Contents

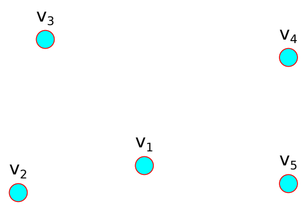
Course material: `https://research.ncl.ac.uk/game/mastersdegree/graphicsforgames/`

1 Overview

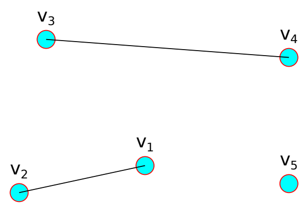
1.1 3D Graphics

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

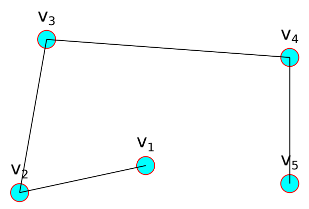
1.1.1 Primitives



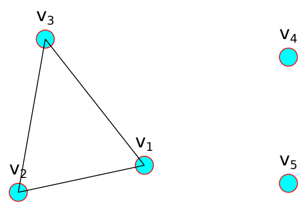
(a) Point



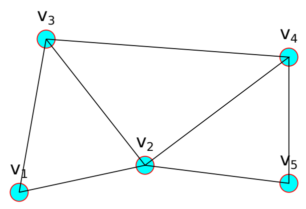
(b) Line



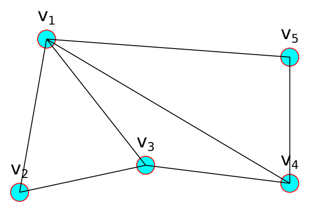
(c) Line Strip



(d) Triangle



(e) Triangle Strip



(f) Triangle Fan

Figure 1

1.2 Graphics Pipeline

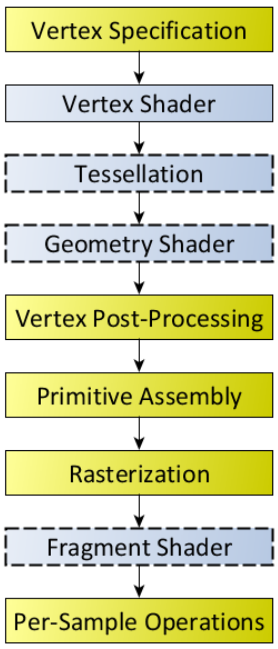


Figure 2: OpenGL Pipeline

1.3 OpenGL

Useful (somewhat) OpenGL related stuff to know:

- Buffers
  - OpenGL buffers are generated from data in main memory
  - Buffer generation copies data to graphics memory
  - Data in main memory is no longer required
  - OpenGL buffers can be bound to CPU memory addresses to modify buffer contents
- Shaders
  - Operate per item (per vertex for vertex and tessellation shaders, per primitive for geometry shader, per fragment for fragment shader)
  - Data common to all shaders passed as uniforms

2 Transformations

2.1 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

2.2 Scale

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

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

2.3 Translation

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

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

2.4 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:

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

2.5 Perspective

Perspective matrix used to give perspective to camera space.

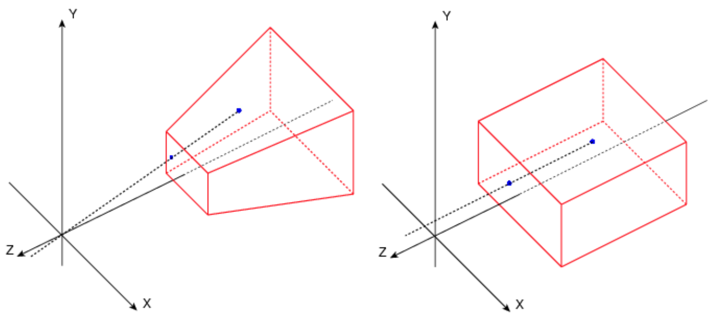


Figure 3: Perspective (left) vs Orthographic (right)

2.5.1 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.

2.5.2 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

2.5.3 Perspective Divide

`https://www.khronos.org/opengl/wiki/Vertex_Post-Processing#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.

2.6 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.

2.7 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

3 Vertex/Geometry Operation

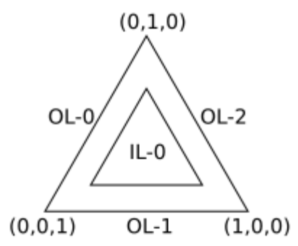
3.1 Amplification (Tessellation Shader)

`https://www.khronos.org/opengl/wiki/Tessellation`

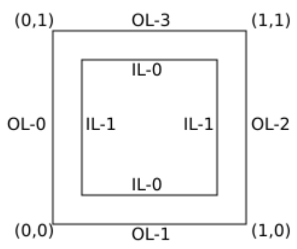
- Tessellator runs after vertex shader
- Operates on patch primitives (generic n-sided primitive)
- Outputs tessellated geometry in either: triangles, quads or isolines
- Tessellation Control Shader is used to set tessellation level and perform any required patch transformations

  (Tessellation Control Shader is optional)
- Tessellation Evaluation Shader takes abstract coordinates generated by the tessellator and the patch coordinates and calculates the vertices of the output geometry

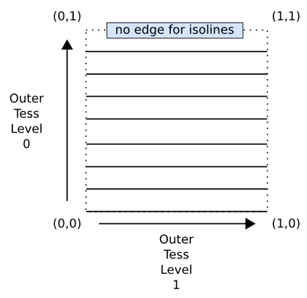
  (Tessellation Evaluation Shader is mandatory)
- Level of tessellation is set for inner and outer regions of tessellated geometry
  - Inner level controls tessellation inside the area
  - Outer level controls tessellation outside the area
  - Edges that are shared between two tessellated areas must share the same outer tessellation level and vertex calculation to ensure continuity between areas



(a) Triangle



(b) Quad



(c) Isoline

Figure 4: Tessellation levels

3.2 Generation (Geometry Shader)

`https://www.khronos.org/opengl/wiki/Geometry_Shader`

- Geometry shader runs after tessellator
- Takes single primitive as input and outputs zero or more new primitives (limited types of primitives supported: points, lines & triangles)
- Used to generate new (additional) geometry based on (limited) existing geometry
- e.g. basic graphical particle system in which points are sent to the GPU pipeline and are converted into quads by the geometry shader

  This saves (75%) CPU time copying primitives to the GPU

3.3 Clipping

`https://www.khronos.org/opengl/wiki/Vertex_Post-Processing#Clipping`

- Remove geometry outside of viewing volume, reduces GPU load in rasterising (parts of) primitives that will never be seen
- Whole primitives that are outside viewing volume are simply culled as a whole
- Primitives that are partially outside are split at the boundary of the viewing volume and new primitives are created as needed to fill any space created inside the viewing volume (e.g. in the case of triangle clipping)

3.4 Face Culling

`https://www.khronos.org/opengl/wiki/Face_Culling`

- Removes (culls) faces that will not be seen due to facing away from the camera
- Can cull front, back or all faces

  Culling all faces effectively disables the rasteriser for triangle based primitives
- Front and back faces are determined by the winding order of the vertices of triangles

4 Fragment Operations

https://www.khronos.org/opengl/wiki/Per-Sample_Processing

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.

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

4.2 Scissor testing

https://www.khronos.org/opengl/wiki/Scissor_Test

- Fragments outside of a given rectangle on the screen are discarded
- Can set unique scissor box per viewport

4.3 Stencil testing

https://www.khronos.org/opengl/wiki/Stencil_Test

- Uses additional buffer attached to active framebuffer
- Each fragment has a stencil value
- Stencil value of a given fragment is tested against value in stencil buffer, if the test passes the fragment is used
- Operation on stencil buffer can be configured based on the result of the stencil and depth tests: keep current value, set to zero, invert current value, replace current value, increment, decrement, increment with wrap, decrement with wrap.
- Test is configurable: always, never, $=$, $\neq$, $<$, $\leq$, $>$, $\geq$
- Initial fragment stencil value is given for all fragments when configuring the stencil test
- Stencil testing can be performed on either front or back faces of a primitive, or both (state is held for both faces)

4.4 Depth Buffer / Depth Test

https://www.khronos.org/opengl/wiki/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.

4.5 Blending

`https://www.khronos.org/opengl/wiki/Blending`

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:

$$\begin{aligned} F_{source} &= \alpha \\ F_{dest} &= 1 - \alpha \end{aligned}$$

One other alternative is additive blending:

$$\begin{aligned} F_{source} &= 1 \\ F_{dest} &= 1 \end{aligned}$$

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 Scene Management and Hierarchy

6.1 Handling Transparent Objects

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

6.2 Early Z test

- Sorting nodes by depth can also be used to make optimal use of the early Z test
- Early Z test happens before the fragment shader is executed
- If opaque objects are drawn from front to back then occluded objects will fail the early Z test and the fragment shader will not run for occluded fragments
- (Hardware specific feature so is difficult to explicit enable or disable it or detect if it is taking place)

6.3 Frustum Culling

- Discard entire objects if they are not in the viewing frustum of the camera
- Frustum represented as 6 infinite planes
- Each object must have a bounding volume, this should be a rough estimation that can be quickly tested for interface with a plane

  Common shapes include spheres and axis aligned bounding boxes
- (there was plane equation stuff in the notes here, it is covered in CSC3503 notes in more detail so look there)
- Frustum is derived from the projection view matrix
- Extract each axis from VP matrix and add/subtract from the w axis to obtain plane normals (which then need to be normalised)

6.4 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.

7 Animation

- Can do simple animation by manipulating a tree of objects and their model matrices
- Animation defined in several keyframes and transformations interpolated between to obtain smooth animation
- Can combine/blend several animations that affect different parts of a model tree

7.1 Skinned meshes

- Model has internal skeleton made up of several joints which can be positioned and rotated relative to each other
- Use a mesh (skin) for a subsection of (or entire) skeleton and set position of vertices on the mesh using weightings based on joint positions
- Two variants:
  - Each skin vertex has a position and a set of joints it is influenced by
  - Each vertex is relative to a number of anchors
    Each anchor has a position and a joint it is relative to
- Final position of a vertex of the skin is determined by position of all its reference points

  Influence of each reference point is determined by a weightings which are stored with the skin vertex (sum of all weights for a single vertex equals 1)
- $position = \sum (t \cdot v) \cdot w$

  t Joint transform

  v Vertex or weight relative position

  w Weight
- Process of assigning weights to the skin (”rigging”) is performed offline by the artist
- Skinning can be performed on the GPU
  - Skeletal transformation data can be sent as a large uniform array or sampled from a texture
  - Vertices require additional attributes, e.g. anchor positions, weightings, joint indices
  - Must enforce a maximum number of joints that a single vertex can be influenced by due to constant size constraint on shader interface blocks
  - Limit on the amount of uniform data that can be sent to shader pipeline, this limits the number of joints in a hardware skinned skeleton

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
- Cannot simply transform normals

8.1.1 Normal Matrix

- Use the normal matrix to transform normals
- Normal matrix is the transposition of the inverse model matrix
  (`mat3 normalMatrix = transpose(inverse(mat3(modelMatrix)))`)
- Preserves rotation while discarding scaling

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.

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.

8.4 Bump mapping

- Adds additional surface texture/bump detail to Phong lighting model
- Surface texture stored in bump map (texture) in tangent space
  Must be transformed to world space before used in lighting calculations
- Perform translation using binormal (see figure 5)
  $B = cross(N, T)$
- Transformation is performed using tangent binormal normal matrix

$$
\begin{bmatrix} T_x & B_x & N_x \\ T_y & B_y & N_y \\ T_z & B_z & N_z \end{bmatrix}
$$

$N_w = TBN \cdot N$

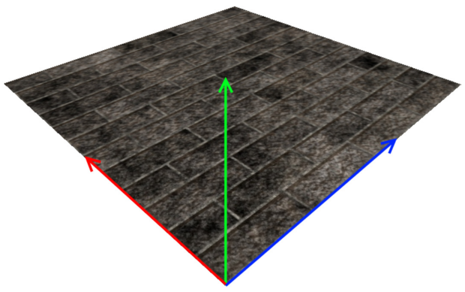


Figure 5: Binormal calculation
Green = Normal, Blue = Tangent, Red = Binormal

8.5 Shadow mapping

- First pass
  - Obtain a shadow map for each light that can cast a shadow.
    1 map for unidirectional lights, 6 maps for omnidirectional lights.
  - Maps are generated by rendering the scene from the position of the light in the relevant direction.
    The map is the depth buffer.
- Second pass
  - Scene rendered from point of view of camera
  - Shadow matrix is passed to shaders, this is the MVP matrix used for light map in first pass
  - Shadow map used to determine position of each fragment from perspective of the light
  - In fragment shader compare the z component of the fragment to the value from the shadow buffer.
    If $z \leq \mathit{light\ depth}$ then the fragment is in front of the closest object in the shadow map, therefore is lit by the light.a
    if $z > \mathit{light\ depth}$ then the fragment is behind the nearest object to the light so is in shadow.
- Shadow map sampling
  - Shadow map is sampled similar to other textures, however a specialised sampler is used that takes a 4 component vector and returns the result of the shadow depth test
  - A bias matrix (uniform scale by 0.5, uniform translation by 0.5) is used to convert from clip space coordinates (-1 to 1) to texture coordinates (0 to 1)

8.5.1 Issues

Surface Acne

- Depth buffer is not high accuracy, especially for objects far from the camera position (or objects far from the light in a shadow map)
- Can cause issues when occluding objects and the surface they occlude are both visible by the camera
- In this case additional shadows may be rendered due to error in the depth values of the occluding and visible surfaces

- One solution is to offset the vertices used in the shadow map depth comparison, either towards the camera or along their normal

  This increases the difference between the two values and reduces the likelihood of error accumulation affecting the result of the shadow depth test

Aliasing

- Low resolution of depth buffer for far geometry can cause poor resolution shadows
- Area of geometry far from camera/light is "covered" by fewer samples than the same sized area near to the camera/light
- Commonly an issue when an object is far from a light but close to the camera
- Simplest solution is to increase resolution of shadow map

8.6 Deferred Rendering

- In the conventional approach to lighting every fragment is processed by the lighting calculation even if they are not visible in the scene

  When many lights are present in the scene this can add up to a considerable amount of processing overhead processing fragments that have no influence on the final rendered scene

  This is the same with objects that are not in range of a given light
- Deferred rendering allows lighting calculations to be run in image space (the final rendered image), this means that only fragments that are visible in the final rendered scene will have lighting calculations applied to them
- Requires splitting data up into several screen sized buffers containing information used in lighting calculations

  This is known as the G-buffer

  Possible contents:

  – Diffuse colours
  – Normals
  – Depth
  – Specular intensity
  – Post processing flags
  – (any per fragment data)
- Lighting calculations are only performed within the volume of a given light, removes calculations for fragments that are outside of a given lights reach
- In the second pass it is already known what fragments are in the final image (from the G-buffer) therefore no calculations take place for fragments that do not contribute to the final image
- Video memory and bandwidth can be an issue due to the large amount of space taken by the G-buffer
- Still does not help with issue of lighting transparent objects
- May need to perform anti-aliasing on final scene to remove jaggedness from edges

8.6.1 Light volumes

- In second pass volumes are rendered for each light
- Light volume encapsulates area which is lit by the light
- For example, for a simple point light the volume is simple a sphere, for a spotlight it could be a cone
- In view space a light volume could encapsulate an object that is in reality far away from the light

  For this reason depth testing in the third stage must be performed

8.6.2 Workflow

1 Fill G-Buffer

  - Very similar to conventional rendering process
  - Transform vertex positions by MVP matrix
  - Sample diffuse colour from texture
  - Fill relevant data in G-buffer

2 Lighting Calculations

  - Render light volumes into scene, marking fragments that may be affected by lights
  - In fragment shader the world position of the fragment is reconstructed using the depth map from the G-buffer

    Can then determine if lighting calculation should be performed or if the fragment should be discarded (if outside light volume)
  - Additive blending is used to combine contributions from multiple lights

3 Generate Final Image

  - Draw a screen sized quad
  - Texture using blend of lighting attachment textures and diffuse texture (from G-buffer)

9 Post processing

- Scene is rendered into a framebuffer to allow post processing stages to manipulate the rendered scene as a texture
- Textures are bound to framebuffers as attachments that can contain colour, depth and stencil data
- Once scene has been rendered into framebuffer the attachments can be manipulated in further pipeline stages as regular textures
- Textures used as framebuffer attachments cannot be compressed (compression/decompression overhead would be undesirable anyway) but can be packed (e.g. using a single texture for depth and stencil attachments)
- A processing stage renders a single quad that fills the screen which is then textured using the colour attachment texture from the framebuffer the scene was rendered into
- Processing stages may be cascaded using multiple processing render passes into multiple framebuffers

  An alternative to this is "ping-pong" texturing, in which a single framebuffer and two textures are used and each processing stage samples from one texture and buffers to the other alternately
- As the scene render pass is available as a texture effects that depend on multiple fragments (e.g. blur) are now possible

  Such effects are not possible without this post processing stage as in the standard pipeline the fragment shader does not have access to adjacent (or any) fragments other than the one it is operating on