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;

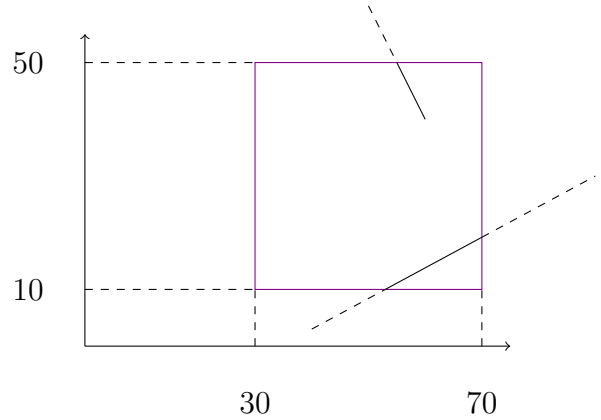
- `SetPixel(int x, int y, int colour);`
- `DrawLine(int xs, int ys, int xf, int yf);`

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;

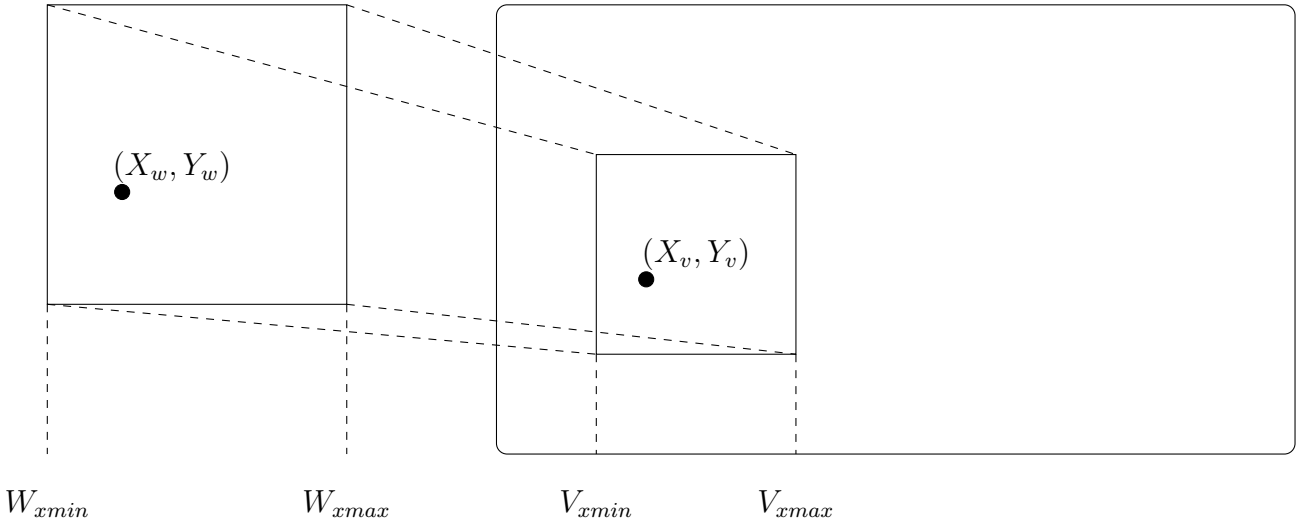
```
SetWindow(30, 10, 70, 50)
DrawLine(40, 3, 90, 30)
DrawLine(50, 60, 60, 40)
```



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;

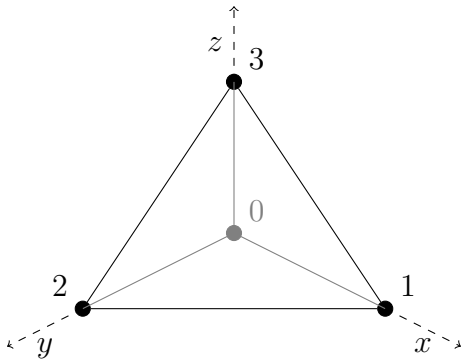
$$\frac{(X_w - W_{xmin})}{(W_{xmax} - W_{xmin})} = \frac{(X_v - V_{xmin})}{(V_{xmax} - V_{xmin})} \Rightarrow X_v = \frac{(X_w - W_{xmin})(V_{xmax} - V_{xmin})}{(W_{xmax} - W_{xmin})} + V_{xmin}$$

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

index	location
0	$(0, 0, 0)$
1	$(1, 0, 0)$
2	$(0, 1, 0)$
3	$(0, 0, 1)$

face data

index	vertices
0	0 1 3
1	0 2 1
2	0 3 2
3	1 2 3

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

Projections

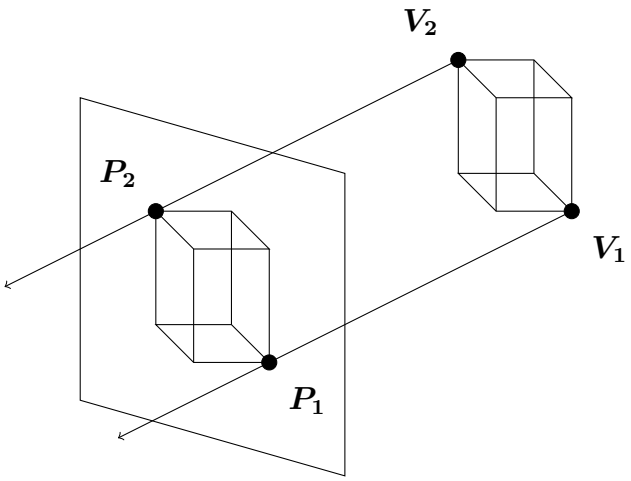
In order to draw a 3D wire frame, the points must first be converted into a 2D representation, via a **projection**, which can then be drawn with simple drawing primitives. Intuitively, we have an observer (a focal point) where all viewing rays converge. The observer is located between a projection surface 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 $\boldsymbol{P} = \boldsymbol{V} + \mu\boldsymbol{d}$;



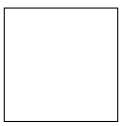
By substituting in the direction $\boldsymbol{d}$ we have determined, it gives the following Cartesian equations for each component;

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

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;

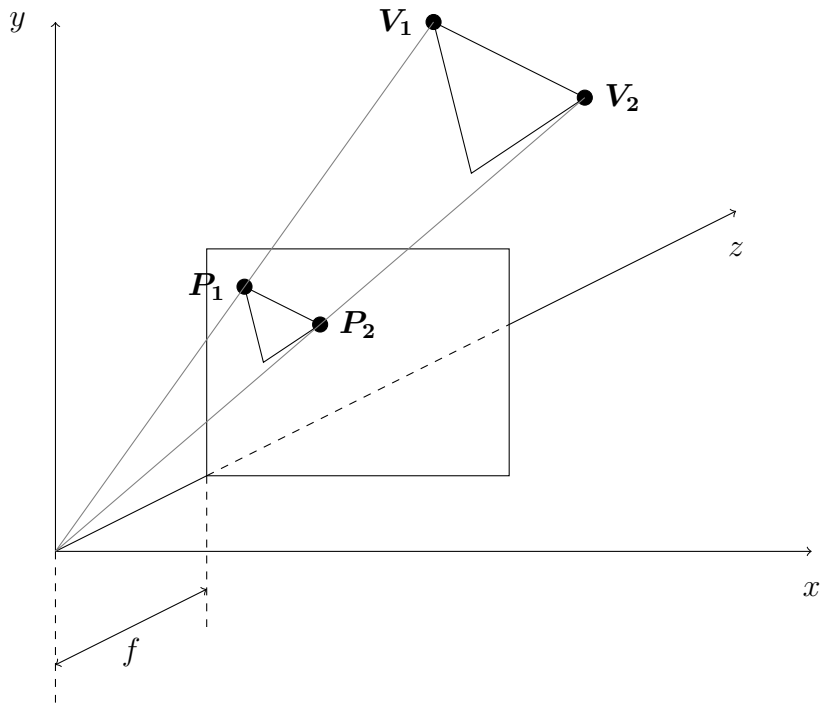
$$\boldsymbol{P} = \begin{bmatrix} V_x \\ V_y \\ 0 \end{bmatrix}$$

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);



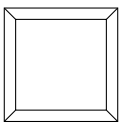
This gives us the following equation (since all projectors must go through the origin);

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

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

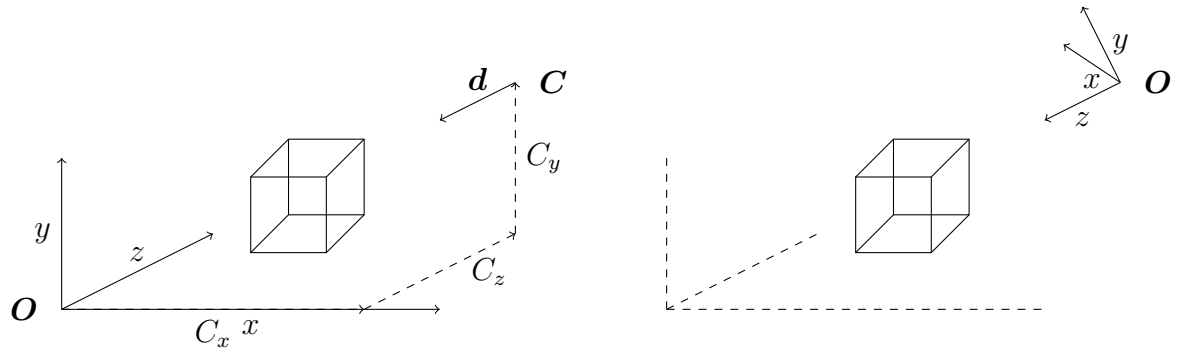
$$\begin{aligned}
P_z &= f && \text{by projection plane} \\
\mu_p &= \frac{P_z}{V_z} \\
&= \frac{f}{V_z} \\
P_x &= \mu_p V_x \\
&= \frac{f V_x}{V_z} \\
P_y &= \mu_p V_y \\
&= \frac{f V_y}{V_z}
\end{aligned}$$

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** by (s_x, s_y, s_z)

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

- **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;

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

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

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

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

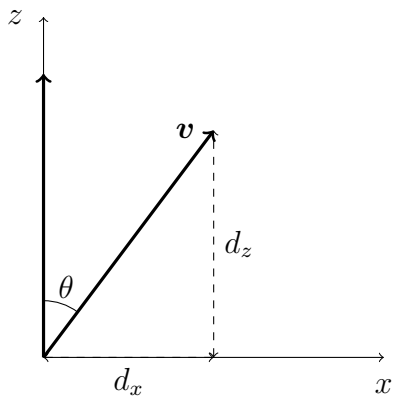
Recall the transformation in the previous lecture, which took way too long to draw, moving the origin to the view point. It consists of three steps, with the latter two being used to align the z-axis with the view direction;

1. translation of the origin fairly trivial to do

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

2. rotation about y-axis

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

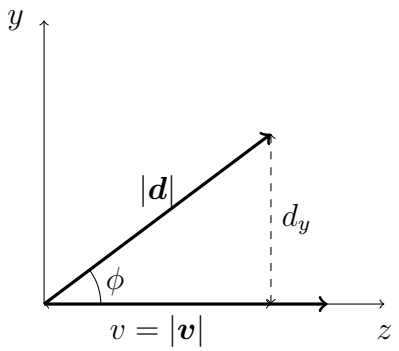


This can be used to calculate the following (note that here we are using a right-handed system);

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

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;

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

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

$$\boldsymbol{\mathcal{T}} = \boldsymbol{\mathcal{C}}\boldsymbol{\mathcal{B}}\boldsymbol{\mathcal{A}}$$

For every point $\boldsymbol{P}$ in the scene, we can obtain $\boldsymbol{P_t} = \boldsymbol{\mathcal{T}}\boldsymbol{P}$, 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;

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

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

$$\rightarrow \begin{bmatrix} \frac{fx}{z} \\ \frac{fy}{z} \\ f \\ 1 \end{bmatrix}$$ 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;

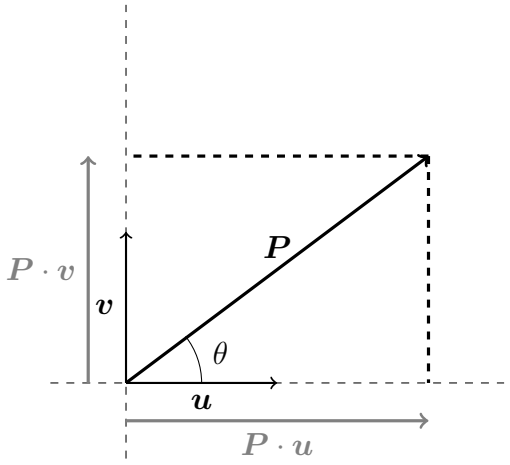
- $\boldsymbol{q}$ transformed x-axis
- $\boldsymbol{r}$ transformed y-axis
- $\boldsymbol{s}$ transformed z-axis
- $\boldsymbol{C}$ transformed origin

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 $\boldsymbol{P}_x^t$;

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

However, this can also be represented in matrix notation;

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

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 $\boldsymbol{p}$;

$$\boldsymbol{u} = \frac{\boldsymbol{p}}{|\boldsymbol{p}|}$$

$p_y = 0$ ensure horizontal, no vertical component

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

$$\boldsymbol{v} = \frac{\boldsymbol{q}}{|\boldsymbol{q}|}$$

$q_y = 1$ vertical, positive component

This gives us the following to solve;

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

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

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);

$$\boldsymbol{d} = \boldsymbol{p} \times \boldsymbol{q}$$

Using this, we can write $\boldsymbol{p}$ in terms of $\boldsymbol{q}$;

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

However, since we know that the vectors $\boldsymbol{p}$ and $\boldsymbol{q}$ are orthogonal, we can say the following;

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

Both of these equations can be fully written in terms of $\boldsymbol{d}$.

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;

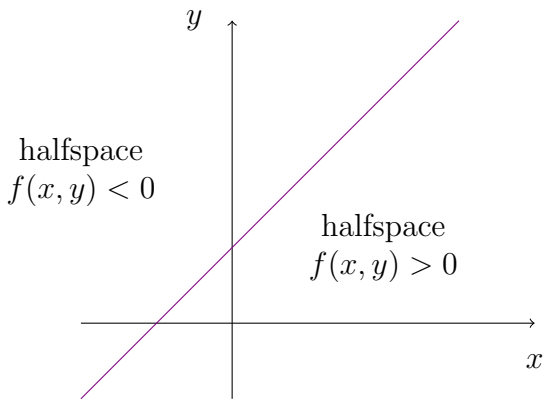
- before perspective transform (3D space) — world co-ordinates
- after perspective transform — homogeneous co-ordinates
- after perspective division — screen space

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;

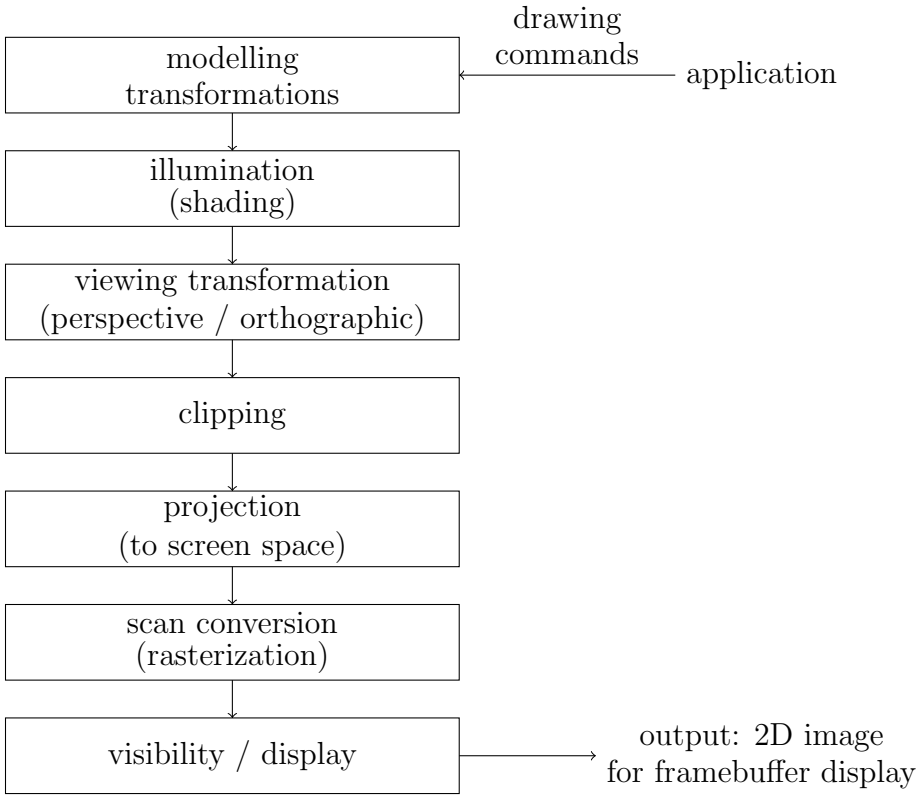
- **declarative** tell system what we want to render, not how

  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)**

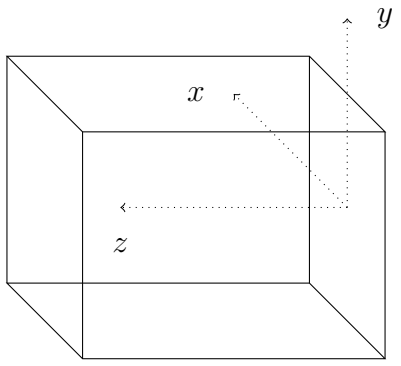
  Approximate lighting model to give initial ideas for how the object may be illuminated. The main illumination is done at this step, before clipping, in order to get accurate shadows (which may have been clipped).

- **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).

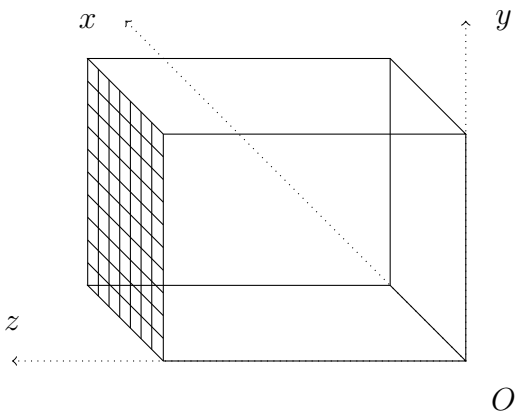
- **clipping**

  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**

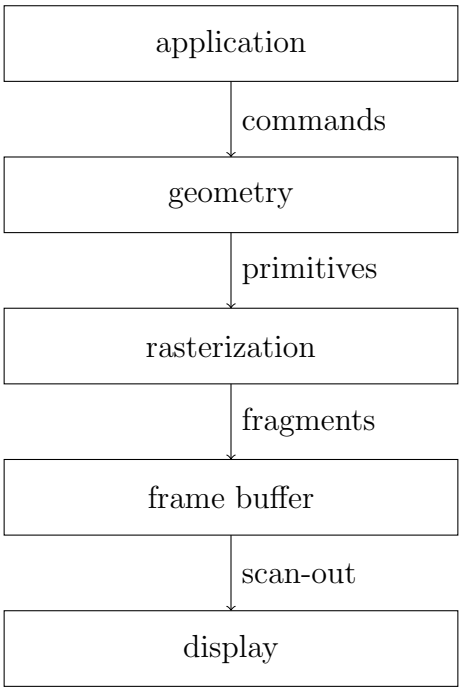
  Converts objects into pixels, and interpolates values inside objects. Due to hardware support, it's more efficient to just rasterize everything.

- **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;



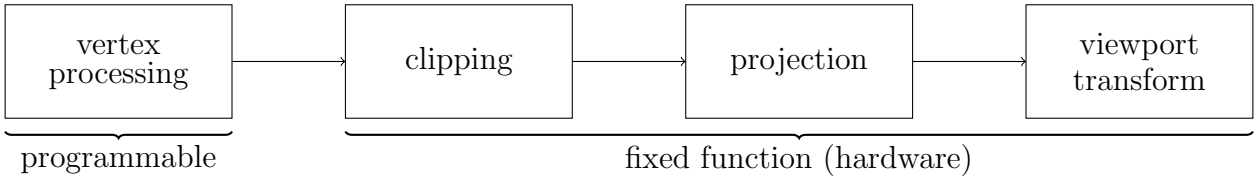
It's important to draw a distinction in which stage we're working. When we have geometry processing and write shaders to do something with our vertices, we are at a vertex stage (everything lives in 3D coordinates).

Application Stage

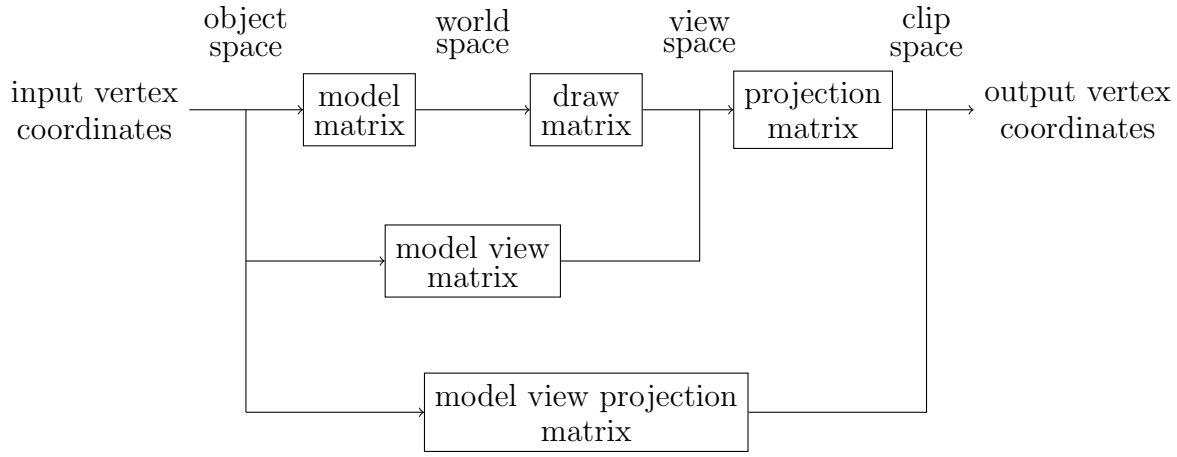
At the application stage, we typically generate a database of the scene (loaded from disks), and perform simulations (physics). Input events are also handled here. In this stage, objects are just vertex coordinates, and how they are connected.

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

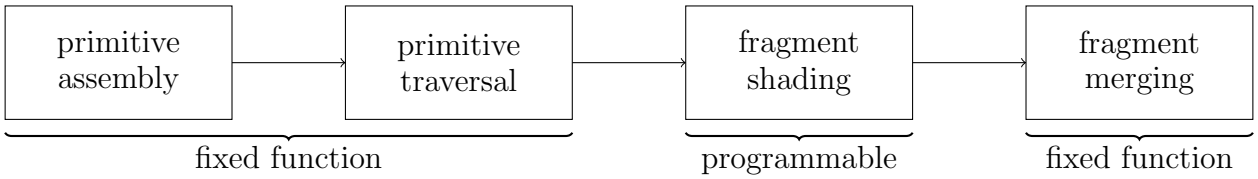


In the vertex processing stage, the input vertex stream (composed of arbitrary vertex attributes such as position and colour) is transformed into a stream of vertices mapped onto the screen by the vertex shader. The following is the vertex post-processing pipeline (we can pre-multiply matrices if we don't need to do anything between the stages - typically we want to do stuff before the **projection matrix** however);

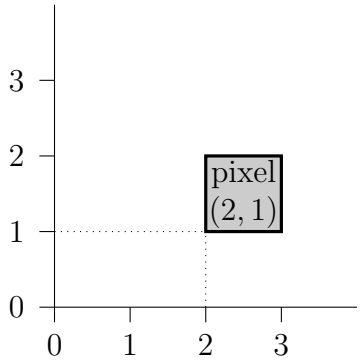


The **geometry shader** is an optional stage between the vertex and fragment shader; it has full knowledge of the primitive it is working on (unlike the vertex shader). It can also generate primitives dynamically (such as procedural geometry in growing plants). Note that this is limited by the GPU.

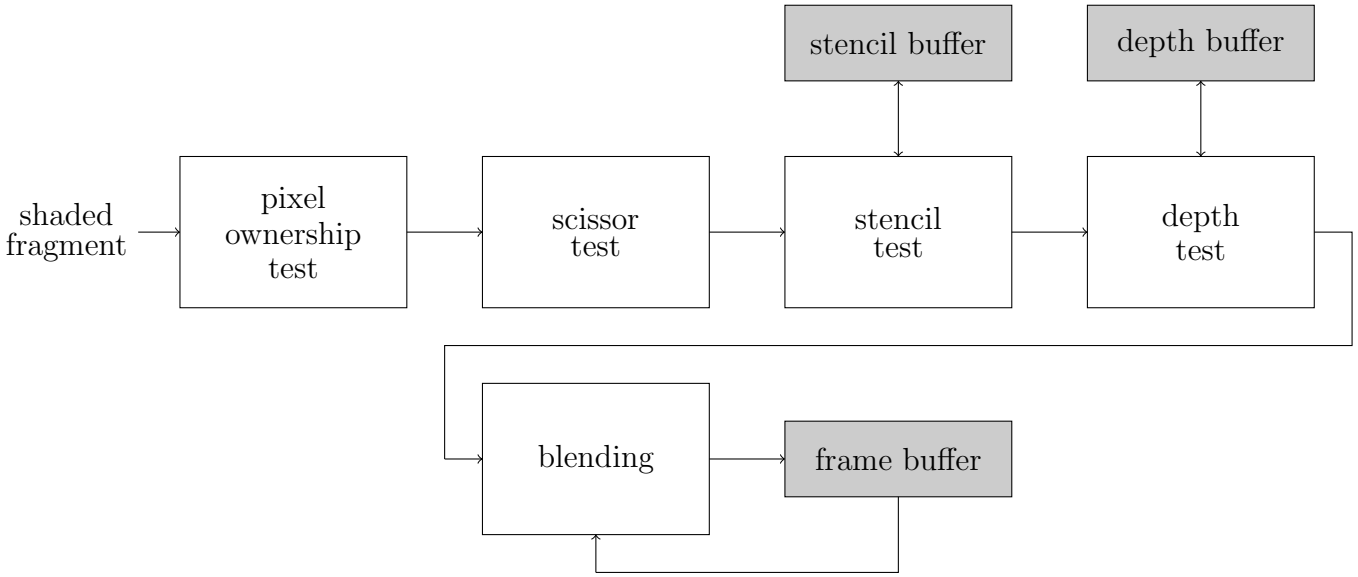
Rasterization Stage



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;

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

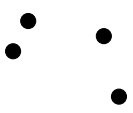
To put immutable data on it, the command `void glBufferStorage(...);` is used, whereas `void glBufferData(...);` is used for mutable data. An example is as follows;

```
GLuint my_buffer;

// request an unused buffer object name
glGenBuffers(1, &my_buffer);

// bind name as GL_ARRAY_BUFFER
// bound for the first time => creates
glBindBuffer(GL_ARRAY_BUFFER, my_buffer);

// put some data into my_buffer
glBufferStorage(GL_ARRAY_BUFFER, ...);

// "unbind" buffer
glBindBuffer(GL_ARRAY_BUFFER, 0);

// bind it again
glBindBuffer(GL_ARRAY_BUFFER, my_buffer);
// use it

// example of drawing (type, startIdx, number of elements)
glDrawArrays(GL_TRIANGLES, 0, 33);

// delete buffer object, free resources, release buffer object name
glDeleteBuffers(1, &my_buffer);
```

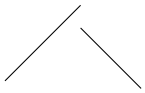
Primitive Types

The primitive types available to us are;

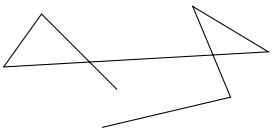
- `GL_POINTS`



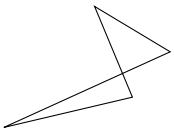
- GL_LINES



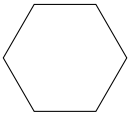
- GL_LINE_STRIP



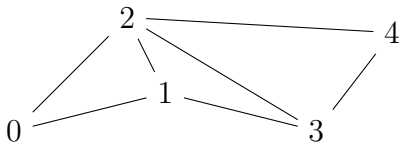
- GL_LINE_LOOP



- GL_POLYGON

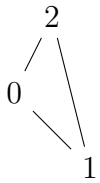


- GL_TRIANGLE_STRIP



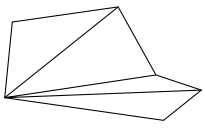
It's important to note that here we alternate between identifying vertices; starting with a counter clockwise ordering, then clockwise, and so on. For example, the first triangle will be 0, 1, 2, then the second triangle is 1, 2, 3, and then the third triangle is 2, 3, 4. This allows vertices to be shared,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

- GL_TRIANGLES

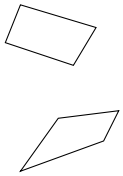


It's important to note that here we identify vertices counter clockwise (the triangle is 0, 1, 2).

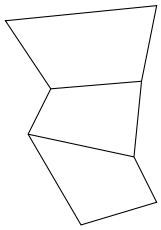
- GL_TRIANGLE_FAN



- GL_QUADS



- GL_QUAD_STRIP



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

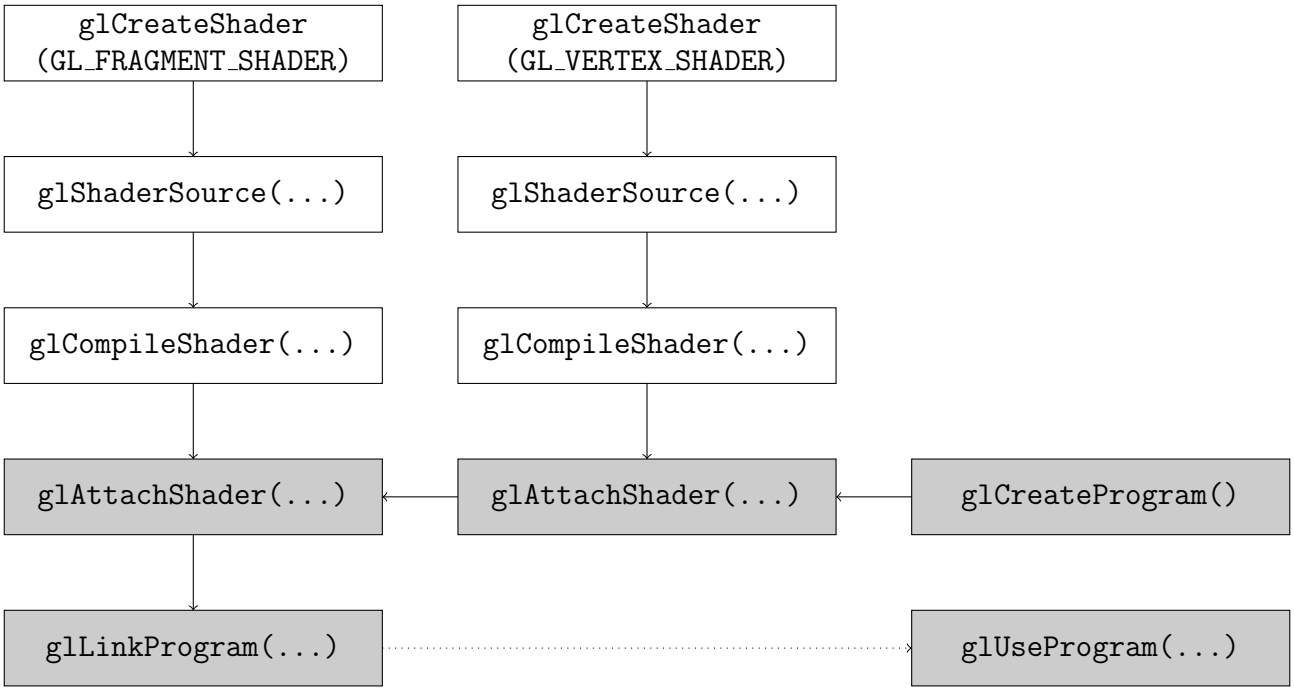
```
glBegin(GL_TRIANGLE_STRIP); // primitive type
glColor3f(0.0, 1.0, 0.0); // colour state
glVertex3f(1.0, 0.0, 0.0); // vertex index
...
glEnd();
```

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

```
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;

```
mat4 m;
vec4 a, b, c;

b = a*m;
c = m*a;
```

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

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

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

```
vec3 a, b;
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;

```
in vec2 tex_coord;
out vec4 colour;

uniform float time;
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;

```
#version 400

uniform mat4 some_uniform // set by application (configuration values e.g. MVP
    Matrix)

// optional flexible register configuration between shaders (location stuff)
layout(location = 0) in vec3 some_input;
layout(location = 1) in vec4 another_input;

out vec4 some_output; // output definition for next shader stage

```

```
void main() {

}
```

An example of a fragment shader is as follows;

```
#version 400

uniform vec4 ambient;
uniform vec4 diffuse;
uniform vec4 specular;
uniform float shininess;

uniform vec4 lightPosition_camSpace; // light position in camera space

in fragmentData {
    vec4 position_camSpace;
    vec3 normal_camSpace;
    vec2 textureCoordinate;
    vec4 color;
} frag;

out vec4 fragColor;

void main(void) {
    fragColor = frag.color;
}
```

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

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

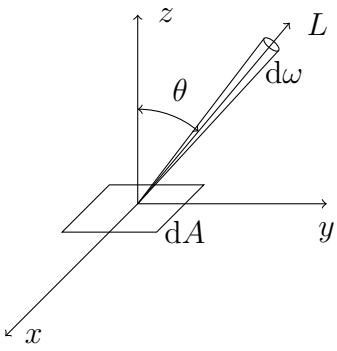
$h \approx 6.63 \times 10^{-34} Js$ Planck constant

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

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

$\Phi = \frac{\mathrm{d}Q}{\mathrm{d}t}$ 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

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

- reciprocity

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

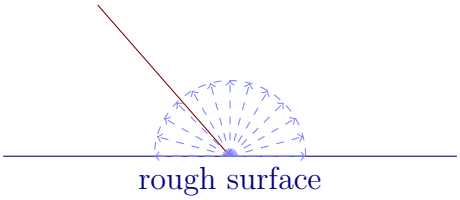
In our case, we only consider the discrete version to compute reflected radiance (with n point light sources)- note that $\Phi_{s,j}$ denotes radiant flux, and d_j is the distance;

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

Some cases;

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

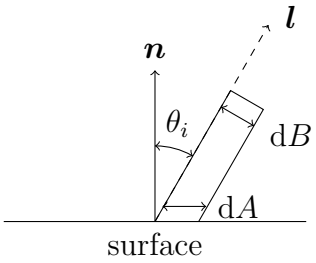
  In this case we define BRDF as a constant. The only dependence is the position of our light, and how that is related to the normal vector of our surface. At a microscopic level, an ideal diffuse surface is very rough (such as chalk, clay, or some paints).



  This has a constant BRDF value;

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

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



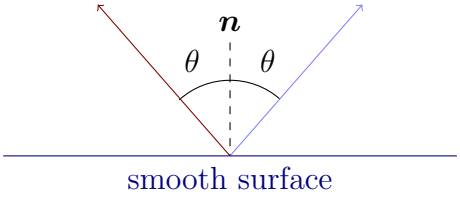
  These reflectors reflect light according to **Lambert's cosine law**; the more parallel the light is to the normal, the higher the strength of the outgoing light. This is physically incorrect, but it is an approximation, and gives a basic shading. From a single point light source **to** direction $\boldsymbol{l}$,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

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

  The direction **vectors must be normalised**. Note that from this point on, using $\boldsymbol{n} \cdot \boldsymbol{l}$ denotes $\max(\boldsymbol{n} \cdot \boldsymbol{l}, 0)$; 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).



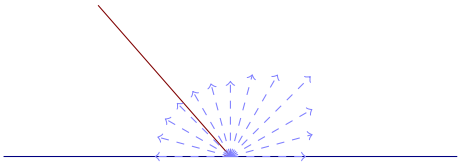
  This uses a special case of Snell's law, where the incoming ray, the surface normal, and reflected ray are all on a common plane;

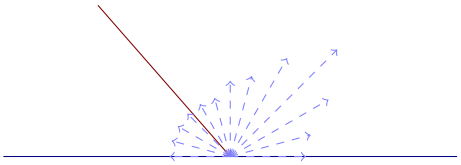
$$\begin{aligned} n_l \sin\theta_l &= n_r \sin\theta_r \\ n_l &= n_r \\ \theta_l &= \theta_r \end{aligned}$$

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

- **non-ideal reflectors**

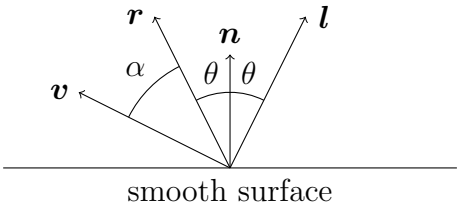
  In the real world, materials are neither ideal mirror reflectors following Snell's law, nor ideal diffuse surfaces. A simple empirical model expects most of the light reflected to travel in the direction of the ideal ray; however due to surface nanostructures some of the light can be expected to be reflected 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.





  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.



  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;

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

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

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

- **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).

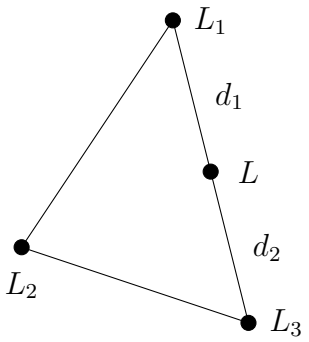
- **Phong shading** fragment stage

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

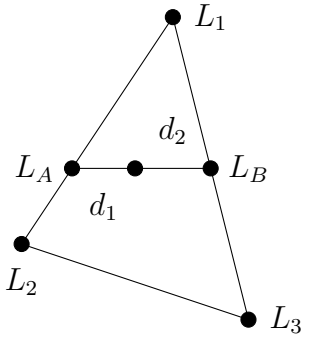
Interpolation shading is a more accurate way to render a shaded polygon, by computing an independent shade at each point;

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



Lecture 7 - Illumination, Shading, and Colour II

Colours are energy distributions. Lasers are light sources that contain a single / narrow band of wavelengths. Light is made up of a mixture of many wavelengths, with an energy distribution. This lecture starts with a lot about colour, similar to **CO316**. The **tri-stimulus colour theory** states receptor performance implies that colours do not have a unique energy distribution, and more importantly colours which are a distributed over all wavelengths can be matched by mixing red, green, and blue.

Colour Matching

Given any colour light source, we can try to match it with a mixture of three light sources (with R, G, B being pure light sources, and r, g, b being their respective intensities);

$$X = rR + gG + bB$$

However, not all colours can be matched with a given set of light sources - we cannot model anything that will take out colour. **Subtractive matching** allows us to add light to the colour we are trying to match, for example;

$$X + r = g + b$$

Printed documents cannot emit light, and are examples of subtractive matching (as opposed to mixing light emitters such as screens, which is additive matching).

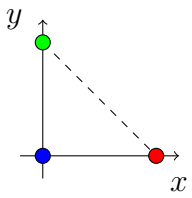
CIE Diagram

The CIE diagram was devised as a standard normalised representation of colours. Consider normalising the ranges between 0 and 1. The colours can be normalised such that the components sum to 1;

$$x = \frac{r}{r + g + b}$$

$$
\begin{aligned}
y &= \frac{g}{r+g+b} \\
z &= \frac{b}{r+g+b} \\
&= 1-x-y
\end{aligned}
$$

This has the following representation, where we have the hypothetical sources as follows;



However, the actual visible colours are a subset of this, done through manual testing. The pure colours (coherent λ) are around the edge of the diagram. In addition, the shape must be convex, since any blend would produce a colour in the visible region. When the three colours are components are equal, the colour is white ($x = y \approx 0.33$). Pure colours are fully saturated (colours on the edge of the horseshoe), and a line from the pure colour P going through the white point W will cross over on the other side at a **complement colour** C.

Conversion

Converting from RGB (monitor's representation) to CIE is done with the following;

$$
\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 0.628 & 0.268 & 0.150 \\ 0.346 & 0.588 & 0.070 \\ 0.026 & 0.144 & 0.780 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}
$$

A conversion between RGB and HSV is done as follows;

$$
\begin{aligned}
V &= \max(r,g,b) \\
S &= \frac{\max(r,g,b) - \min(r,g,b)}{\max(r,g,b)} \\
H &= \begin{cases} \text{undefined} & r = g = b \\ 120 \cdot \frac{g-b}{(r-b)+(g-b)} & (r > b) \wedge (g > b) \\ 120 + 120 \cdot \frac{b-r}{(g-r)+(b-r)} & (g > r) \wedge (b > r) \\ 240 + 120 \cdot \frac{r-g}{(r-g)+(b-g)} & (r > g) \wedge (b > g) \end{cases}
\end{aligned}
$$

Transparency

In addition, we can model transparency with an α channel;

- transparent $\alpha = 0$
- semi-transparent $0 < \alpha < 1$
- opaque $\alpha = 1$

Lecture 8 - Texture Mapping

With the current techniques we have, to create any detailed scenery, we'd need to create individual models for small details. The solution is to use images, in the form of textures on top of basic models. The key is to generate colours from some underlying function;

- one-dimensional functions

- two-dimensional functions
- three-dimensional functions
- raster images (texels) most common

A procedural texture maps a function $F(\boldsymbol{p})$ to a colour. This can be non-intuitive and quite difficult to match real textures.

Photo Textures

The idea is to define a 2D coordinate system on an image, which is then mapped onto a 3D object. For each fragment on an object's surface, we want to work out what coordinate needs to be sampled in the image's 2D space to get the right colour.

Conventionally, the texture coordinates are denoted with (s, t), for horizontal and vertical - really just a two-dimensional coordinate system. The object surface is similarly denoted with (u, v) and the pixel on the screen denoted with (x, y) - we need to know how the pixel on the screen maps to a (u, v) surface coordinate, and how the surface coordinate maps to a (s, t) texture coordinate. Typically, the mapping from (s, t) to (u, v) is manually predefined.

Paramterisation

A simple method is planar mapping, where we simply ignore one of the coordinates; however, this only looks good from the front of the object (any other sides will have some sort of smearing artifact). Another possible way is to build a cylinder (which can be easily parameterised, similar to a plane), and place the object inside. As expected, this works well for mostly cylindrical objects. A similar approach is to create a spherical mapping. One common method for environment mapping is **box mapping**. This creates a box around the object, which performs six planar mappings (one for each face).

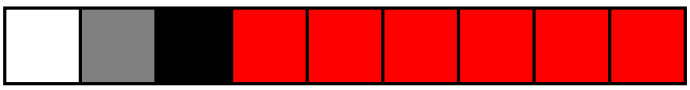
Unwrapping is the process of creating this manual mapping. Typically, all mappings have distortions and singularities, which will often require manual fixing.

Texture Coordinates

At each vertex, we specify a texture coordinate. The canonical texture coordinates go from $(0, 0)$ to $(1, 1)$. When we leave the boundaries of $[0, 1]$, we can take multiple approaches (note that for these examples, for my sake, I'm only drawing a simple 1×3 image). The original texture is the following;

Some of the approaches we can take are as follows, known as **texture addressing modes**;

- **static colour**

  Outside the border, we use a static colour (red in this case).
- **clamped**

  Outside the border, we use the last colour in the range (black, in this case).
- **repeated**

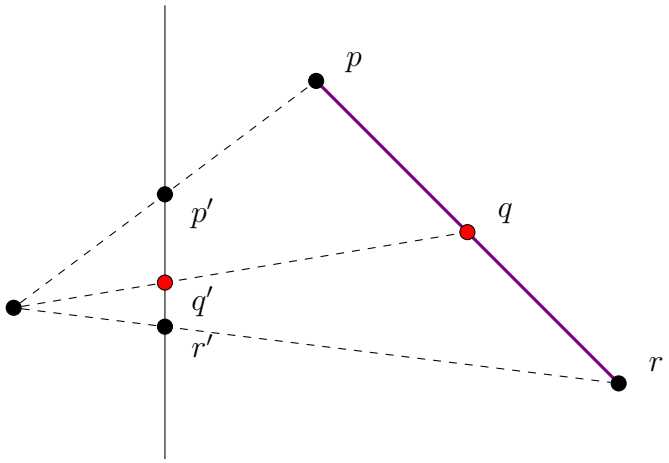
Once we exit the range, we simply wrap back to the first coordinate, repeating the texture.

- **mirrored**

Similar to above, however after ending the texture, we go backwards, and then back forwards through the texture (therefore 1.25 would map to 0.75).

Note that we cannot naively use a texture for tiling; there may be obvious seams if done incorrectly. If the boundaries at the top and bottom (and left and right) are the same, we are able to create a seamless tiling with repeating patterns. This is an efficient use of memory, as we are able to use a repeated small texture rather than a single large texture.

However, note that we cannot simply perform linear interpolation (as in Gouraud shading) on texture coordinates; this gives similar streaky artifacts to the specular in Gouraud shading. The reason is that we are performing a 3D interpolation in 2D; while this can be acceptable for colours in shading, for coordinates that undergo two transformations from screen space (x, y) to 3D (u, v), back to 2D (s, t), it is not acceptable. The problem is that perspective projection does not preserve linear combinations of points; equal distances in 3D **do not** map to equal distances in screen space. This can be shown in the following;



Let us assign some parameter t to the vertices p and r, with $t_p = 0$ and $t_r = 1$ (where t controls the linear blend of the texture coordinates of p and r). For simplicity, assume the image plane exists at $z = 1$ (or $f = 1$ in the projection matrix). Our goal is to work out the value of t_q, which ideally would be 0.5 (since it lies directly between p and q).

We can divide by the z coordinate;

$$p' = \frac{p}{z_p}$$
$$q' = \frac{q}{z_q}$$
$$r' = \frac{r}{z_r}$$

Note we cannot linearly interpolate t between p' and r', since only projected values can be linearly interpolated in screen space. We will instead need perspective-correct interpolation. The idea is to now linearly interpolate $\frac{t}{z}$ between p' and r', by computing $t_{p'}$ and $t_{r'}$;

$$t_{p'} = \frac{t_p}{z_p}$$

$$t_{q'} = \text{lerp}(t_{p'}, t_{r'})$$
$$t_{r'} = \frac{t_r}{z_r}$$

However, since we want to work out t_q, the un-projected parameter, rather than just $t_{q'}$, we do the following. First note that $t_{q'}$ is related to t_q by the perspective factor $\frac{1}{z_q}$, which we can obtain by linear interpolation;

$$\begin{matrix}\frac{1}{z_p} \\ \frac{1}{z_q} = \text{lerp}\left(\frac{1}{z_p}, \frac{1}{z_r}\right) \\ \frac{1}{z_r}\end{matrix}$$

This gives us the following;

$$t_q = t_{q'} z_q = \frac{\text{lerp}(t_{p'}, t_{r'})}{\text{lerp}\left(\frac{1}{z_p}, \frac{1}{z_r}\right)} = \frac{\text{lerp}\left(\frac{t_p}{z_p}, \frac{t_r}{z_r}\right)}{\text{lerp}\left(\frac{1}{z_p}, \frac{1}{z_r}\right)}$$

Therefore, given texture parameter t at vertices, we do the following (*unproject, project, unproject, and then normalise?*);

1. compute $\frac{1}{z}$ for each vertex
2. linearly interpolate $\frac{1}{z}$ across the triangle
3. linearly interpolate $\frac{t}{z}$ across the triangle
4. perform perspective division (divide $\frac{t}{z}$ by $\frac{1}{z}$ to obtain interpolated t)

The above is done automatically on the GPU's texture sampler; if we sample from the texture, it is implied (and we'd have to purposely modify the pipeline to induce distortions) - using the commands in GLSL will apply perspective corrected interpolation.

The lecture then covers bi-linear map, an alternative to map textures to individual pixels.

Texture Mapping and Illumination

Note that texture mapping can also be used to alter parts of the illumination equation. Typically, we can use the texture image as the diffuse component (but we are able to use it for other components, with varying effects).

When we shift lighting / camera position, we can notice a difference between a brick wall and an image of a brick wall mapped to a plane. It would be too difficult and time consuming to model individual geometry for bumps in the wall. We can instead feed in another texture to manipulate the normals - while it doesn't change the shape of the surface, it will cause shading to shade it as if it were a different shape.

However, bump-mapping will change the silhouette of a bump-mapped object; if we move the viewpoint for a texture-mapped object, the shape will remain the same, however the bump-mapped object will not. Displacement maps will actually move the surface point (modifying the positions of the vertices) - therefore the geometry must be displaced before visibility is determined.

Environment maps can be used to simulate reflections, by using the direction of the reflected ray to index a texture map at infinity. The texture map is dynamic, and would capture the environment around an object, however this is computationally expensive, and does not capture repeated bounces (two objects with environment maps will only show up once on each other, instead of infinitely).

Lecture 9 - Rasterization, Visibility, and Anti-aliasing

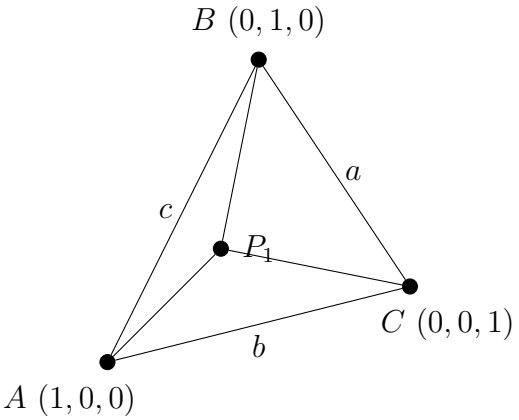
The rasterizer has two jobs; first to rasterize objects into pixels, and also to interpolate values inside objects. Note that at this point, we are already working in 2D. In the visibility part of the pipeline, we need to handle occlusion (for example, if something opaque is in the front, we likely want to only render that) - generally determines which objects are closest, and therefore visible.

Rasterization

The core of the algorithm is to check if a pixel is within a projected triangle, which is done for every pixel. While in reality, this is done fully in parallel, the general idea is still the same **occlusion test**. A simple method to deal with triangles not fully occupying a pixel is to check whether the centre of the pixel lies within a triangle - however, in practice there would be very little difference between implementations.

Barycentric Coordinates

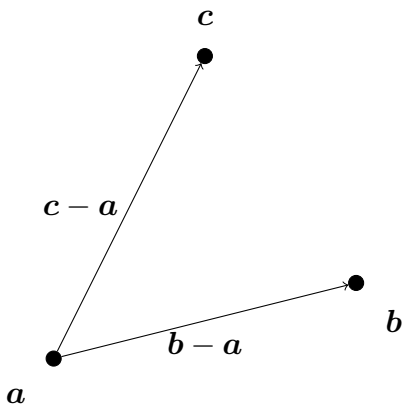
An approach for coordinates is to to define coordinates based on the areas of the subtriangles (note that the coordinates shown are linearly dependent - they are **not** 3D coordinates).



This stems from the idea of the barycentre, being the centre of mass. However, we can use it to characterise any point on a plane with masses relative to a triangle. There are two degrees of freedom (note we can define the third point in terms of the first two, so long as they are not collinear).

Points and Planes

We can use the vertices $\boldsymbol{a}$, $\boldsymbol{b}$, and $\boldsymbol{c}$ to specify points of a triangle; this can also be used to compute the edge vectors;



Note that here we have two vectors $\boldsymbol{b}-\boldsymbol{a}$ and $\boldsymbol{c}-\boldsymbol{a}$, which form a basis for a plane; hence three non-collinear points determine a plane. This non-orthogonal basis can be used to specify the location

of any point $\boldsymbol{p}$ on the plane (working with (β, γ) coordinates, rather than (x, y));

$$\boldsymbol{p} = \boldsymbol{a} + \beta(\boldsymbol{b} - \boldsymbol{a}) + \gamma(\boldsymbol{c} - \boldsymbol{a})$$

We can also reorder the equation as follows, to use (α, β, γ) as barycentric coordinates;

$$\begin{aligned}
&\alpha = 1 - \beta - \gamma \\
&\boldsymbol{p} = \boldsymbol{a} + \beta(\boldsymbol{b} - \boldsymbol{a}) + \gamma(\boldsymbol{c} - \boldsymbol{a}) \\
&\quad = (1 - \beta - \gamma)\boldsymbol{a} + \beta\boldsymbol{b} + \gamma\boldsymbol{c} \\
&\quad = \alpha\boldsymbol{a} + \beta\boldsymbol{b} + \gamma\boldsymbol{c}
\end{aligned}$$

In **homogenous barycentric coordinates**, the sum $\alpha + \beta + \gamma$ is normalised to be the area of the triangle, whereas **areal coordinates / absolute barycentric coordinates** have the coordinates normalised by the area of the original triangle, such that $\alpha + \beta + \gamma = 1$ (this is what we assume). Therefore, we have the following rules;

- **any point inside the triangle** $0 < \alpha, \beta, \gamma < 1$
- **edge** one coefficient is 0
- **vertex** two coefficients are 0, remaining must be 1

Note that a line $\beta = 0$ would go through $\boldsymbol{a}$ and $\boldsymbol{c}$, and a line $\beta = 1$ would go through $\boldsymbol{b}$ (and be parallel to the previous line).

Recall that an implicit equation in 2D is defined as;

$$f(x, y) = 0$$

Where only points with $f(x, y) = 0$ are on a given line. The general implicit form is;

$$Ax + By + C = 0$$

And an implicit line through (x_a, y_a) and (x_b, y_b) is;

$$(y_a - y_b)x + (x_b - x_a)y + x_a y_b - x_b y_a = 0$$

Therefore, given a triangle with vertices (x_a, y_a), (x_b, y_b) , and (x_c, y_c), we have the following three line equations;

$$\begin{aligned}
f_{ab}(x, y) &= (y_a - y_b)x + (x_b - x_a)y + x_a y_b - x_b y_a \\
f_{bc}(x, y) &= (y_b - y_c)x + (x_c - x_b)y + x_b y_c - x_c y_b \\
f_{ca}(x, y) &= (y_c - y_a)x + (x_a - x_c)y + x_c y_a - x_a y_c
\end{aligned}$$

Note that a barycentric coordinate such as β is a **signed distance** from a line (in this case, the line through $\boldsymbol{a}$ and $\boldsymbol{c}$). We need to choose k such that

$$k f_{ac}(x, y) = \beta$$

Since we know $\beta = 1$ at $\boldsymbol{b}$;

$$k f_{ac}(x_a, y_b) = 1 \Leftrightarrow k = \frac{1}{f_{ac}(x_b, y_b)}$$

Using this, we therefore have the barycentric coordinates for some arbitrary point $\boldsymbol{p}$ as;

$$\begin{aligned}
\alpha &= \frac{f_{bc}(x, y)}{f_{bc}(x_a, y_a)} \\
\beta &= \frac{f_{ac}(x, y)}{f_{ac}(x_b, y_b)}
\end{aligned}$$

$$\gamma = 1 - \alpha - \beta$$

In general, we can also define the barycentric area coordinates as the solution for the following linear system of equations;

$$\begin{bmatrix} x_a & x_b & x_c \\ y_a & y_b & y_c \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \\ \gamma \end{bmatrix} = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

We can convert from (t_1, t_2, t_3) in trilinear coordinates to barycentric coordinates $(t_1 a, t_2 b, t_3 c)$ (where we have the lengths of the sides). Similarly, conversion backwards would involve dividing through by a, b, and c.

Triangle Rasterization

One method of generating fragments for a triangle is to check (α, β, γ).

```
for all x do
  for all y do
    compute (alpha, beta, gamma) for (x, y)
    if (0 < alpha < 1 and
        0 < beta  < 1 and
        0 < gamma < 1) then
      c = alpha c0 + beta c1 + gamma c2
      draw pixel at (x, y) with colour c
```

It's important to note that we can also use the barycentric coordinates as weighting for colours; this can be done as we have normalised the coordinates. However in practice, this is done with optimised methods in the graphics hardware, including using fixed point precision (rather than floating-point), and done incrementally (withe the results from the previous pixel). Another optimisation is to take an axis aligned bounding box (by taking the minimum and maximum x and y coordinates).

Visibility

While the above works fine for a single triangle, generally we have multiple triangles in our scene. Typically, we render the triangle closest to the camera, as that would be in front of the other triangles (there are exceptions such as transparency).

Each pixel has a unique location in the framebuffer (image); however multiple fragments may be at the same address. Some approaches are as follows;

- **painter's algorithm**

  Here the triangles are sorted, using z values in camera space. The triangles are drawn from back to front (the highest z value would be drawn first). This not only suffers from efficiency issues due to the costly sorting, there are also issues with correctness. These issues could arise from intersections of triangles, as well as cycles of overlap; this could be solved by splitting triangles (which is also expensive).

- **depth buffer (z-buffer)**

  In this solution, we keep a 2D buffer which has the same size as the image, initialised with some infinitely large number in each position. When we render a fragment, we can also interpolate the z value, and when each fragment is drawn we keep the one with the lowest z value. When a smaller value is encountered, the previous colour is overwritten.

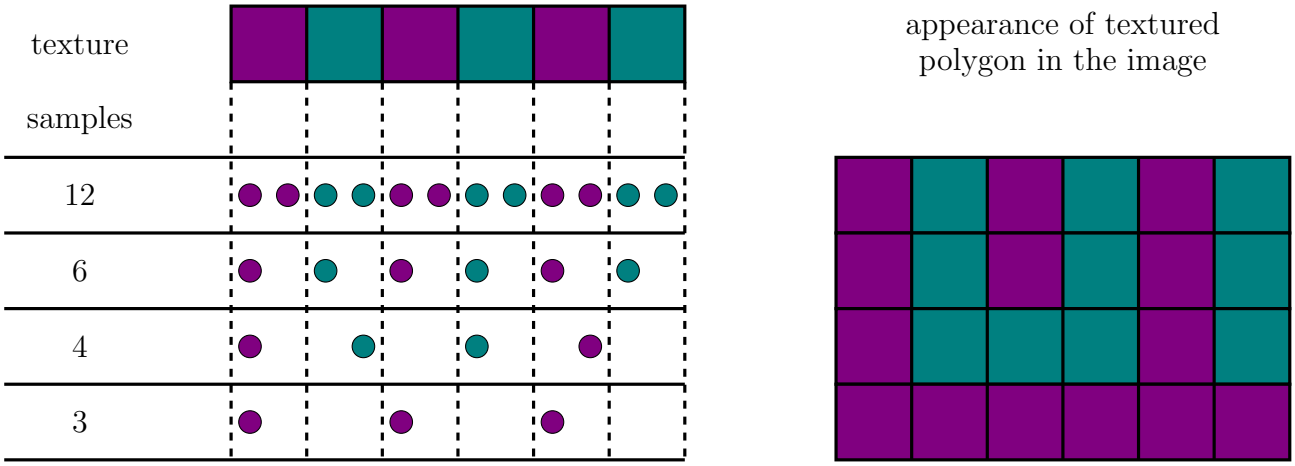
  ```
  let CB be colour (frame) buffer
  let ZB be z-buffer
  initialise z-buffer contents to be 1.0 (far)
  for each triangle T
  ```

```
5    rasterize T to generate fragments
6    for each fragment F with screen position (x, y, z) and colour C
7      if (z < ZB[x, y]) then
8        update colour CB[x, y] = C
9        update depth  ZB[x, y] = z
```

This method has multiple benefits, as it can handle intersections and cycles, as well as being simple to implement. It can draw opaque polygons in any order.

Alias Effect

Rasterization can cause alias effects, where straight lines or triangle boundaries may look gagged. These are caused by undersampling, and can cause visual artefacts. Undersampling can be shown as follows;



The general solution is to apply some blurring to the boundary, to reduce the effect. The most successful technique is to perform **supersampling**, which is then downsampled to the resolution of the screen. A local average can cause undesirable effects, therefore a weighted average is normally taken, such as the following (allowing us to sample from the texels adjacent, like in computer games);

$$\frac{1}{36}\begin{bmatrix}1 & 4 & 1\\ 4 & 16 & 4\\ 1 & 4 & 1\end{bmatrix}$$

Lecture 10 - Ray Tracing I

From here we look at a different method of generating images. Note that before we were unable to create infinitely reflecting surfaces, nor did we cover how to create shadows (which is an important depth cue) - including self shadowing. We were also unable to create distortions in reflective geometry or refraction.

Illumination

Note that we've only covered the **direct** illumination (which can be done easily with the Phong model). In direct illumination, a point receives light **directly** from all light sources in the scene. On the other hand, in **global** illumination, a point only receives light after **interacting** with other objects in the scene. As such, points might be in shadow or rays may be refracted through transparent material.

Ray Casting

A technique developed for pen-plotter in 1968 was to cast a ray per pixel (**from** the viewpoint outwards). If the ray from the viewpoint hits an object, that point is traced to the light source; if it doesn't intersect with anything else, the global illumination model is applied. However, if it intersects another object, it is not drawn (or filled in black). While the algorithm is quite simple, this is computationally expensive.

```
trace ray
  intersect all objects
  colour = ambient term
  for every light cast shadow ray
    colour += local shading term # add local shading term if light is not occluded
```

Turner Whitted took the concept further, evaluating not only shadows, but also reflections, refractions, and other effects. Note that casting a ray is simply a point plus a direction. At each intersection, if the surface is reflective, we can also cast another ray. If the surface is transparent, we can also attempt to perform refraction. This was the first global illumination mode; an object's colour is influenced by lights and other objects in the scene, and can simulate real lighting effects such as specular reflection and refractive transmission.

See slide 9 for a diagram

The algorithm is simple, but gives way for a recursive algorithm (note we can stop this at some arbitrary point);

```
trace ray
  intersect all objects
  colour = ambient term
  for every light cast shadow ray
    colour += local shading term # add local shading term if light is not occluded
  if mirror
    colour += k_refl * trace reflected ray
  if transparent
    colour += k_trans * trace transmitted ray
```

We can either stop after some recursion depth (after some number of bounces), or we can stop once the ray contribution (from the reflected / transmitted rays) become too small.

We have the following types of rays;

- **primary (viewing) rays**

  These are rays cast from the viewpoint to the nearest intersection. Local illumination is computed according to the Phong model;

  $$L = k_a + (k_d(\boldsymbol{n} \cdot \boldsymbol{l}) + k_s(\boldsymbol{v} \cdot \boldsymbol{r})^q)I_s$$

- **secondary rays**

  Secondary rays are any other rays (that aren't primary rays) - rays originating at intersection points, and can be caused by a number of things;

  – reflected ray

    This computes the mirror contribution, by casting a ray in the direction symmetric with respect to the surface normal, and multiply by the reflection coefficient. The direction of the ray is calculated in the same way as the ideal reflection ray (from the specular component), or can be worked out as follows, where $\boldsymbol{v'}$ is the direction of the secondary ray, $\boldsymbol{v}$ is the primary ray, and $\boldsymbol{n}$ is the unit surface normal;

    $$\boldsymbol{v'} = \boldsymbol{v} - (2\boldsymbol{v} \cdot \boldsymbol{n})\boldsymbol{n}$$

  – shadow ray

    Shadows can be added as follows, where we omit the diffuse and specular if the light source is obscured (hard shadows);

    $$L = k_a + s(k_d(\boldsymbol{n} \cdot \boldsymbol{l}) + k_s(\boldsymbol{v} \cdot \boldsymbol{r})^q)I_s + k_{\text{reflected}}L_{\text{reflected}} + k_{\text{refracted}}L_{\text{refracted}}$$

$$s = \begin{cases} 0 & \text{if light source obscured} \\ 1 & \text{if light source not obscured} \end{cases}$$

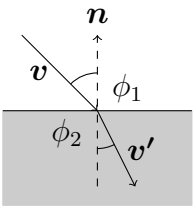
However, we can use multiple shadow rays to sample an area light source, allowing us to create soft shadows.

– transmitted (refracted) ray

The angle of the refracted ray is determined by Snell's law;

$$\eta_1 \sin(\phi_1) = \eta_2 \sin(\phi_2)$$

Note that η_1 and η_2 are constants for mediums 1 and 2, respectively. Similarly, ϕ_1 and ϕ_2 are angles between the respective rays and the surface normal.



Snell's law can also be written as the following, in vector notation;

$$k_1(\boldsymbol{v} \cdot \boldsymbol{n}) = k_2(\boldsymbol{v'} \cdot \boldsymbol{n})$$

The direction of the refracted ray can be written as;

$$\boldsymbol{v'} = \frac{\eta_1}{\eta_2}\left(\left(\sqrt{(\boldsymbol{n}\cdot\boldsymbol{v})^2 + \left(\frac{\eta_2}{\eta_1}\right)^2 - 1} - \boldsymbol{n}\cdot\boldsymbol{v}\right)\boldsymbol{n} + \boldsymbol{v}\right)$$

It's important to note that the above only has a solution given the following condition;

$$(\boldsymbol{n}\cdot\boldsymbol{v})^2 > 1 - \left(\frac{\eta_2}{\eta_1}\right)^2$$

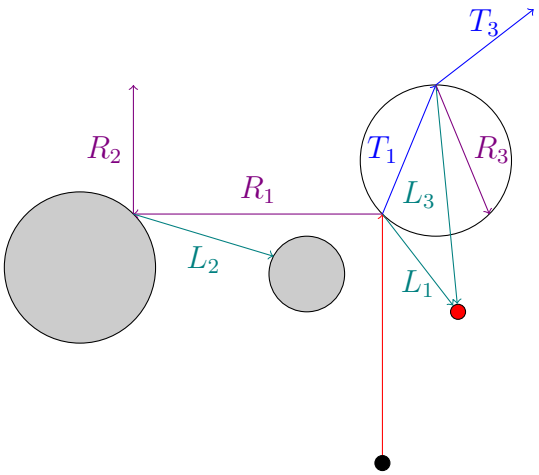
If the angle of the refracted ray is larger than 90° (due to a large incident ray), the ray becomes reflected rather than refracted (total internal reflection).

In the stack based implementation, we will also need to keep track of the material (η), and whether we are entering or leaving.

This allows us to express illumination as the following;

$$L = k_a + (k_d(\boldsymbol{n}\cdot\boldsymbol{l}) + k_s(\boldsymbol{v}\cdot\boldsymbol{r})^q)I_s + \underbrace{k_{\text{reflected}}L_{\text{reflected}} + k_{\text{refracted}}L_{\text{refracted}}}_{\text{recursion}}$$

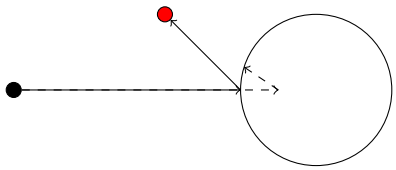
Note that the grey circles below denote **opaque** surfaces, and the red dot denotes a light source.



Since we cannot perform recursion in GLSL, this should be done iteratively (which can be done for the reflections, refraction would require state to be stored).

Precision Problems

However; while this will work completely fine in practice (as well as the majority of the time), consider the case where we have a sphere and a ray. Due to rounding errors, the intersection point between these two may end up **inside** inside the sphere (beneath the surface), and when we cast the shadow ray from the intersection point, it will end up intersecting with itself again, hence leading to unwanted self occlusion (note that the dashed lines represent the issue caused by the rounding error, and the solid lines represent the desired effect) - the effect is obviously exaggerated here;



As such, we can assign some ε value; if the length of the shadow ray (μ) is smaller than some ε, we assume it is on the surface, and shift the ray slightly in the required direction (along the surface normal) for it to be outside of the object.

Fresnel Factor

Traditional ray tracing has a constant coefficient for reflection. A better approach is to mix the reflected and refracted light according to the Fresnel factor;

$$L = k_{\text{fresnel}} L_{\text{reflected}} - (1 - k_{\text{fresnel}}) L_{\text{refracted}}$$

Schlick's approximation does the following;

$$k_{\text{fresnel}}(\theta) = k_{\text{fresnel}}(0) + (1 - k_{\text{fresnel}}(0))(1 - (\boldsymbol{n} \cdot \boldsymbol{l}))^5$$

Setting $k_{\text{fresnel}}(0) = 0.8$ gives a surface like stainless steel, apparently.

Monte-Carlo

A more computationally expensive approach which simulates the randomness of real light better is to cast multiple random rays from an intersection point (recursively). At some point, all of the intersection points are sent to the light source as before. On the other hand, Monte-Carlo path tracing only traces one secondary ray per recursion, but will send many primary rays per pixel.

Lecture 11 - Ray Tracing II

Calculating Intersections

For each ray, we need to find the nearest intersection point, and calculate all possible intersections with each object inside the viewing volume. Our scene can be defined with solid models (such as spheres and cylinders), as well as surface models (such as planes, triangles, and polygons). In general, anything that can be parameterised can be rendered with ray tracing.

Rays are parametric lines, defined as an origin $\boldsymbol{p_0}$ and a direction $\boldsymbol{d}$;

$$\boldsymbol{p}(\mu) = \boldsymbol{p_0} + \mu \boldsymbol{d}$$

For a ray, we can determine $\boldsymbol{d}$ as follows, where we have the position of the pixel on the viewing plane $\boldsymbol{p_0}$ and the position of the viewpoint $\boldsymbol{p_v}$;

$$\boldsymbol{d} = \frac{\boldsymbol{p_0} - \boldsymbol{p_v}}{|\boldsymbol{p_0} - \boldsymbol{p_v}|}$$

Since the viewing ray is parameterised by μ, $\mu > 0$ denotes the part of the ray **behind** the viewing plane, and $\mu < 0$ denotes the part of the ray in **front** of the viewing plane.

Consider a sphere, where any point on the surface $\boldsymbol{d}$ will satisfy the following (where r is the radius and $\boldsymbol{p_s}$ is the position in world space);

$$|\boldsymbol{q} - \boldsymbol{p_s}|^2 - r^2 = 0$$

Assuming that our viewing ray will intersect the sphere, we can substitute $\boldsymbol{q}$ with the line equation;

$$|\boldsymbol{p_0} + \mu\boldsymbol{d} - \boldsymbol{p_s}|^2 - r^2 = 0$$

If we allow $\Delta\boldsymbol{p} = \boldsymbol{p_0} - \boldsymbol{p_s}$ (and expand the dot product), we obtain the following quadratic equation (and corresponding solution);

$$\mu^2 + 2\mu(\boldsymbol{d} \cdot \Delta\boldsymbol{p}) + |\Delta\boldsymbol{p}|^2 - r^2 = 0 \Rightarrow \mu = \boldsymbol{d} \cdot \Delta\boldsymbol{p} \pm \sqrt{(\boldsymbol{d} \cdot \Delta\boldsymbol{p})^2 - |\Delta\boldsymbol{p}|^2 + r^2}$$

This gives us two solutions (since the ray would pass through the object) - in the case of solid objects, we want the one that's closer (hence the smaller μ). If the ray does not intersect, there will be no solution (negative under the square root; discriminant) - if this is the case, we want to stop in order to save computation. Once we have a value for μ, we can plug it back into the line equation to obtain the position.

A cylinder can be described by two position vectors ($\boldsymbol{p_1}$ and $\boldsymbol{p_2}$), describing the first and second end points of the long axis of the cylinder, and a radius r. The axis can be written as $\Delta\boldsymbol{p} = \boldsymbol{p_1} - \boldsymbol{p_2}$, and parameterised by $\alpha \in [0, 1]$. We can solve the following equation to obtain an intersection with the ray - note that $\boldsymbol{q}$ denotes a normal from the axis vector to the surface;

$$\boldsymbol{p_1} + \alpha\Delta\boldsymbol{p} + \boldsymbol{q} = \boldsymbol{p_0} + \mu\boldsymbol{d}$$

However, since we know $\boldsymbol{q} \cdot \Delta\boldsymbol{p} = 0$ (since it is a normal), we have;

$$\alpha(\Delta\boldsymbol{p} \cdot \Delta\boldsymbol{p}) = \boldsymbol{p_0} \cdot \Delta\boldsymbol{p} + \mu\boldsymbol{d} \cdot \Delta\boldsymbol{p} - \boldsymbol{p_1} \cdot \Delta\boldsymbol{p}$$

Solving for α gives the following result;

$$\alpha = \frac{\boldsymbol{p_0} \cdot \Delta\boldsymbol{p} + \mu\boldsymbol{d} \cdot \Delta\boldsymbol{p} - \boldsymbol{p_1} \cdot \Delta\boldsymbol{p}}{\Delta\boldsymbol{p} \cdot \Delta\boldsymbol{p}}$$

This can be substituted back into the equation to obtain;

$$\boldsymbol{q} = \boldsymbol{p_0} + \mu\boldsymbol{d} - \boldsymbol{p_1} - \left(\frac{\boldsymbol{p_0} \cdot \Delta\boldsymbol{p} + \mu\boldsymbol{d} \cdot \Delta\boldsymbol{p} - \boldsymbol{p_1} \cdot \Delta\boldsymbol{p}}{\Delta\boldsymbol{p} \cdot \Delta\boldsymbol{p}}\right)\Delta\boldsymbol{p}$$

Since we have $\boldsymbol{q} \cdot \boldsymbol{q} = r^2$, we can use the same approach for the quadratic equation for μ;

$$r^2 = \left(\boldsymbol{p_0} + \mu\boldsymbol{d} - \boldsymbol{p_1} - \left(\frac{\boldsymbol{p_0} \cdot \Delta\boldsymbol{p} + \mu\boldsymbol{d} \cdot \Delta\boldsymbol{p} - \boldsymbol{p_1} \cdot \Delta\boldsymbol{p}}{\Delta\boldsymbol{p} \cdot \Delta\boldsymbol{p}}\right)\Delta\boldsymbol{p}\right)^2$$

If this gives two solutions, μ_1, μ_2, there is an intersection, otherwise there is no intersection. Assuming that $\mu_1 < \mu_2$, we obtain the two following solutions;

$$\alpha_1 = \frac{\boldsymbol{p_0} \cdot \Delta\boldsymbol{p} + \mu_1\boldsymbol{d} \cdot \Delta\boldsymbol{p} - \boldsymbol{p_1} \cdot \Delta\boldsymbol{p}}{\Delta\boldsymbol{p} \cdot \Delta\boldsymbol{p}}$$
$$\alpha_2 = \frac{\boldsymbol{p_0} \cdot \Delta\boldsymbol{p} + \mu_2\boldsymbol{d} \cdot \Delta\boldsymbol{p} - \boldsymbol{p_1} \cdot \Delta\boldsymbol{p}}{\Delta\boldsymbol{p} \cdot \Delta\boldsymbol{p}}$$

If $\alpha_1 \in (0, 1)$, then the intersection is on the outside surface, whereas if $\alpha_2 \in (0, 1)$, the intersection is on the inside surface. If we only have one of these αs, we may have to intersect with some disc at the top or bottom of the cylinder.

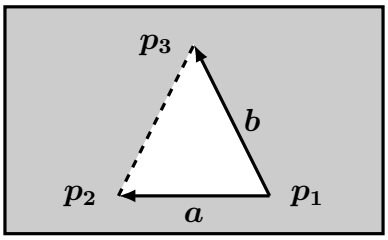
We can intersect a ray with a plane using the following (where $\boldsymbol{p_1}$ is a point on the plane, and $\boldsymbol{q}$ is a vector from the point on the plane to the intersection point);

$$\boldsymbol{p_1} + \boldsymbol{q} = \boldsymbol{p_0} + \mu$$

Since $\boldsymbol{q}$ would be perpendicular to the plane's normal $\boldsymbol{n}$, we can subtract $\boldsymbol{p_1}$ and multiply by the normal to obtain μ;

$$\boldsymbol{q} \cdot \boldsymbol{n} = 0(\boldsymbol{p_0} - \boldsymbol{p_1}) \cdot \boldsymbol{n} + \mu \boldsymbol{d} \cdot \boldsymbol{n} \Rightarrow \mu = -\frac{(\boldsymbol{p_0} - \boldsymbol{p_1}) \cdot \boldsymbol{n}}{\boldsymbol{d} \cdot \boldsymbol{n}}$$

Note that a triangle is a smaller part of a plane;



In order to calculate intersections for triangles, we need to test if the triangle is **front facing** ($\boldsymbol{d} \cdot \boldsymbol{n} < 0$), and whether the ray intersects the plane of the triangle (as before). We also need to test whether the intersection point is **inside** the triangle.

$\boldsymbol{a} = \boldsymbol{p_2} - \boldsymbol{p_1}$	
$\boldsymbol{b} = \boldsymbol{p_3} - \boldsymbol{p_1}$	
$\boldsymbol{n} = \boldsymbol{a} \times \boldsymbol{b}$	normal of plane
$\boldsymbol{q} = \alpha \boldsymbol{a} + \beta \boldsymbol{b}$	test intersection point inside triangle

We know that a point is inside the triangle if the all of the following three hold;

- $0 \leq \alpha \leq 1$
- $0 \leq \beta \leq 1$
- $\alpha + \beta \leq 1$

α and β can be calculated as follows;

$$\alpha = \frac{(\boldsymbol{b} \cdot \boldsymbol{b})(\boldsymbol{q} \cdot \boldsymbol{a}) - (\boldsymbol{a} \cdot \boldsymbol{b})(\boldsymbol{q} \cdot \boldsymbol{b})}{(\boldsymbol{a} \cdot \boldsymbol{a})(\boldsymbol{b} \cdot \boldsymbol{b})^2 - (\boldsymbol{a} \cdot \boldsymbol{b})^2}$$

$$\beta = \frac{\boldsymbol{q} \cdot \boldsymbol{b} - \alpha(\boldsymbol{a} \cdot \boldsymbol{b})}{\boldsymbol{b} \cdot \boldsymbol{b}}$$

A more efficient implementation of this concept is to translate the origin of the ray, and change the base of the vector yielding parameter vector $(t, u, v) \equiv (\mu, \alpha, \beta)$ - here we have t being the distance to the plane in which the triangle lies and (u, v) being the barycentric coordinates inside the triangle. The plane equation doesn't need to be computed on the fly (as we have done previously).

```
1 // v1,v2,v3 are triangle vertices,
2 // origin denotes the ray origin
3 bool triangle_intersection(vec3 v1, vec3 v2, vec3 v3, vec3 origin, vec3 ray_dir,
     float* out) {
4     vec3 edge1 = v2 - v1;
5     vec3 edge2 = v3 - v1;
6
7     // begin determinant calculation, also used for u
8     vec3 p = cross(ray_dir, edge2);
9
```

```
10      // if det is near 0, ray is parallel to (or in) plane
11      float det = dot(edge1, p);
12      if (det > -EPSILON && det < EPSILON) {
13          return false;
14      }
15
16      float inv_det = 1.0f / det;
17
18      // distance from v1 to ray origin
19      vec3 t = origin - v1;
20
21      // calculate u and test bound
22      float u = dot(t, p) * inv_det;
23
24      // intersection out of triangle
25      if (u < 0.0f || u > 1.0f) {
26          return false;
27      }
28
29      // prepare to test v parameter
30      vec3 q = cross(t, edge1);
31      float v = dot(ray_dir, q) * inv_det;
32
33      // intersection out of triangle
34      if (v < 0.0f || v > 1.0f) {
35          return false;
36      }
37
38      float mu = dot(edge2, q) * inv_det;
39      if (t > EPSILON) {
40          *out = mu;
41          return true;
42      }
43  }
```

Performance

The core algorithm is easy to implement, and extends well to global illumination (including shadows, reflection / refraction, multiple bounces, and atmospheric effects). However, there is a significant performance cost, as it takes multiple seconds per frame, rather than generating multiple frames per second.

In order to improve performance, we can reduce the number of bounces (however this may lead to less realistic lighting), and in generally we can reduce the number of rays. If we are rendering an object in a generally empty scene, we are wasting many rays that will never intersect. This can be accelerated by implementing an axis aligned bounding box for the object; where we discard any rays that don't intersect the box.

We can use the projection matrix to project the box, allowing us to immediately know which pixels will be inside the box (and which won't). Most complicated effects are implemented using multi-pass, or to use parts of the pipeline.

Another approach is to use a **regular grid**, where we divide the world into a grid, and at each region we store pointers to the objects that are in the region. A ray is shot through the space and only objects in the regions it passes are checked. This technique is rarely used. Instead, adaptive grids are use,

which subdivide the world until each cell contains no more than n elements (or some maximum depth d has been reached). This would be done in world space (rather than screen space), as this computation is expensive; we would rather do it once than to do it on every change of the camera position.

Another approach is to create a (Binary Space Partition (BSP) Tree), where the space is recursively partitioned by planes. Each cell is a convex polyhedron. Rays can be traced by recursion on the tree, and the construction enables a simple front-to-back traversal.