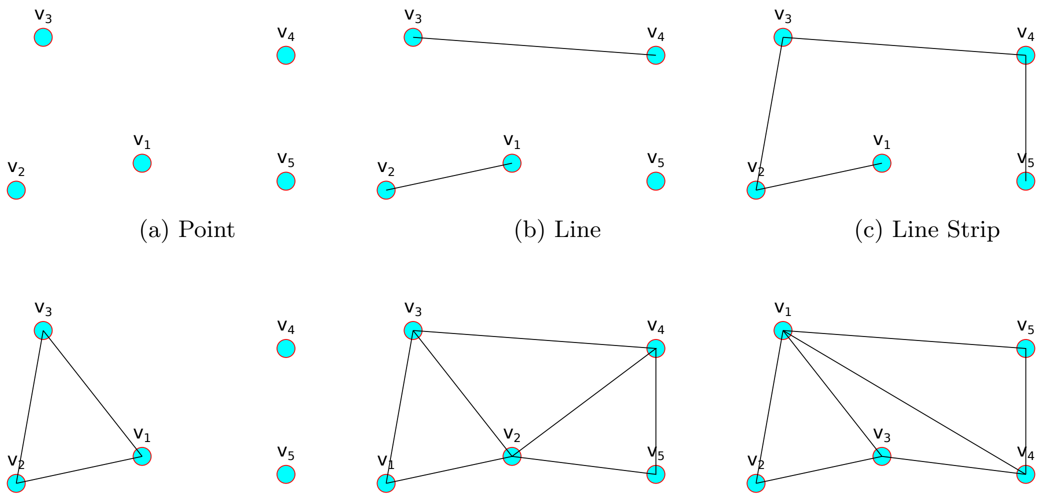
Contents

1 Overview

1.1 3D Graphics

- Scene made up of objects made up of primitives
- Primitives are a collection of vertices
- Vertices have attributes (position, colour, texture coordinate, etc.)

1.1.1 Primitives



(a) Point (b) Line (c) Line Strip

(d) Triangle (e) Triangle Strip (f) Triangle Fan

Figure 1

1.2 Graphics Pipeline

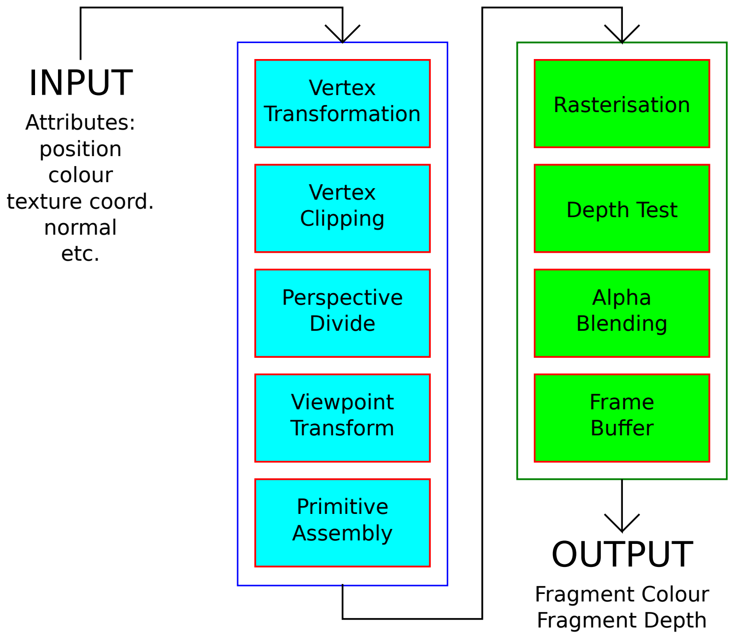


Figure 2: Pipeline

1.2.1 Vertex Operations

1 Vertices are transformed through a number of matrix operations (section 3)

2 Primitives outside screen space are removed (culled)

3 Vertices outside the screen space are clipped

4 3D scene is projected onto a 2D plane

1.2.2 Fragment Operations

1 Once screen area for a primitive is determined it is rasterised (section 2)

2 Each fragment is then shaded depending on colour, texture, lighting, etc.

2 Rasterisation

2.1 Line

Rasterise using Bresenham's line algorithm.

Very simple and fast algorithm for rasterising a line given two 2D points (x_0, y_0) and (x_1, y_1).

1 Determine if the line is steep or shallow with respect to the x axis
A steep line is closer being parallel with the y axis

2 Calculate gradient $m = \delta y / \delta x = (y_1 - y_0)/(x_1 - x_0)$

3 Iterate over the scan (longer) axis
Maintain coordinates of current pixel and *error* variable

  i Shade pixel at current coordinates

  ii Increment the scan axis of current coordinate

  iii $error = error + m$

  iv If $error \geq 0.5$ then increment the periodic axis of current coordinate and set $error = error - 1$

2.2 Triangle

1 Compute bounding box around triangle

2 Iterate through each pixel of each line

3 If the pixel is inside the triangle then shade it

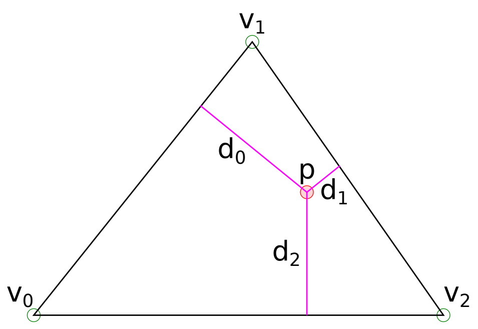
Two common methods for determining if a point is inside a triangle.

Both based on the creation of a temporary vertex p with coordinates of the pixel being tested.

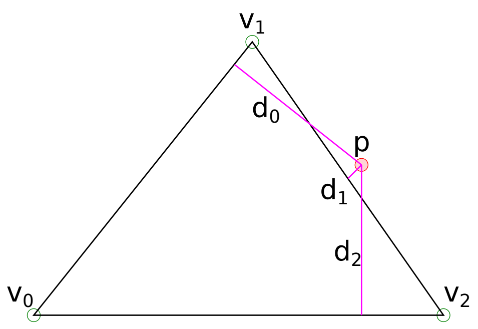
2.2.1 Line Equation method

1 Extend lines out from p to each edge of the triangle that are perpendicular to the edge

2 When the direction of each line is towards p, p is inside the triangle if the distance of each line is positive



(a) Inside triangle



(b) Outside triangle ($d_1 < 0$)

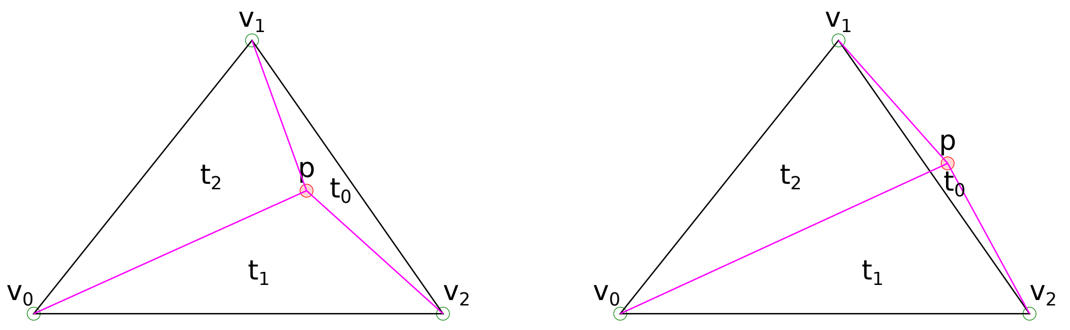
Figure 3

2.2.2 Barycentric Coordinate method

1 Create three new triangles (t_0, t_1 and t_2) using p and vertices of triangle

2 Calculate area of sub triangles as a proportion of the area of the original triangle (using shoelace formula)

3 If sum of areas of sub triangles $\sum_i t_i = 1$ then p is inside triangle



(a) Inside triangle (b) Outside triangle

Figure 4

Barycentric areas typically denoted as α, β and γ.

2.2.3 Shoelace formula

Method of calculating area of any 2D non self-intersecting polygon.

1 Create a matrix of the vertices in a closed loop

2 Multiply the first set of stepped pairs (s_1)

3 Multiply the second set of stepped pairs (s_2)

4 Calculate area $a = (s_1 - s_2)/2$

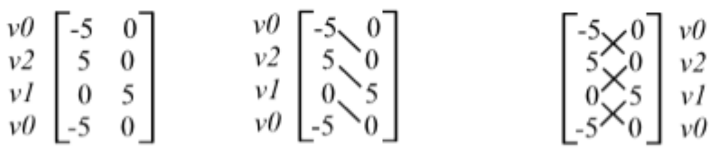


Figure 5: Shoelace formula example

2.2.4 Triangle spans

Triangles can also be rendered using spans, where it is made up of several lines.

Span start and end points are obtained using the lines between vertices v_0 and v_1 (start) and v_2 and v_1 (end).

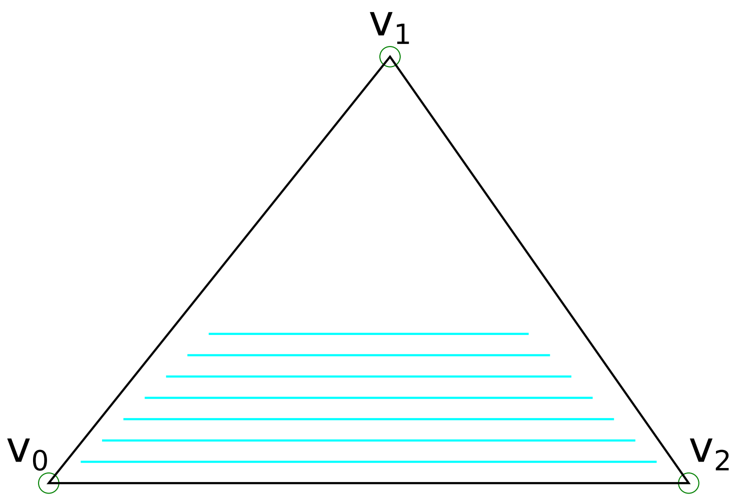


Figure 6: Triangle spans

3 Transformations

3.1 Spaces

World
: 3D space containing everything

Camera
: 3D space containing the view from the camera
Origin is camera position
Obtained through camera transform

Clip
: Only the primitives that can be seen by the camera
Obtained through perspective transform

Normalised Device Coordinates
: Transformed from clip space
Coordinates normalised to 1 for hardware compatibility

Viewport Coordinates
: Coordinates on a particular screen

3.2 Scale

Scale matrix to scale by x, y and z is each respective axis:

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

3.3 Translation

Translation matrix to move by x, y and z is each respective axis:

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

3.4 Rotation

Rotation about x axis by θ:

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

Rotation about y axis by θ:

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

Rotation about z axis by θ:

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

3.5 Perspective

Perspective matrix used to give perspective to camera space.

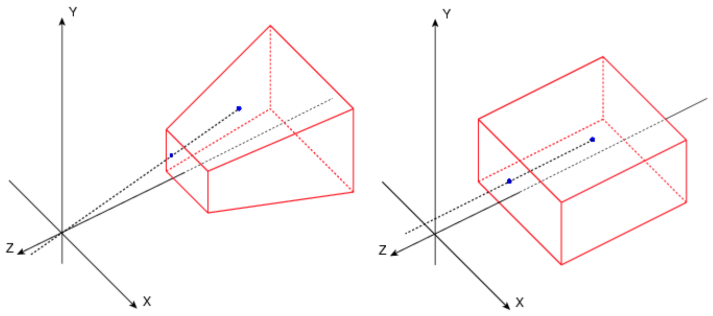


Figure 7: Perspective (left) vs Orthographic (right)

3.5.1 Orthographic

Simple clip to a defined box defined by vertices $(left, bottomnear)$ and $(right, top, far)$.

$$P = \begin{bmatrix} \frac{2}{right-left} & 0 & 0 & -\frac{right+left}{right-left} \\ 0 & \frac{2}{top-bottom} & 0 & -\frac{top+bottom}{top-bottom} \\ 0 & 0 & \frac{2}{far-near} & -\frac{far+near}{far-near} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Typically used for "flat" elements, e.g. menu, HUD, etc.

3.5.2 Perspective

Traditional perspective as perceived in real life.

$$P = \begin{bmatrix} \frac{f}{aspect} & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & \frac{near+far}{near-far} & \frac{2 \cdot near \cdot far}{near-far} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

where:

$$f = \frac{1}{tan(fov/2)}$$

and fov is the desired field of view.

x and y
Used to obtain distance along x and y axis with relation to z and fov

z
Scale and translate z position such that z is in the range -1 to 1 after division by w

w
: Set to $w = -1 \cdot z = -z$, this is used for the perspective divide

3.5.3 Perspective Divide

Have a vector V for a vertex:

$$V = \begin{bmatrix} V_x \\ V_y \\ V_z \\ w \end{bmatrix}$$

Divide components of V by w. This operation moves objects that are further away (in z axis) closer to the centre of the screen, this gives the effect of a vanishing point.

3.6 Object definition

$$\begin{bmatrix} r_{xx} & r_{xy} & r_{xz} & x \\ r_{yx} & r_{yy} & r_{yz} & y \\ r_{zx} & r_{zy} & r_{zz} & z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} V_x \\ V_y \\ V_z \\ 1 \end{bmatrix}$$

(x, y, z) is the object position.

r is the object orientation.

(r_{xx}, r_{xy}, r_{xz}) is the object left vector.

(r_{yx}, r_{yy}, r_{yz}) is the object up vector.

(r_{zx}, r_{zy}, r_{zz}) is the object facing vector.

The vertices of the object V are transformed using the object matrix.

3.7 MVP matrix

Three stages of the standard transformation pipeline:

Model
: Local space to the world space

View
: World space to camera space

Projection
: Camera space to clip space

4 Fragment Operations

4.1 Interpolation

Lines

Colour at point p computed through simple linear interpolation between vertices v_0 and v_1.

$$C_p = (C_b * t) + (C_a * (1 - t))$$
$$t = |(v_1 - p)/(v_1 - v_0)|$$

Triangles

Use Barycentric coordinates (section 2.2.2).

$$C_p = (\alpha * C_a) + (\beta * C_b) + (\gamma * C_c)$$

4.2 Transparency

Transparency denoted by alpha value in colour.

$\alpha = 1$ denotes full opacity, $\alpha = 0$ denotes full transparency.

Colour computed by blend equation:

$$C = (C_{source} * F_{source}) + (C_{dest} * F_{dest})$$

Factors F_{source} and F_{dest} are usually programmable but a common approach is standard linear blending:

$$F_{source} = \alpha$$
$$F_{dest} = 1 - \alpha$$

One other alternative is additive blending:

$$F_{source} = 1$$
$$F_{dest} = 1$$

4.3 Depth Buffer / Depth Test

Have a depth buffer which records the depth (z coordinate) of the fragment that has been rendered on a each pixel.

1 When a pixel is to be shaded compare the z coordinate of the new fragment with that in the depth buffer D_i

2.1 If $x \leq D_i$ then depth test passes, the pixel is shaded based on the new fragment and the depth buffer updated

2.1 Otherwise the test fails and the fragment is discarded

3 The depth buffer is rest to maximum depth at the start of each frame

Can have "z fighting" when two objects with close z coordinates are rasterised inside each other. A higher precision depth buffer avoids this.

5 Texture Mapping

- Texture coordinates (u, v) defined per vertex
- Coordinated interpolated to obtain per fragment texture coordinates
- (u, v) are normalised texture coordinates within $[0, 1]$
- Textures coordinates out of the $[0, 1]$ can be handled differently:

  Clamp
  Anything above 1 is set to 1
  Anything below 0 is set to 0

  Repeat
  $1.1 = 0.1$
  $-0.1 = 0.9$
  etc.

  Mirror
  $1.1 = 0.9$
  $-0.1 = 0.1$
  etc.

- All textures for a mesh typically stored in a single texture image

5.1 Affine Transform

Textures may not appear correctly if an object is tilted with respect to the camera.

Caused by texture interpolation being linear but not fragment area.

Solution is to use affine transform:

1 Divide texture coordinates by P_w

2 Interpolate texture coordinates

3 Multiply by P_w

5.2 Bilinear Filtering

When a texture is viewed close enough to the camera such that the rasterised object takes up more pixel space than the texture image.

Sample multiple texels and blend them together.

e.g. for texel coordinate 7.6 blend colour of texel 7 and 8 by factor 0.6.

5.3 MIP mapping / minification

Generating smaller textures using the original fill size texture so that objects further away can sample a smaller texture.

Forms a set of textures in decrecing size, known as a MIP chain.

MIP map is selected using the level of detail (LOD) λ.

This is calculated using the derivatives of the interpolated x and y texture coordinates, i.e. how fast the texture coordinates are changing.

Faster change in texture coordinates (higher λ) means less unique texels hence less detail. The size of λ denotes how far down the MIP change the texture is selected.

MIP chain is pre processed when the texture is loaded.

Requires more memory to store entire MIP chain opposed to a single texture, but gives faster processing as less work needs to be done during rasterisation and gives a better texture quality due to texel averaging.

5.3.1 Trilinear filtering

Similar to bilinear filtering (operating in x and y axes) but also operating in z axis to interpolate between two MIP map levels.

Solves issue when an object spans multiple MIP levels and a noticeable line where the texture quality changes can be seen.

6 OpenGL

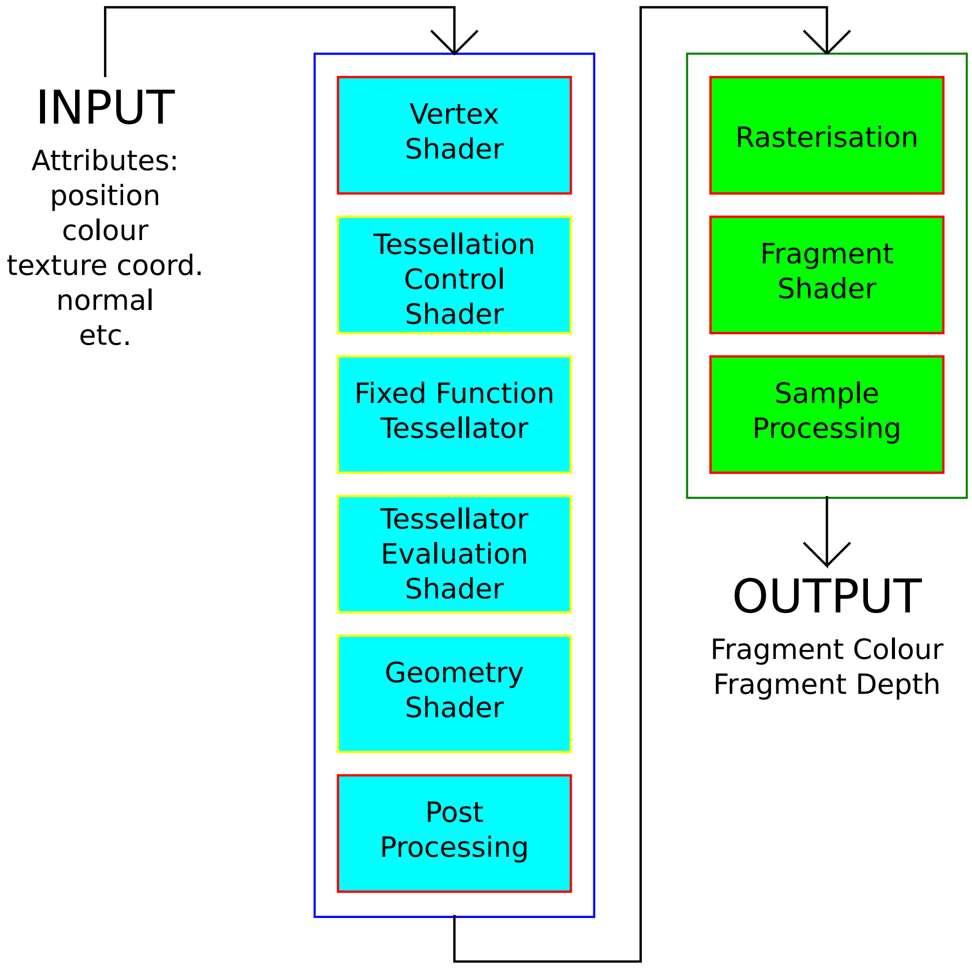


Figure 8: OpenGL Pipeline

Vertex post processing operations:

- Vertex clipping
- Perspective divide
- Viewport transform

Sample processing operations:

- Depth test
- Alpha blending

Note that tessellation shaders, the tessellator, and geometry shaders are optional.

6.1 Shaders

Attributes
Values specific to the vertex or fragment being processed

Uniforms
Constant for all shader executions

Interface block
Used to interface shaders to each other, e.g. vertex shader → fragment shader

Automatic interpolation between shaders (e.g. values in output block of vertex/geometry shaders get interpolated before the fragment shader) (can be disabled using `flat` layout qualifier).

Textures accessed from any shader through samplers. Textures are bound to a texture unit which maps to texturing hardware.

6.2 Geometry Shader

Used to create new primitives from those created by the CPU side code.

- Invoked once per primitive
- Input is array of vertices
- Output is a number of new primitives

Usage examples:

- Normal visualisation
- Particle systems

Reduces processing to be done by the CPU and earlier stages of the pipeline by reducing the number of vertices they need to process.

Limited number of output vertices (based on specific hardware).

6.3 Tessellation

- Used for larger scale geometry amplification
- Tessellation used to ”fill” an existing primitive with more primitives
- More vertices generated which can be transformed

The hardware tessellator operates on patches (areas bound by a number of vertices) and turns them in to wither lines, triangles or quads.

Instead of a position, output vertices have a weighting which specifies its position relative to a number of the input vertices.

Patches have tessellation factors that define how many new new vertices are generated. One factor for the inside of the patch and several for the outside.

Using these factors it is possible to correctly line up patches with differing tessellation levels without having ”cracks” in the object.

6.3.1 Tessellation Control Shader

Invoked once per input patch vertex.

Feeds vertex information to tessellator and controls tessellation levels.

6.3.2 Tessellation Evaluation Shader

Invoked for every new vertex.

Converts barycentric weightings created by the tessellator into positions.

7 Scene Hierarchy and Skeletal animation

7.1 Scene Graphs

- Hierarchical tree structure of meshes
- Each mesh has a model matrix that gets applied to it and all its children
- Typically use a tree as shallow as possible to reduce traversal time
- Can include many other (non graphical) objects on the tree:
  - Sound emitters
  - Shaders
  - etc.

Need to ensure transparent objects are drawn in the correct order. Solution is to add a ”transparency” tag to each node.

When processing objects:

1 Traverse the tree and build a list of opaque objects and a list of transparent objects

2 Render all the opaque objects

3 Sort the list of transparent objects by their z position

4 Render transparent object from furthest away to closest

7.2 Animation

Can do simple animation by manipulating a tree of objects and their model matrices. This works well for simple objects such as cars and robots.

It will not work for objects that have a flexible skin (such as humans).

7.2.1 Skinned meshes

Instead a skinned mesh is used and each vertex is pulled in the direction of several skeletal nodes (joints) by a set of weights.

Skeletal nodes are arranged in a hierarchical tree and inherit transformations.

Skinning (the process of transforming vertices of the mesh based on weights) is typically done on the vertex shader. An array of transformations can be passed to the shader through a uniform.

The assignment of weights to each vertex and node (rigging) is done offline in the 3D modelling suite.

In order to get a good frame rate in a skeletal animation without having to reskin the mesh every frame animation can be interpolated.

In order to combine multiple animations that affect different parts of the skeleton, different animations can be blended together to form a new animation.

Joints can usually be queried to obtain the transformation, useful when attaching other meshes to them (e.g. attach a gun to a hand).

Inverse kinematics can be used to position a child node in a given position and have the parent nodes move in a realistic manner (e.g. placing a hand on a door handle).

8 Lighting

8.1 Normals

- Unit vector perpendicular to the surface
- Can be calculated using cross product of two side vectors
- Usually stored as part of the model
- Vertex normals are interpolated across the primitive
- Interpolation may cause problems if an object has sharp corners

8.2 Lighting Models

Static lighting
Combining texture with a light map.
No real time updates.
No additional computation.

Flat shading
Per surface lighting.
Single value used on a surface.
Computationally fast.

Gourard shading
Per vertex shading.
Interpolated across primitive.
More computationally expensive.

Phong shading
Per fragment lighting.
Most computationally intensive.

Bump Mapping
Normals stored in texture.
Each texel has its own normal.

8.3 Phong Reflection Model

Types of light:

Ambient
Lights all faces of all objects in a scene equally

Diffuse
Light from a source that has been scattered evenly

Specular
Light from a source that has been reflected towards the camera

Lighting colour c of a fragment (for a single light):

$$\begin{aligned} c &= c_a + (c_d + c_s) \times a \\ c_d &= (N \cdot |(L - P)|) \times C_d \\ c_s &= (N \cdot \frac{1}{2}(V + L))^n \times C_s \\ a &= 1 - \frac{L}{L_{max}} \end{aligned}$$

where:

- k_a is the constant ambient light for the scene
- L_{max} is the maximum distance that a light source can be away from a source
- L is the distance from the light source to the surface
- n is the specular power (higher for shinier materials)
- C_d and C_s are colours of the diffuse and specular light
- P is the fragment position
- V is the view vector
- L is the light vector
- N is the normal

Normal N can be calculated using two side vectors: $N = (v_0 - v_1) \times (v_0 - v_2)$
Attenuation factor a ensures that the light gets weaker as the light source moves away from the surface.