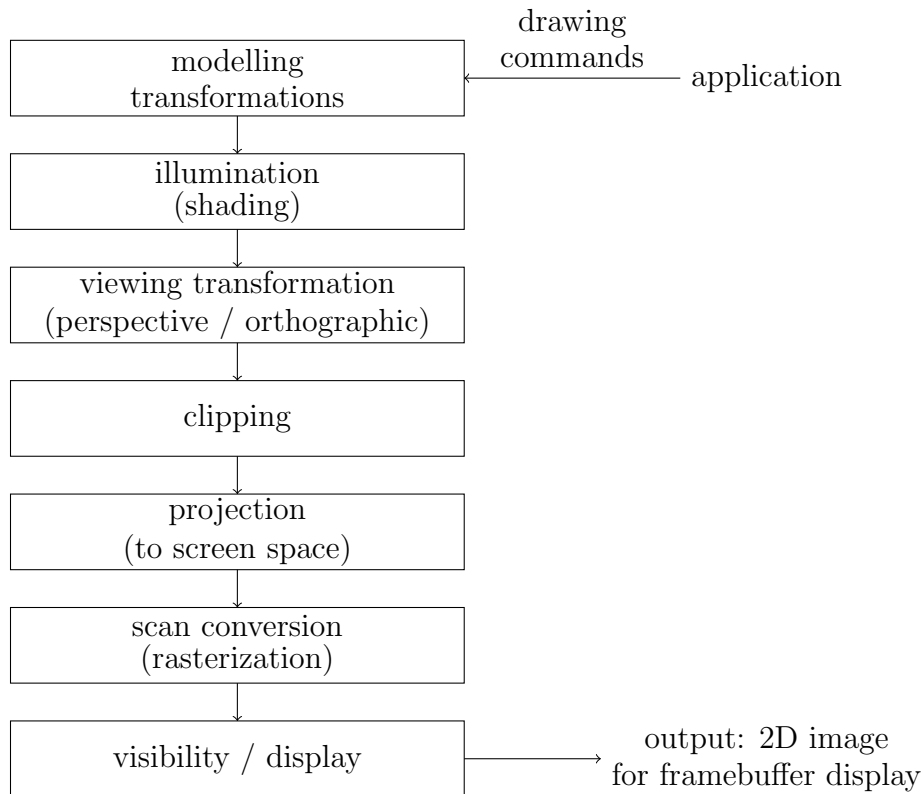
The steps of the pipeline perform the following;

- **modelling transformations**

  As 3D models are defined in the own coordinate systems (might all be at the origin, etc.), modelling transformations orients the models within the world coordinates (as a common coordinate frame). This can include scaling, rotation, and transformations.

- **illumination (shading)**

  Approximate lighting model to give initial ideas for how the object may be illuminated. The main illumination is done at this step, before clipping, in order to get accurate shadows (which may have been clipped).

- **viewing transformation**

  The world space is mapped into camera space. The viewing position is transformed to the origin, and the viewing direction is oriented along the z axis (typically).
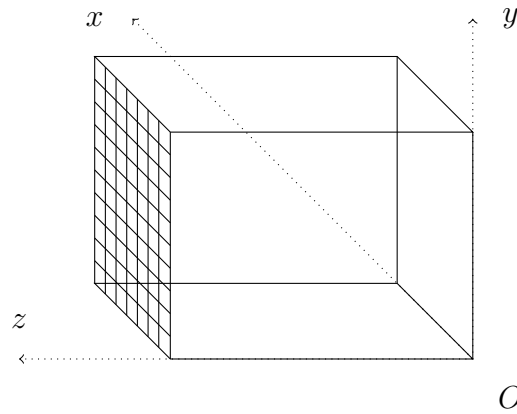
- **clipping**

  Remove portions of the scene outside the view frustum. Transformation is also performed to Normalised Device Coordinates (pictured below).
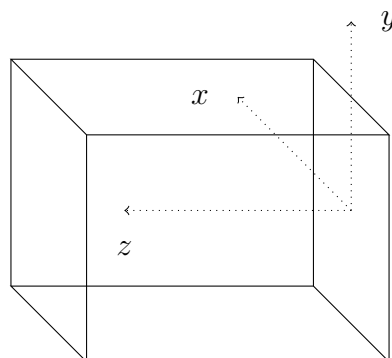
- **projection**

  The objects are projected into the 2D imaging plane screen space (see below);

- **rasterization**

  Converts objects into pixels, and interpolates values inside objects. Due to hardware support, it's more efficient to just rasterize everything.

- **visibility / display**

  Handles occlusions and transparency blending, as well as determining which objects are closest (and hence visible).

The majority of real-time graphics is based on the rasterization of graphic primitives (points, lines, triangles, etc). This is typically implemented in hardware (GPU), and is controlled through an API such as OpenGL. Certain parts of the pipeline are programmable, such as with GLSL (shaders).

A **vertex** is a point in space defining geometry and a **fragment** is a sample produced during rasterization (multiple of which are merged into pixels). The high level view of the pipeline is as follows;
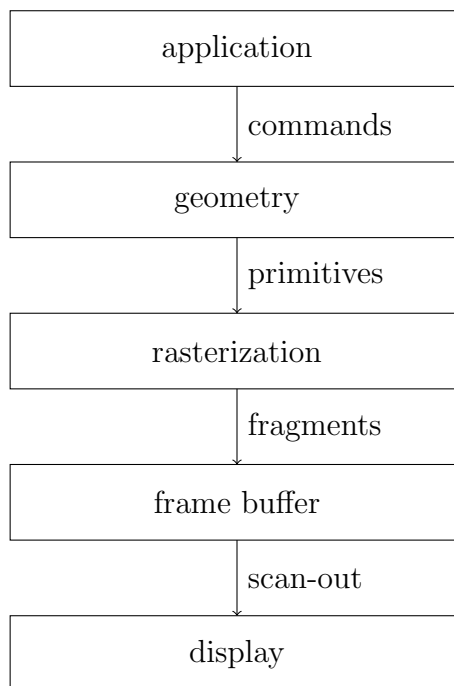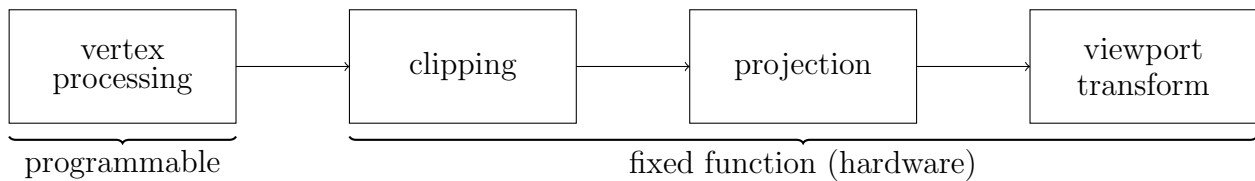

It's important to draw a distinction in which stage we're working. When we have geometry processing and write shaders to do something with our vertices, we are at a vertex stage (everything lives in 3D coordinates).
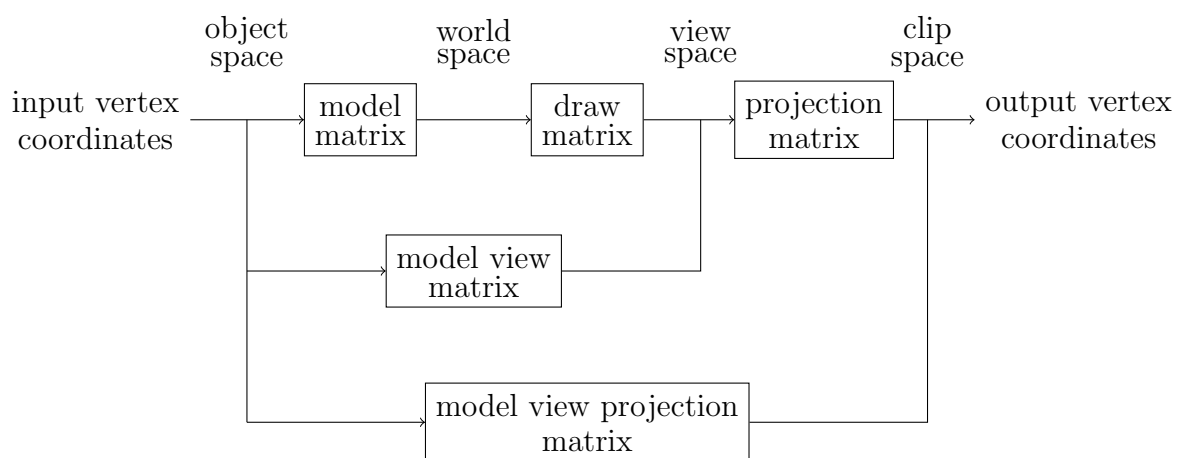
Application Stage

At the application stage, we typically generate a database of the scene (loaded from disks), and perform simulations (physics). Input events are also handled here. In this stage, objects are just vertex coordinates, and how they are connected.

The lecture now goes over the graphics pipeline in **OpenGL 3.2**. It's important to note that many fixed functions (including tessellation primitive generation, rasterization and interpolation) are implemented in hardware, for performance.

Geometry Stage

```
┌──────────┐      ┌──────────┐      ┌──────────┐      ┌──────────┐
│  vertex  │  →   │ clipping │  →   │projection│  →   │ viewport │
│processing│      │          │      │          │      │transform │
└──────────┘      └──────────┘      └──────────┘      └──────────┘
  programmable              fixed function (hardware)
```
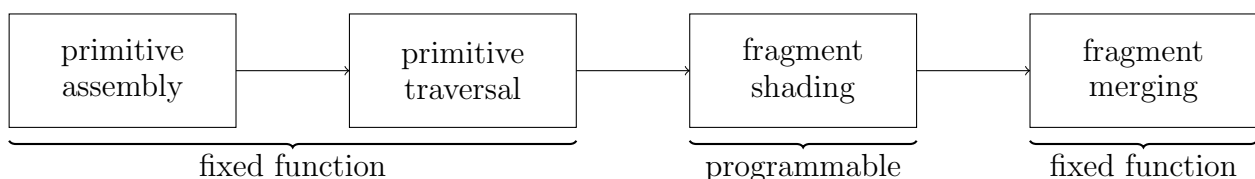
In the vertex processing stage, the input vertex stream (composed of arbitrary vertex attributes such as position and colour) is transformed into a stream of vertices mapped onto the screen by the vertex shader. The following is the vertex post-processing pipeline (we can pre-multiply matrices if we don't need to do anything between the stages - typically we want to do stuff before the **projection matrix** however);

```
        object          world          view          clip
        space           space          space         space
                    ┌───────┐      ┌──────┐      ┌──────────┐
input vertex    →   │ model │  →   │ draw │  →   │projection│  →   output vertex
coordinates         │ matrix│      │matrix│      │  matrix  │      coordinates
                    └───────┘      └──────┘      └──────────┘
                        ┌────────────┐
                        │ model view │
                        │   matrix   │
                        └────────────┘
                    ┌────────────────────┐
                    │model view projection│
                    │      matrix         │
                    └────────────────────┘
```

The **geometry shader** is an optional stage between the vertex and fragment shader; it has full knowledge of the primitive it is working on (unlike the vertex shader). It can also generate primitives dynamically (such as procedural geometry in growing plants). Note that this is limited by the GPU.

Rasterization Stage

```
┌──────────┐      ┌──────────┐      ┌──────────┐      ┌──────────┐
│ primitive│  →   │ primitive│  →   │ fragment │  →   │ fragment │
│ assembly │      │ traversal│      │ shading  │      │ merging  │
└──────────┘      └──────────┘      └──────────┘      └──────────┘
        fixed function              programmable       fixed function
```

A lot of merging can be performed at the fragment shading stage. Also note that in graphics, we let $(0,0)$ denote the **lower left** corner of the window, and we refer to the lower left corner of a pixel. For example, the pixel centre (sample location) would be at $(2.5, 1.5)$;
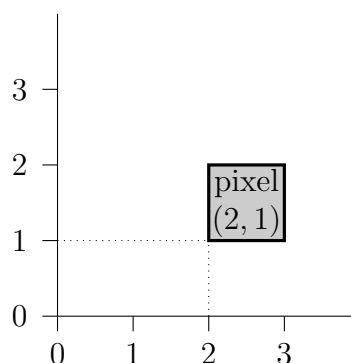
In the fragment shading step, given the interpolated vertex attributes (output by the vertex shader), the fragment shader computes colour values for each fragment, by applying textures, performing lighting calculations, etc. The fragment merging step follows the following pipeline;
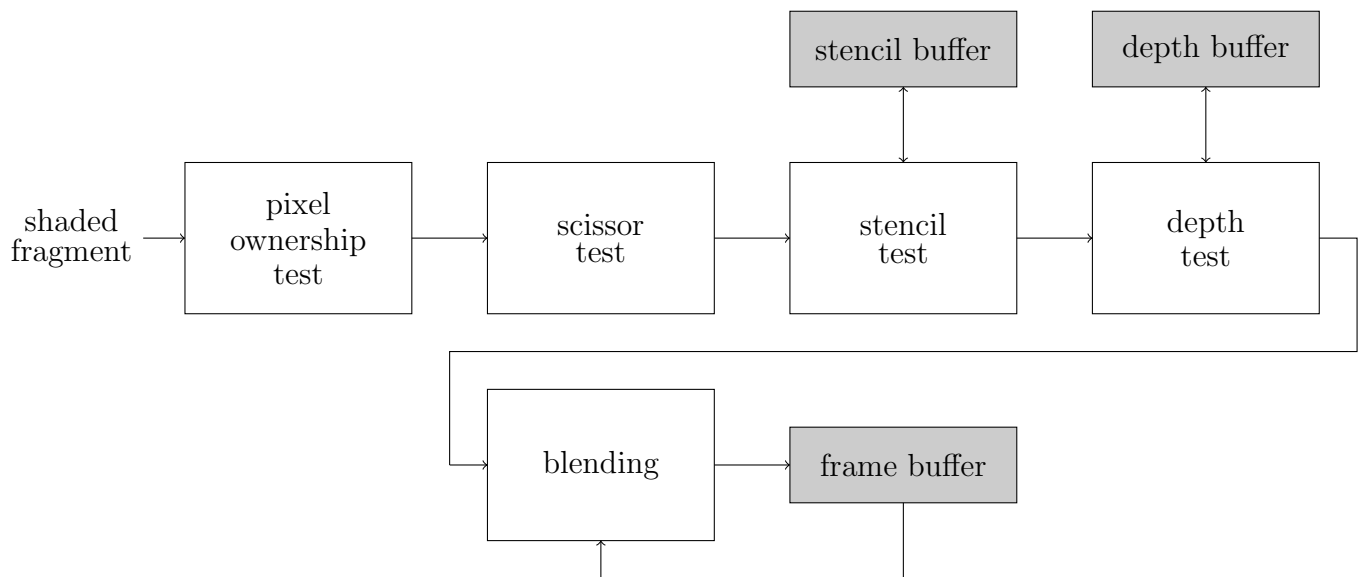
At this point, we have everything in the frame buffer, generally in the RGBA format, ready for the display stage.

Considerations with Pipeline

It's important to keep in mind that the size of the pipeline grows. While we may start with a few million vertices in a complex scene, the rasterizer may lead to billions of fragments / pixel candidates. Another note is that the vertex and fragment processing stages are highly parallel. Instead of iterating over the incoming list of vertices, we write programs that work on an individual vertex, for the GPU to perform in parallel (which is the power of GPU programming) - a similar concept applies for the fragment processing stage.

Architectural Overview

It's important to note that graphics hardware is a shared resource. The user mode driver (UMD) prepares command buffers for the hardware (and also provides a unified interface for different hardware), which is then submitted to the hardware by the kernel mode driver (KMD). The graphics kernel subsystem schedules hardware access.

The lecture continues with a list of graphics APIs.

Lecture 5 - Shading Languages

OpenGL

OpenGL is a low-level "immediate mode" graphics API specification (and not a library). It has a platform independent interface, with a platform dependent implementation. It defines an abstract rendering device, which can be operated by a set of functions, and therefore we don't need to care about what the actual hardware is. It also uses a **state machine** for high efficiency. In order to write a program, we need to do the following;

1. set up a **render window** (OS dependent, usually use a library such as *glut*, *Qt*, etc.)
2. setup viewport, model transformation, and file I/O (including shaders and textures)
3. frame generation (define what happens in every frame)

Contexts are abstract graphics devices, which usually do not communicate with each other; each representing one instance of OpenGL. There is only one **current** context per thread. We may have multiple OpenGL windows in a single application and they are independent contexts. A **resource** is something that is read from (some sort of data). They act as sources of inputs (such as texture images), and sinks for outputs (such as buffers).

The overall concept uses **object models**, where objects have unique names (unsigned integer handle). Commands work on targets, which has an object **bound** to the target. When a name is bound to a target, the object it identifies becomes current for that target. Exceptions exist such as shader objects and program objects, where commands work directly on object names. We can consider targets as analogous to types, and commands analogous to methods (drawing analogies to OOP).

Buffer Objects

A buffer object stores an array of unformatted memory allocated by the OpenGL context (the GPU) - and they are regular OpenGL objects. In order to set up the internal state, it must be bound to the context;

```
1 void glBindBuffer(enum target, uint bufferName)
```

To put immutable data on it, the command `void glBufferStorage(...);` is used, whereas `void glBufferData(...);` is used for mutable data. An example is as follows;

```
1  GLuint my_buffer;
2
3  // request an unused buffer object name
4  glGenBuffers(1, &my_buffer);
5
6  // bind name as GL_ARRAY_BUFFER
7  // bound for the first time => creates
8  glBindBuffer(GL_ARRAY_BUFFER, my_buffer);
9
10 // put some data into my_buffer
11 glBufferStorage(GL_ARRAY_BUFFER, ...);
12
13 // "unbind" buffer
14 glBindBuffer(GL_ARRAY_BUFFER, 0);
15
16 // bind it again
17 glBindBuffer(GL_ARRAY_BUFFER, my_buffer);
18 // use it
19
20 // example of drawing (type, startIdx, number of elements)
21 glDrawArrays(GL_TRIANGLES, 0, 33);
22
23 // delete buffer object, free resources, release buffer object name
24 glDeleteBuffers(1, &my_buffer);
```

Primitive Types
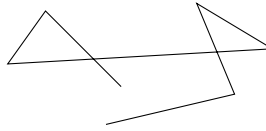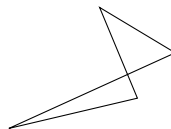
The primitive types available to us are;
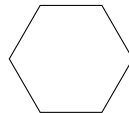
- GL_POINTS
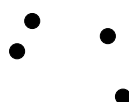
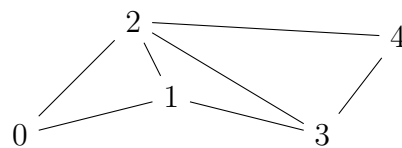- `GL_LINES`

- `GL_LINE_STRIP`

- `GL_LINE_LOOP`

- `GL_POLYGON`
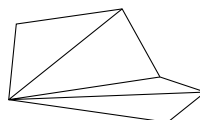
- `GL_TRIANGLE_STRIP`

It's important to note that here we alternate between identifying vertices; starting with a counter clockwise ordering, then clockwise, and so on. For example, the first triangle will be 0, 1, 2, then the second triangle is 1, 2, 3, and then the third triangle is 2, 3, 4. This allows vertices to be shared, and is therefore more memory efficient.
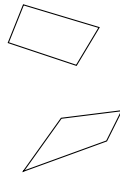
- `GL_TRIANGLES`
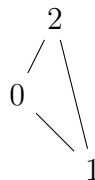
It's important to note that here we identify vertices counter clockwise (the triangle is 0, 1, 2).
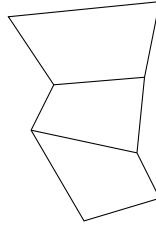
- `GL_TRIANGLE_FAN`

- `GL_QUADS`

- `GL_QUAD_STRIP`

A draw call, which isn't used after OpenGL 4, is done as follows;

```
1  glBegin(GL_TRIANGLE_STRIP); // primitive type
2  glColor3f(0.0, 1.0, 0.0); // colour state
3  glVertex3f(1.0, 0.0, 0.0); // vertex index
4  ...
5  glEnd();
```

The more efficient method of doing it is to do the following (using buffer objects) for continuous groups of vertices;

```
1  glDrawArrays(GL_TRIANGLES, 0, num_vertices);
```

Shaders

A shader is a piece of code that can be put in the graphics pipeline. To create a vertex or fragment shader, the following has to be done (anything that's not 'filled in' will default to fixed functions);
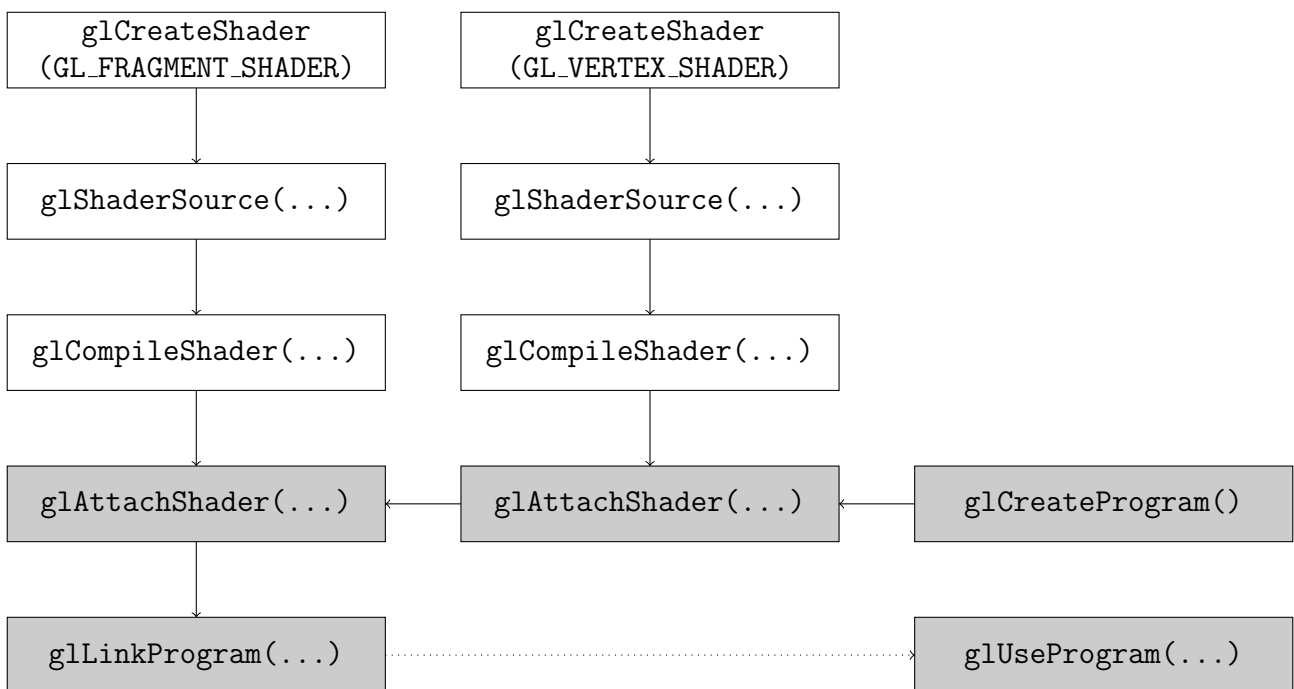
GLSL

We have the following types available, initialised with C++ style constructors (`vec3 a = vec3(1.0, 2.0, 3.0);`);

- **scalar types** `float`, `int`, `bool`
- **vector types** `vec2`, `vec3`, `vec4`, `ivec2`, `ivec3`, `ivec4`, `bvec2`, `bvec3`, `bvec4`
- **matrix types** `mat2`, `mat3`, `mat4`
- **texture sampling** `sampler1D`, `sampler2D`, `sampler3D`, `samplerCube`

Specific piece of hardware that does memory access very efficiently.

The standard C/C++ arithmetic and logic operators hold, as well as operators being overloaded for matrix and vector operations;

```
1  mat4 m;
2  vec4 a, b, c;
3
4  b = a*m;
5  c = m*a;
```

Vectors can be accessed via index (`[]`), `xyzw`, `rgba`, or `stpq`, therefore;

```
1  vec3 v;
2  v[1] == v.y == v.g == v.t
```

Swizzling is the following (not really mentioned in detail) - asking for objects in a different order;

```
1  vec3 a, b;
2  a.xy = b.yx;
```

`in` and `out` copy vertex attributes and other variables to / from shaders (they need to match the next step of the pipeline), whereas `uniform` denotes a variable from the application;

```
1  in vec2 tex_coord;
2  out vec4 colour;
3
4  uniform float time;
5  uniform vec4 rotation;
```

We are also given some functions;

- **arithmetic** e.g. `sqrt`, `power`, `abs`
- **trigonometric** e.g. `sin`, `asin`
- **graphical** e.g. `length`, `reflect`
- users can also define functions

Built-in variables include `gl_Position` which denotes the output position from the vertex shader, and `gl_FragColor`, denoting the output colour from the fragment shader. This is only for ES, WebGL, and older versions of GLSL (present versions use an out variable).

The anatomy of a GLSL shader is as follows;

```
1  #version 400
2
3  uniform mat4 some_uniform // set by application (configuration values e.g. MVP
       Matrix)
4
5  // optional flexible register configuration between shaders (location stuff)
6  layout(location = 0) in vec3 some_input;
7  layout(location = 1) in vec4 another_input;
8
9  out vec4 some_output; // output definition for next shader stage
10
```

```
11  void main() {
12
13  }
```

An example of a fragment shader is as follows;

```
1   #version 400
2
3   uniform vec4 ambient;
4   uniform vec4 diffuse;
5   uniform vec4 specular;
6   uniform float shininess;
7
8   uniform vec4 lightPosition_camSpace; // light position in camera space
9
10  in fragmentData {
11      vec4 position_camSpace;
12      vec3 normal_camSpace;
13      vec2 textureCoordinate;
14      vec4 color;
15  } frag;
16
17  out vec4 fragColor;
18
19  void main(void) {
20      fragColor = frag.color;
21  }
```

Lecture 6 - Illumination, Shading, and Colour I

With very basic illumination, the result is a flat image with a single colour (and we cannot infer a shape). We want to give the impression that light acts in a more complex way. In local illumination, we focus on how **one** object interacts with the light sources; we do not consider how objects interact with other objects (such as reflections).

Physics of Shading

The light we want to model is whatever is reflected from our object. In this lecture, we only consider the brightness at each point. Illumination is dependent on various environmental factors, including the properties of the light source;

- intensity of emitted light
- distance to the point on the surface (light spreads out)

As well as object (surface) properties;

- surface normal vector
- object position relative to light source
- reflectivity / albedo (ability to absorb light energy) of the surface

Radiometry

$$e_\lambda = \frac{hc}{\lambda} \qquad\qquad\qquad \text{energy of a photon}$$

$$h \approx 6.63 \times 10^{-34} Js \qquad \qquad \text{Planck constant}$$

$$c \approx 3 \times 10^8 ms^{-1} \qquad \qquad \text{speed of light}$$

$$Q = \sum_{i=1}^{n} \frac{hc}{\lambda_i} \qquad \qquad \text{radiant energy of } n \text{ photons}$$

$$\Phi = \frac{\mathrm{d}Q}{\mathrm{d}t} \qquad \qquad \text{radiation / electromagnetic / radiant flux (in Watts)}$$

Radiant flux is the number of photons we have available to reflect at any one time. **Radiance** is defined as the radiant flux per unit solid angle per unit projected area; the number of photons per time at a small area in a particular direction (units are watts per meter^2 steradian);
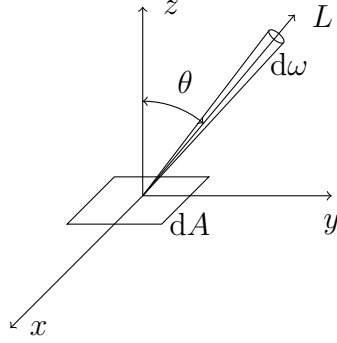
$$L(\omega) = \frac{\mathrm{d}^2\Phi}{\cos\theta \mathrm{d}A \mathrm{d}\omega}$$

Irradiance is how much light arrives from any other surface on another surface. It is the differential flux falling onto differential area (in Watts per meter^2).

$$E = \frac{\mathrm{d}\Phi}{\mathrm{d}A}$$

It can be seen as a density of incident flux falling onto a surface, and can be obtained by integrating the radiance over the solid angle.

Reflection and Reflectance

The actual property we want to model is reflection. We want to work out how much is reflected to the viewer. **Reflection** is the process by which electromagnetic flux incident on a surface leaves the surface without a change in frequency (fluorescence), and **reflectance** is a fraction of the incident flux that is reflected. We do not consider absorption, transmission nor diffraction.

We want to find out how much is reflected when E_i reaches an infinitely small patch of surface. To model this, we need the Bidirectional Reflectance Distribution Function (BRDF) (also seen in **CO316**) - the units are steradian^{-1};

$$f_r(\theta_i, \phi_i, \theta_r, \phi_r) = \frac{\mathrm{d}L_r(\theta_r, \phi_r)}{\mathrm{d}E_i(\theta_i, \phi_i)}$$

In this case, L_r is the viewer, and E_i the light source.

Given our surface is well behaved and generally flat, we can simplify it by stating that a rotation along the surface normal does not change the reflectance;

$$f_r(\theta_i, \theta_r, \phi_r - \phi_i) = f_r(\theta_i, \theta_r, \phi_d) = \frac{\mathrm{d}L_r(\theta_r, \phi_d)}{\mathrm{d}(\theta_i, \phi_d)}$$

However, this isn't true for surfaces with strongly oriented nanostructures (such as *vantablack*, hair, fur, velvet). The properties of BRDFs are as follows;

- non-negative

$$f_r(\theta_i, \phi_i, \theta_r, \phi_r) \geq 0$$

- energy conservation   we never have more light come out than comes in

$$\forall \theta_i, \phi_i \quad \int_\Omega f_r(\theta_i, \phi_i, \theta_r, \phi_r)\, \mathrm{d}\mu(\theta_r, \phi_r) \leq 1$$

- reciprocity

$$f_r(\theta_i, \phi_i, \theta_r, \phi_r) = f_r(\theta_r, \phi_r, \theta_i, \phi_i)$$

In our case, we only consider the discrete version to compute reflected radiance (with n point light sources)- note that $\Phi_{s,j}$ denotes radiant flux, and d_j is the distance;

$$L_r(\omega_r) = \sum_{j=1}^n f_r(\omega_{i,j}, \omega_r) E_j = \sum_{j=1}^n f_r(\omega_{i,j}, \omega_r) \cos\theta_j \frac{\Phi_{s,j}}{4\pi d_j^2}$$

Some cases;

- **ideal diffuse reflectance**   depends on light source position and surface normal

  In this case we define BRDF as a constant. The only dependence is the position of our light, and how that is related to the normal vector of our surface. At a microscopic level, an ideal diffuse surface is very rough (such as chalk, clay, or some paints).

rough surface
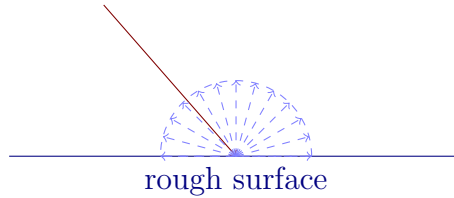
  This has a constant BRDF value;

$$L_r(\omega_r) = \int_\Omega f_r \omega_i, \omega_r \, \mathrm{d}E_i \omega_i$$
$$= f_r \int_\Omega \mathrm{d}E_i \omega_i$$
$$= f_r E_i$$
$$\mathrm{d}B = \mathrm{d}A \cos\theta_i$$

  Using the following, we can see that the reflection is maximal when it is parallel to the normal, and minimal when it is perpendicular;
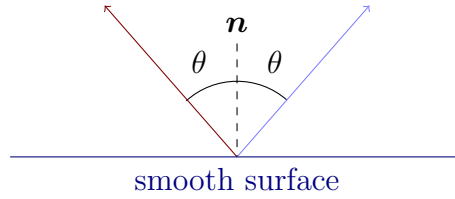

surface
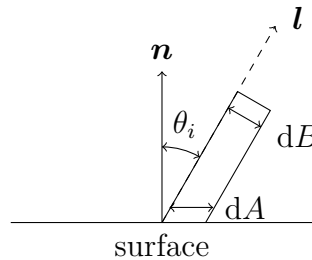
  These reflectors reflect light according to **Lambert's cosine law**; the more parallel the light is to the normal, the higher the strength of the outgoing light. This is physically incorrect, but it is an approximation, and gives a basic shading. From a single point light source **to** direction l, on a a surface with diffuse reflection coefficient k_d (the colour of the surface in the simplest case) and surface normal n;

$$L(\omega_r) = k_d(n \cdot l)\frac{\Phi_s}{4\pi d^2}$$

  The direction **vectors must be normalised**. Note that from this point on, using $n \cdot l$ denotes $\max(n \cdot l, 0)$; ensuring that the result is at least 0.

- **ideal specular reflectance**   depends on light source position, surface normal, and viewpoint position

In contrast to diffuse, reflection is **only** at mirror angle, and is view dependent. Nanostructures of the surface are usually oriented in the same direction as the surface (such as polished metals or mirrors).

smooth surface

This uses a special case of Snell's law, where the incoming ray, the surface normal, and reflected ray are all on a common plane;

$$n_l \sin \theta_l = n_r \sin \theta_r$$
$$n_l = n_r$$
$$\theta_l = \theta_r$$

What we want to do is figure out how our viewpoint is related to the reflectance direction.

- **non-ideal reflectors**

In the real world, materials are neither ideal mirror reflectors following Snell's law, nor ideal diffuse surfaces. A simple empirical model expects most of the light reflected to travel in the direction of the ideal ray; however due to surface nanostructures some of the light can be expected to be reflected slightly offset from the ideal. As we move further out (in the angular sense) from the reflected ray, we expect to see less light reflected.

slightly specular surface   highly specular surface
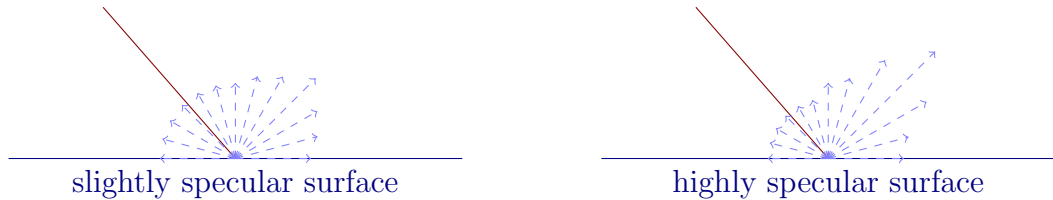
In the case of the slightly shiny surface, there is slightly higher intensity in the reflected direction, however in the highly shiny surface, there is high intensity in the reflected direction.

The **Phong** model states that the closer the viewer direction α is to the ideal reflection direction, the brighter the light should appear.

smooth surface
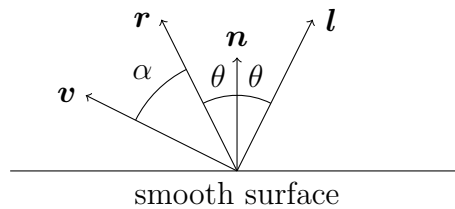
Therefore we have the following, where k_s is the specular reflection coefficient, and q the specular reflection exponent;

$$L(\omega_r) = k_s (\cos \alpha)^q \frac{\Phi_s}{4\pi d^2} = k_s (\boldsymbol{v} \cdot \boldsymbol{r})^q \frac{\Phi_s}{4\pi d^2}$$

A higher q leads to a more focused light (a shiner surface), and k_s is the surface property we want to manipulate. In order to obtain $\boldsymbol{r}$, we can calculate the following;

$$\boldsymbol{r} + \boldsymbol{l} = 2\cos\theta\boldsymbol{n} \qquad\qquad\qquad \Rightarrow$$
$$\boldsymbol{r} = 2(\boldsymbol{n} \cdot \boldsymbol{l})\boldsymbol{n} - \boldsymbol{l}$$
$$L(\omega_r) = k_s(\boldsymbol{v} \cdot (2(\boldsymbol{n} \cdot \boldsymbol{l})\boldsymbol{n} - \boldsymbol{l}))^q \frac{\Phi_s}{4\pi d^2}$$

The **Blinn-Phong** variation uses the halfway vector $\boldsymbol{h}$ between $\boldsymbol{l}$ and $\boldsymbol{v}$;

$$\boldsymbol{h} = \frac{\boldsymbol{l} + \boldsymbol{v}}{||\boldsymbol{l} + \boldsymbol{v}||}$$
$$L(\omega_r) = k_s(\cos\beta)^q \frac{\Phi_s}{4\pi d^2}$$
$$= k_s(\boldsymbol{n} \cdot \boldsymbol{h})^q \frac{\Phi_s}{4\pi d^2}$$

- **ambient**

  This represents the reflection of all indirect illumination (it is a hack). This avoids the complexity of global illumination;

  $$L(\omega_r) = k_a$$

The Phong model is a sum of three components; the ambient, the diffuse reflection, and the specular reflection.

$$L(\omega_r) = k_a + (k_d(\boldsymbol{n} \cdot \boldsymbol{l}) + k_s(\boldsymbol{v} \cdot \boldsymbol{r})^q)\frac{\Phi_s}{4\pi d^2}$$

Note on Inverse Square Law

Light falls off according to an inverse square law, hence the d^2 term. However, it may not produce the best results, and we can instead often use $(d + s)$ in place, where s is a heuristic constant.

Shading

The three levels at which shading can be applied in polygon based systems, providing increasing realism at the cost of computation, are;

- **flat shading**

  This can be done wherever. Each polygon is shaded uniformly over is surface, and computed by taking a point in the centre and at the surface normal (consider a light source at infinity). Normally, only the diffuse and ambient components are used

- **Gouraud shading**

  Interpolates colour across triangles. This is fast, and supported by most GPUs, but cannot accurately model specular components (since we don't have normal vectors at each point on a polygon).

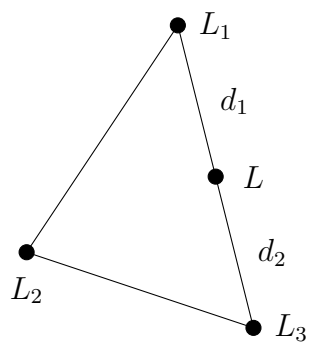- **Phong shading**   fragment stage

  Interpolates normals across triangles; more accurate modelling of specular components, but slower.

Interpolation shading is a more accurate way to render a shaded polygon, by computing an independent shade at each point;

1. compute a shade value at each vertex

2. interpolate to find shade value at the boundary

$$L = \frac{d_1 L_3 + d_2 L_1}{d_1 + d_2}$$

3. interpolate to find shade values in the middle

$$L = \frac{d_1 L_B + d_2 L_A}{d_1 + d_2}$$