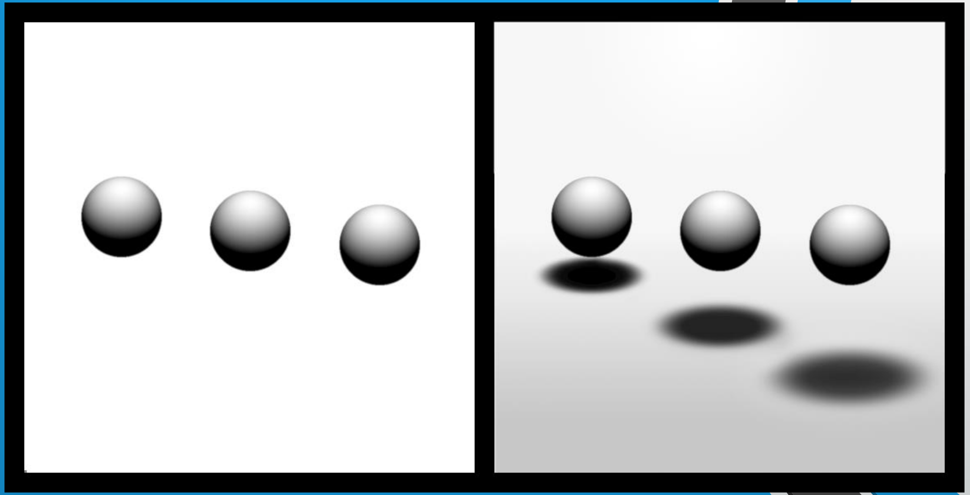
Shadows

in OpenGL

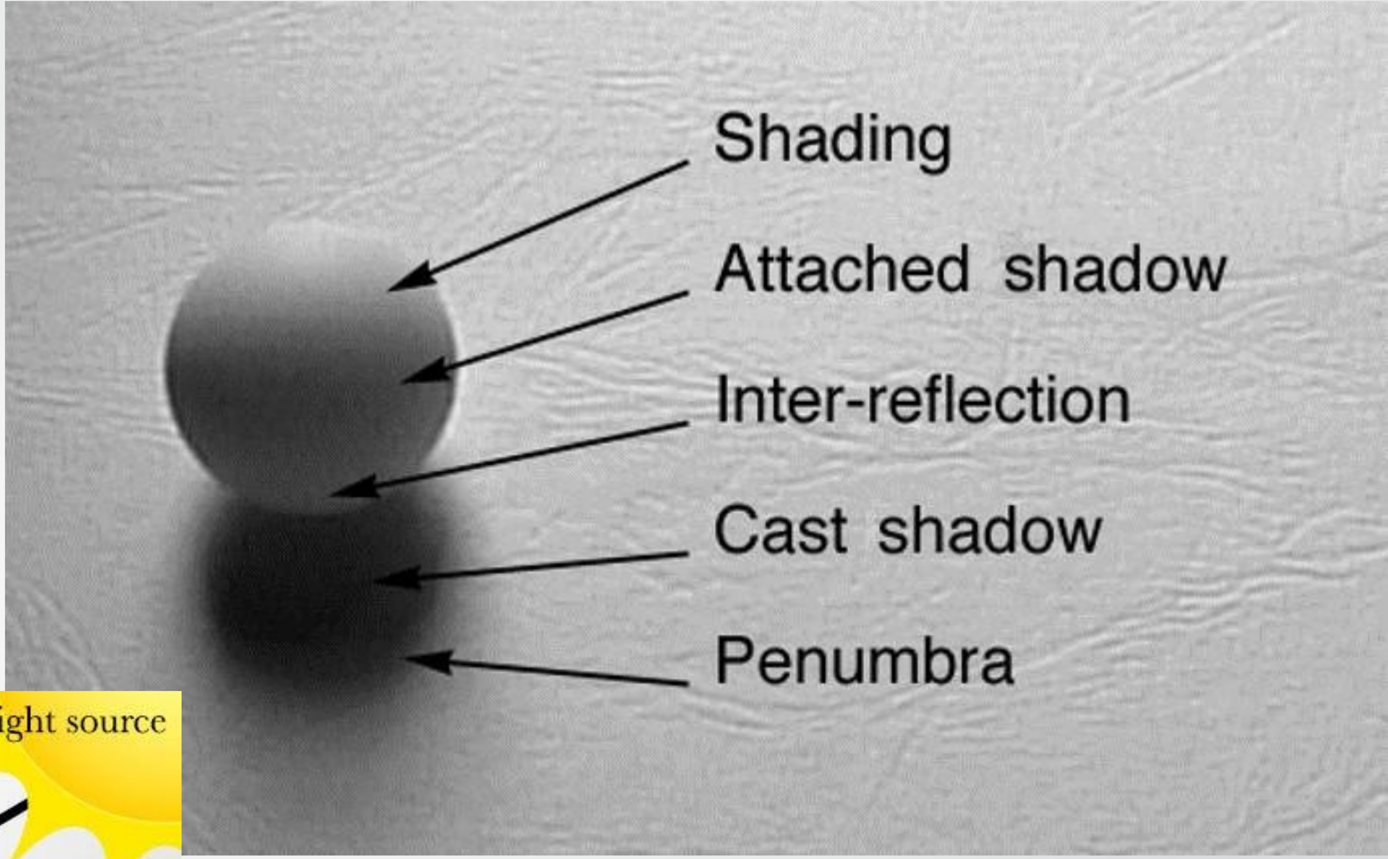
Introduction

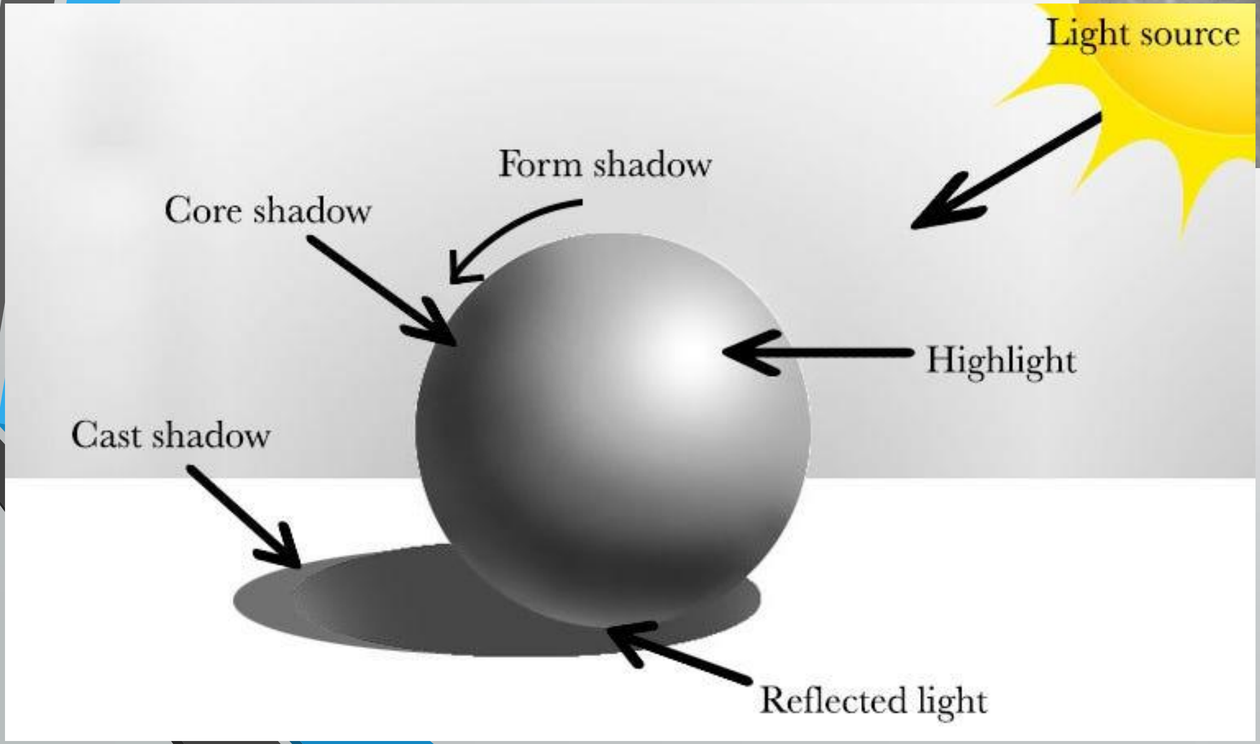
Shadows

Introduction



- Shadows bring realism to a scene, helping with the relative locations in the scene.
- Shadows are also important to give the observer visual cues – lighting, realism, contact points
- Shadows also have a great contribution to the depth cues, helps us understand a complex 3 dimensional environment
- We'll look into **shadow mapping** algorithm (**depth shadows**) and a couple of ways to improve it
- We'll look into **shadow volumes**.


Shading
Attached shadow
Inter-reflection
Cast shadow
Penumbra


Light source
Form shadow
Core shadow
Highlight
Cast shadow
Reflected light

Shadow maps

in a shader

Shadow maps

- One of the most popular algorithm for creating shadows in **shadow mapping**
- The algorithm involves 2 passes.
- First pass:
  - The scene is rendered from the point of view of the light source
  - The depth information from this pass is saved into a texture called **shadow map**.

Shadow maps

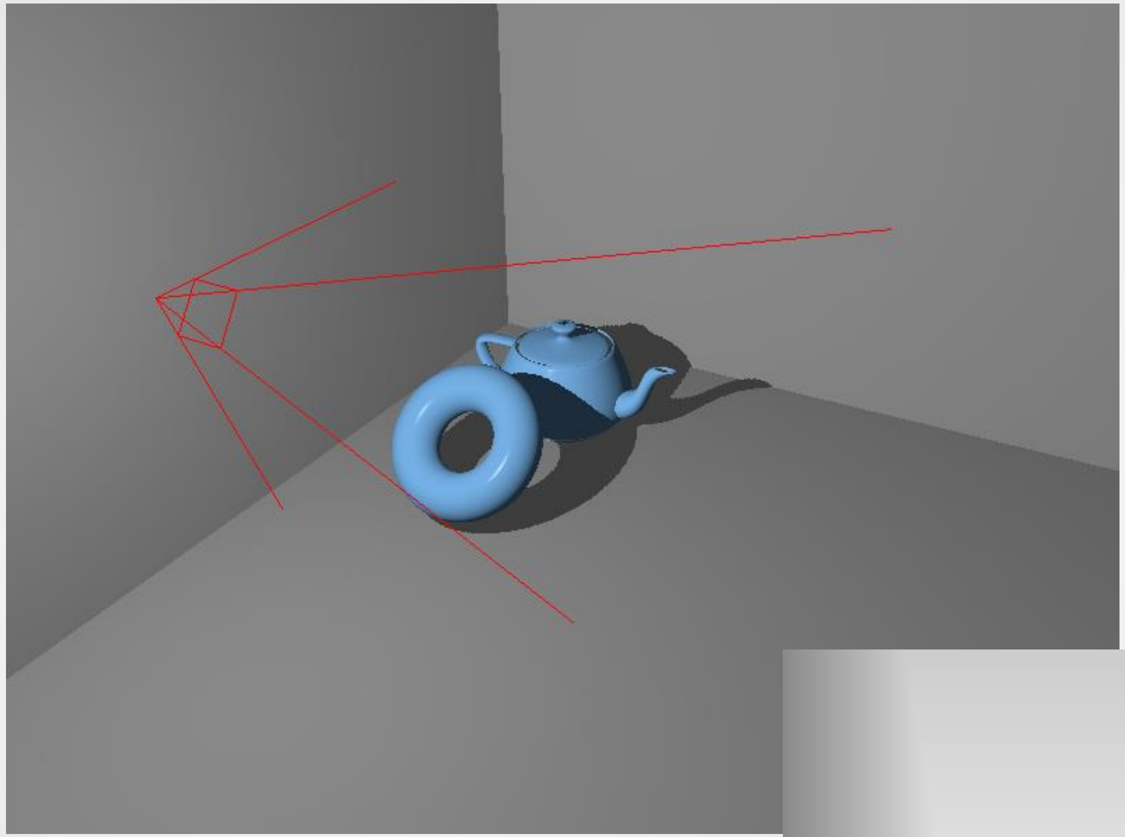
- In the second pass:
  - The scene is rendered normally
  - But each fragment's depth is tested against the shadow map to determine if the fragment is in the shadow or not.
  - The fragment is then shaded differently depending on the result of this test

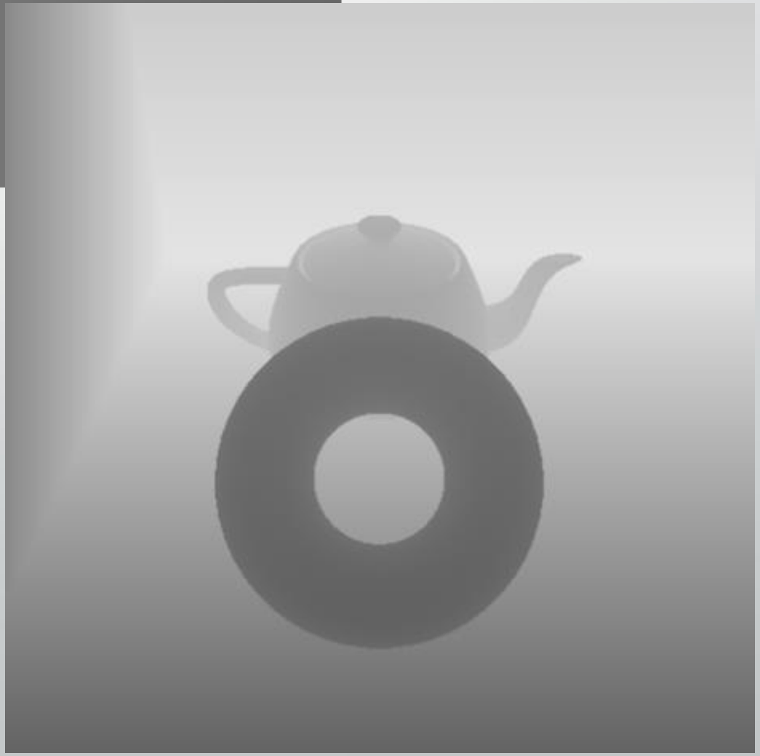
Shadow maps steps

First step – creation of the shadow map

- Set up a view matrix, located at the position of the light source and orientated towards the objects that cast the shadow
- Set up a projection matrix
  - the view frustrum encloses all the objects that might cast shadows
  - Defines the area where shadows will appear
- We then render the scene normally and store the information from the depth buffer in a texture

Shadow maps steps





Shadow maps steps

Second step– rendering of the shadow

- We render the scene again but from the point of view of the camera
- We use a fragment shader that shades each fragment based on the result of a depth test with the shadow map
  - The position of the fragment is first converted into the coordinate system of the light source and projected using the light source's projection matrix.
  - The result is then biased (in order to get valid texture coordinates) and tested against the shadow map.
  - If the depth of the fragment is greater than the depth stored in the shadow map, then there must be some surface that is between the fragment and the light source.
  - The fragment is in shadow and is shaded using **ambient lighting only**. Otherwise, the fragment must have a clear view to the light source, and so it is **shaded normally**.

Shadow maps accessing values

- The **x** and **y** components of the position in clip coordinates are roughly what we need to access the shadow map. The **z** coordinate contains the depth information that we can use to compare with the shadow map.
- Before we can use these values we need to do two things.
  - First, we need to bias them so that they range from zero to one (instead of -1 to 1), and
  - Second, we need to apply **perspective division**

Shadow maps accessing values

- To convert the value from clip coordinates to a range appropriate for use with a shadow map, we need the **x**, **y**, and **z** coordinates to range from zero to one (for points within the light's view frustum).
- The depth that is stored in an OpenGL depth buffer (and also our shadow map) is simply a fixed or floating-point value between zero and one (typically). A value of zero corresponds to the near plane of the perspective frustum, and a value of one corresponds to points on the far plane.
- If we are to use our **z** coordinate to accurately compare with this depth buffer, we need to scale and translate it appropriately.
- We need to do the same thing for the **x** and **y** components as the range for texture access is between 0 and 1

Shadow maps formulas

- For that we use the bias matrix:

$$B = \begin{bmatrix} 0.5 & 0 & 0 & 0.5 \\ 0 & 0.5 & 0 & 0.5 \\ 0 & 0 & 0.5 & 0.5 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

This matrix will scale and translate our coordinates such that the **x**, **y**, and **z** components range from 0 to 1 (after perspective division) for points within the light's frustum.

$$Q = B P_l V_l W$$

Q: shadow map access

B: bias matrix

P_l: projection matrices

V_l: light’s view

W: positions in world coordinates

Shadow maps formulas

- We introduce the shadow matrix:

$$Q = SC$$

S: shadow matrix

C: object coordinates

$$S = BP_lV_lM$$

Q: homogenous coordinates

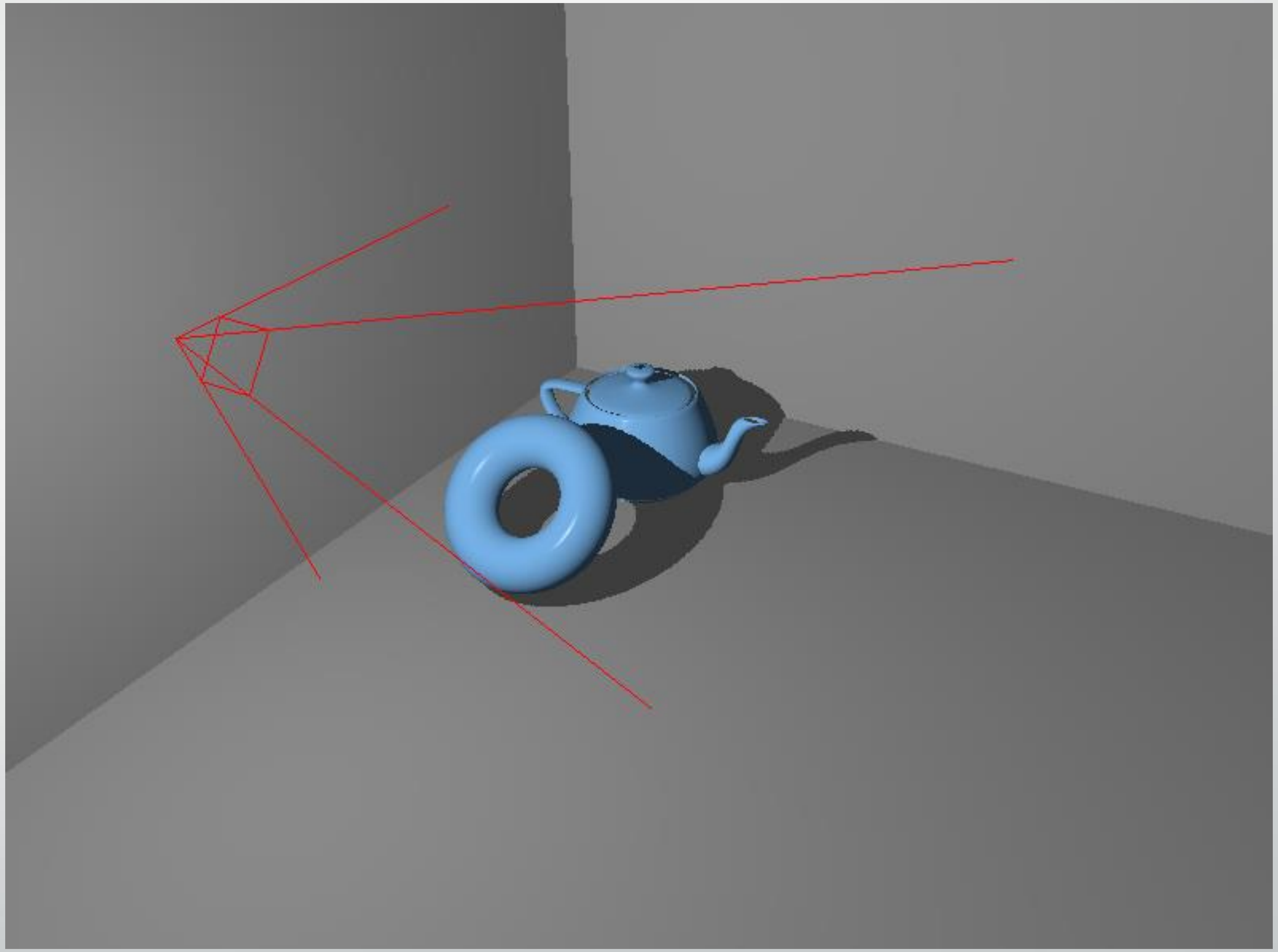
B: bias matrix

P_l: projection matrices

V_l: light's view

M: model matrix

Shadow maps steps



Antialiasing shadows

with PCF

Antialiasing shadows

- One of the most common techniques for dealing with the aliasing of shadow edges is called **percentage-closer filtering** (**PCF**).
- The name comes from the concept of sampling the area around the fragment and determining the percentage of the area that is closer to the light source (in shadow).
- The percentage is then used to scale the amount of shading (diffuse and specular) that the fragment receives. The overall effect is a blurring of the shadow's edges.

Antialiasing shadows

- The basic technique was first published by Reeves et al. in a 1987 paper (**SIGGRAPH Proceedings, Volume 21, Number 4, July 1987**).
- The concept involves transforming the fragment's extents into shadow space, sampling several locations within that region, and computing the percent that is closer than the depth of the fragment.
- The result is then used to attenuate the shading. If the size of this filter region is increased, it can have the effect of blurring the shadow's edges

Antialiasing shadows

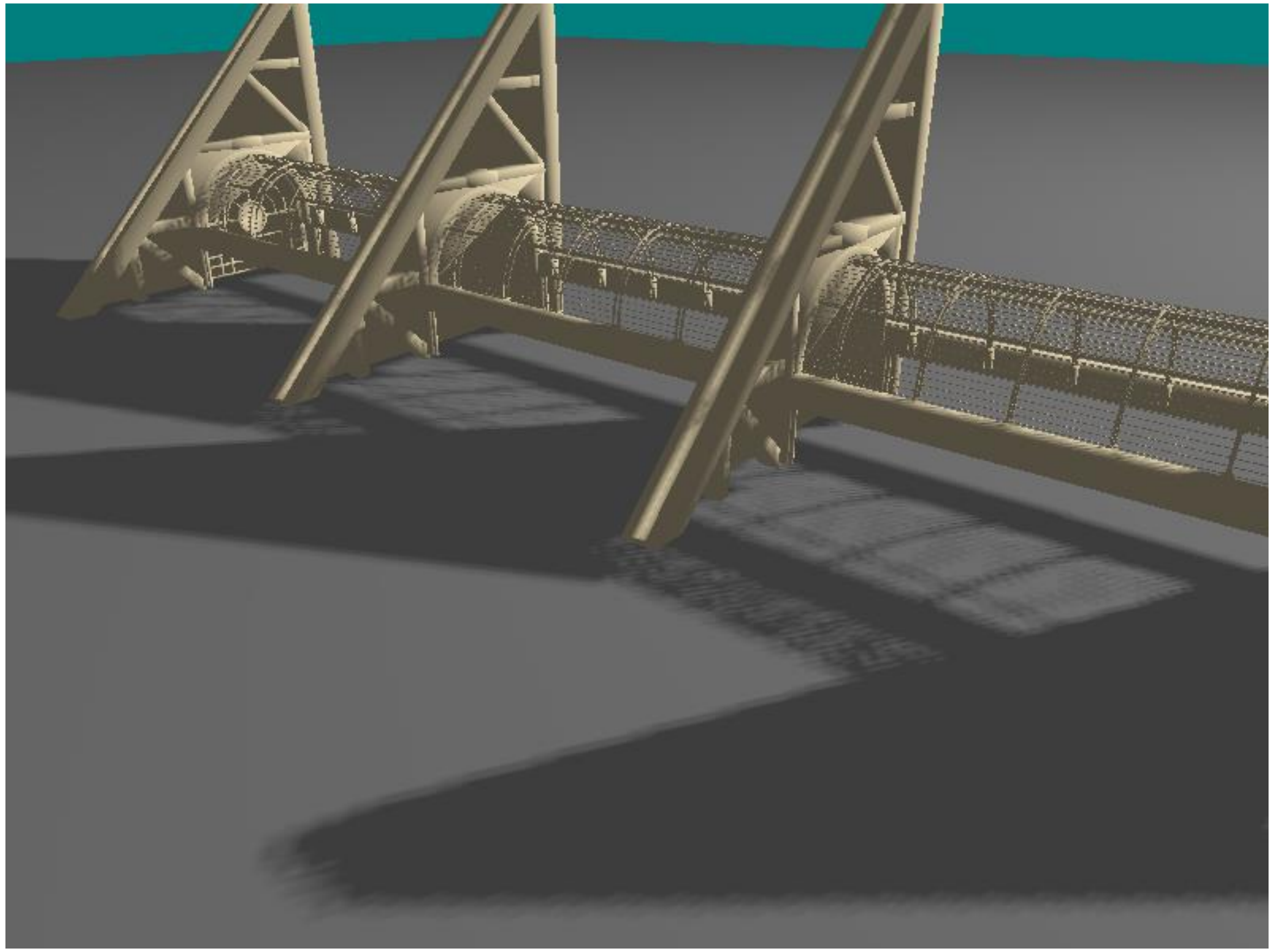
- We’ll use a common variant of the PCF algorithm, which involves just sampling a constant number of nearby texels within the shadow map.
- The percent of those texels that are closer to the light is used to attenuate the shading. This has the effect of blurring the shadow's edges.
- In other words, we'll calculate an average of the resulting comparisons and use that result to scale the diffuse and specular components.

Antialiasing shadows

- We'll make use of OpenGL's built-in support for PCF by using linear filtering on the depth texture.
- When linear filtering is used with this kind of texture, the hardware can automatically sample four nearby texels (execute four depth comparisons) and average the results .Therefore, when linear filtering is enabled, the result of the textureProj function can be somewhere between 0.0 and 1.0.
- OpenGL provides the textureProjOffset texture access function, which has a third parameter (the offset) that is added to the texel coordinates before the lookup/comparison.

Antialiasing shadows

- Within the fragment shader, we use the textureProjOffset function to sample the four texels (diagonally) surrounding the texel nearest to ShadowCoord.
- The third argument is the offset. It is added to the texel's coordinates (not the texture coordinates) before the lookup takes place.
- As linear filtering is enabled, each lookup will sample an additional four texels, for a total of 16 texels. The results are then averaged together and stored within the variable shadow.
- The result is used to attenuate the diffuse and specular components of the lighting model



Antialiasing shadows

Soft shadows

with random sampling

Soft shadows with random sampling

- The technique presented in this recipe is based on a chapter published in **GPU Gems 2**, edited by Matt Pharr and Randima Fernando, Addison-Wesley Professional, 2005. (Chapter 17 by Yury Uralsky.)
- Instead of sampling texels around the fragment's position (in shadow map space) using a constant set of offsets, we use a random, circular pattern of offsets
- In addition, we sample only the outer edges of the circle first in order to determine whether or not the fragment is in an area that is completely inside or outside of the shadow

Soft shadows with random sampling

- Additionally, we vary the sample locations through a set of precomputed sample patterns.
- We compute random sample offsets and store them in a texture prior to rendering. Then, in the fragment shader, the samples are determined by first accessing the offset texture to grab a set of offsets and using them to vary the fragment's position in the shadow map.
- The results are then averaged together in a similar manner to the basic PCF algorithm.

Soft shadows with random sampling

- We'll store the offsets in a three-dimensional texture (**w** x **h** x **d**). The first two dimensions are of arbitrary size, and the third dimension contains the offsets.
- Each (**w**,**h**) location contains a list (size **d**) of random offsets packed into an RGBA colour. Each RGBA colour in the texture contains two 2D offsets.
- The **R** and **G** channels contain the first offset, and the **B** and **A** channels contain the second. Therefore, each (**w**,**h**) location contains a total of **2*d** offsets.

Soft shadows with random sampling

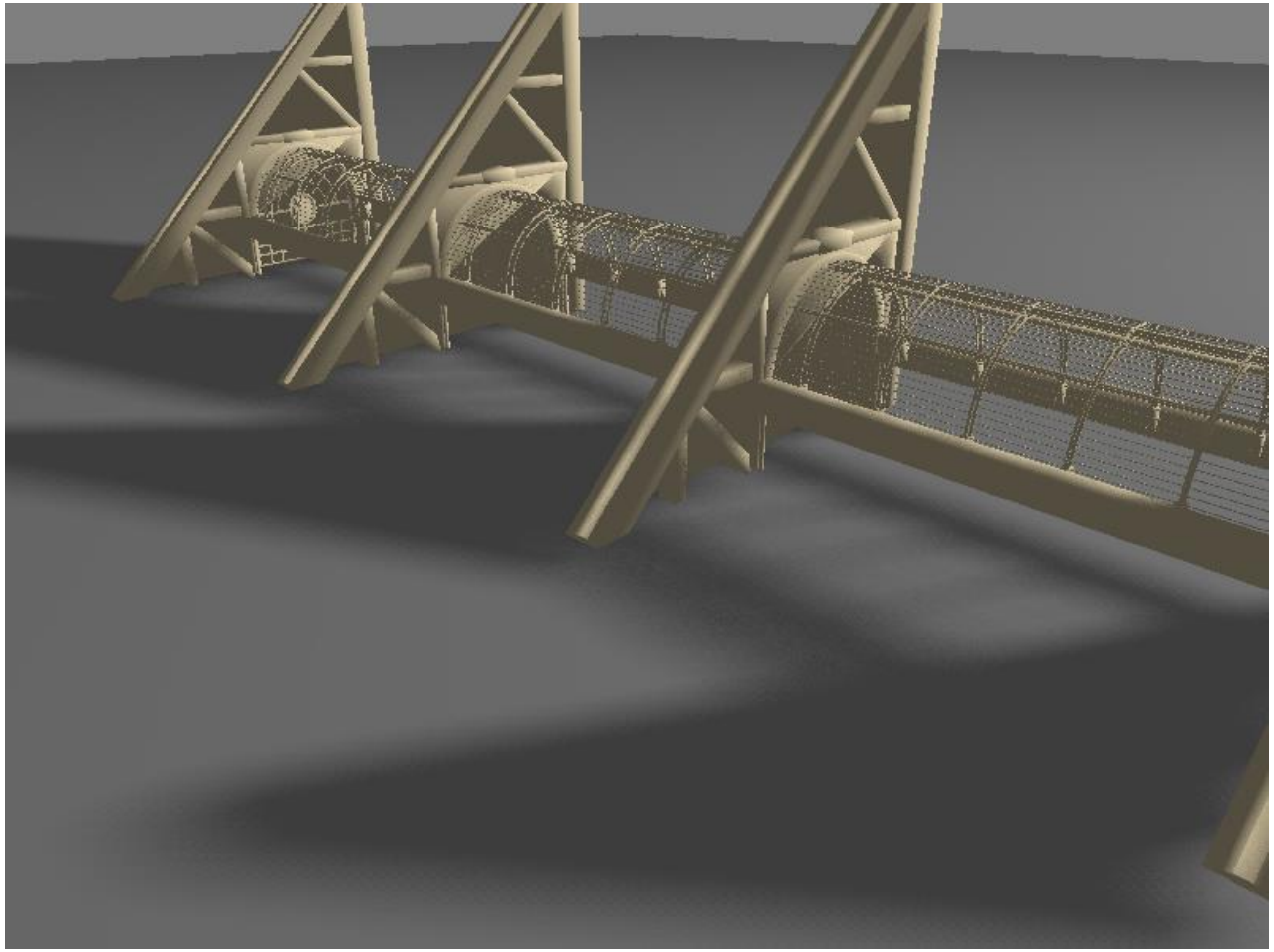
- The offsets are initially defined to be centred on a grid of size samplesU x samplesV.
- The coordinates of the offsets are scaled such that the entire grid fits in the unit cube (side length 1) with the origin in the lower-left corner.
- Then, each sample is randomly jittered from its position to a random location inside the grid cell.
- Finally, the jittered offsets are warped so that they surround the origin and lie within the circular grid shown on the right.

$$w_x = \sqrt{v}\,\cos(2\pi u)$$

$$w_y = \sqrt{v}\,\sin(2\pi u)$$

Soft shadows with random sampling

- **w** is the warped coordinate. What we are left with is a set of offsets around the origin that are a maximum distance of 1.0 from the origin.
- Additionally, we generate the data such that the first samples are the ones around the outer edge of the circle, moving inside toward the centre. This will help us avoid taking too many samples when we are working completely inside or outside of the shadow.
- We also pack the samples in such a way that a single texel contains two samples. This is not strictly necessary, but is done to conserve memory space. However, it does make the code a bit more complex.



Soft shadows with random sampling

Shadow volumes

using the Geometry shader

Shadow volumes

- The shadow volume method completely avoids the aliasing problem that we experienced with shadow maps.
- With shadow volumes, you get pixel-perfect hard shadows, without the aliasing artifacts of shadow maps.
- The shadow volume technique works by making use of the stencil buffer to mask out areas that are in shadow. We do this by drawing the boundaries of the actual shadow volumes.
- A **shadow volume** is the region of space where the light source is occluded by an object.

Shadow volumes

- The boundaries of a shadow volume are made up of quads formed by extending the edges of the object away from the light source.
- For a single triangle, the boundaries would consist of three quads, extended from each edge, and triangular caps on each end.
- One cap is the triangle itself and the other is placed at some distance from the light source.

Shadow volumes

- For an object that consists of many triangles, such as the preceding sphere, the volume can be defined by the so-called **silhouette edges**.
- These are edges that are on or near the boundary between the shadow volume and the portion of the object that is lit. In general, a silhouette edge borders a triangle that faces the light and another triangle that faces away from the light.
- To draw the shadow volume, we find all of the silhouette edges and draw extended quads for each edge. The caps of the volume could be determined by making a closed polygon (or triangle fan) that includes all the points on the silhouette edges, and similarly on the far end of the volume.

Shadow volumes

- In OpenGL, with the stencil buffer, we can increment/decrement a counter for each pixel based on whether a front or back face is rendered into that pixel.
- We can draw the boundaries of all of the shadow volumes, then for each pixel, increment the stencil buffer's counter when a front face is rendered to that pixel and decrement when it is a back face.
- The key here is to realize that each pixel in the rendered figure represents an eye-ray. So, for a given pixel, the value in the stencil buffer is the value that we would get if we actually traced a ray through that pixel. The depth test helps to stop tracing when we reach a surface.

Shadow volumes

- In our example, rather than computing the shadow volumes on the CPU side, we'll render the geometry normally, and have the geometry shader produce the shadow volumes.
- The geometry shader can be provided with adjacency information for each triangle.
- With adjacency information, we can determine whether a triangle has a silhouette edge. If the triangle faces the light, and a neighbouring triangle faces away from the light, then the shared edge can be considered a silhouette edge, and used to create a polygon for the shadow volume.

Shadow volumes

The entire process is done in 3 passes:

1. Render the scene normally, but write the shaded colour to two separate buffers. We'll store the ambient component in one and the diffuse and specular components in another.
2. Set up the stencil buffer so that the stencil test always passes, and front faces cause an increment and back faces cause a decrement. Make the depth buffer read-only, and render only the shadow-casting objects. In this pass, the geometry shader will produce the shadow volumes, and only the shadow volumes will be rendered to the fragment shader.
3. Set up the stencil buffer so that the test succeeds when the value is equal to zero. Draw a screen-filling quad, and combine the values of the two buffers from step one when the stencil test succeeds.

Shadow volumes implementation

- We'll start by creating our buffers. We'll use a framebuffer object with a depth attachment and two colour attachments. The ambient component can be stored in a renderbuffer (as opposed to a texture) because we'll blit (a fast copy) it over to the default framebuffer rather than reading from it as a texture. The **diffuse + specular** component will be stored in a texture.
- Create the ambientBuffer, depthBuffer and a texture for diffuse and specular together

Shadow volumes implementation

```
// The depth buffer
    GLuint depthBuf;
    glGenRenderbuffers(1, &depthBuf);
    glBindRenderbuffer(GL_RENDERBUFFER, depthBuf);
    glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT, width, 
height);

    // The ambient buffer
    GLuint ambBuf;
    glGenRenderbuffers(1, &ambBuf);
    glBindRenderbuffer(GL_RENDERBUFFER, ambBuf);
    glRenderbufferStorage(GL_RENDERBUFFER, GL_RGBA, width, height);

    // The diffuse+specular component
    glActiveTexture(GL_TEXTURE0);
    GLuint diffSpecTex;
    glGenTextures(1, &diffSpecTex);
    glBindTexture(GL_TEXTURE_2D, diffSpecTex);
    glTexStorage2D(GL_TEXTURE_2D, 1, GL_RGBA8, width, height);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);

    // Create and set up the FBO
```

Shadow volumes implementation

- Set up the draw buffers so that we can write to the color attachments:

```
GLenum drawBuffers[] = { GL_COLOR_ATTACHMENT0, GL_COLOR_ATTACHMENT1 };
glDrawBuffers(2, drawBuffers);
```

Shadow volumes implementation

- For the first pass, enable the framebuffer object that we set up earlier, and render the scene normally. In the fragment shader, send the **ambient** component and the **diffuse + specular** component to separate outputs:

```
layout( location = 0 ) out vec4 Ambient;
layout( location = 1 ) out vec4 DiffSpec;

void shade( ) {
// Compute the shading model, and separate out the ambient
// component.
Ambient = ...;   // Ambient
DiffSpec = ...;  // Diffuse + specular
}
void main() { shade(); }
```

Shadow volumes implementation

- In the second pass, we'll render our shadow volumes. We want to set up the stencil buffer so that the test always succeeds, and that front faces cause an increment, and back faces cause a decrement:

```
glClear(GL_STENCIL_BUFFER_BIT);
glEnable(GL_STENCIL_TEST);
glStencilFunc(GL_ALWAYS, 0, 0xffff);
glStencilOpSeparate(GL_FRONT, GL_KEEP, GL_KEEP, GL_INCR_WRAP);
glStencilOpSeparate(GL_BACK, GL_KEEP, GL_KEEP, GL_DECR_WRAP);
```

Shadow volumes implementation

- Also in this pass, we want to use the depth buffer from the first pass, but we want to use the default frame buffer, so we need to copy the depth buffer over from the FBO used in the first pass. We'll also copy over the color data, which should contain the ambient component:

```
glBindFramebuffer(GL_READ_FRAMEBUFFER, colorDepthFBO);
glBindFramebuffer(GL_DRAW_FRAMEBUFFER,0);
glBlitFramebuffer(0,0,width,height,0,0,width,height,
        GL_DEPTH_BUFFER_BIT|GL_COLOR_BUFFER_BIT, GL_NEAREST);
```

Shadow volumes implementation

- We don't want to write to the depth buffer or the color buffer in this pass, since our only goal is to update the stencil buffer, so we'll disable writing for those buffers:

```
glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);
glDepthMask(GL_FALSE);
```

Shadow volumes implementation

- Next, we render the shadow-casting objects with adjacency information. In the geometry shader, we determine the silhouette edges and output only quads that define the shadow volume boundaries:

```
layout( triangles_adjacency ) in;
layout( triangle_strip, max_vertices = 18 ) out;
in vec3 VPosition[];
in vec3 VNormal[];
uniform vec4 LightPosition;  // Light position (eye coords)
uniform mat4 ProjMatrix;     // Proj. matrix (infinite far plane)

bool facesLight( vec3 a, vec3 b, vec3 c ) {…}

void emitEdgeQuad( vec3 a, vec3 b ) { …}

void main() {
if( facesLight(VPosition[0], VPosition[2], VPosition[4]) ) {
if( ! facesLight(VPosition[0],VPosition[1],VPosition[2]) )
…}
```

Shadow volumes implementation

- In the third pass, we'll set up our stencil buffer so that the test passes only when the value in the buffer is equal to zero:

```
glStencilFunc(GL_EQUAL, 0, 0xffff);
glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);
```

- We want to enable blending so that our ambient component is combined with **diffuse +specular** when the stencil test succeeds:

```
glEnable(GL_BLEND);
glBlendFunc(GL_ONE,GL_ONE);
```

Shadow volumes implementation

- In this pass, we just draw a screen-filling quad, and output the **diffuse + specular** value. If the stencil test succeeds, the value will be combined with the ambient component, which is already in the buffer

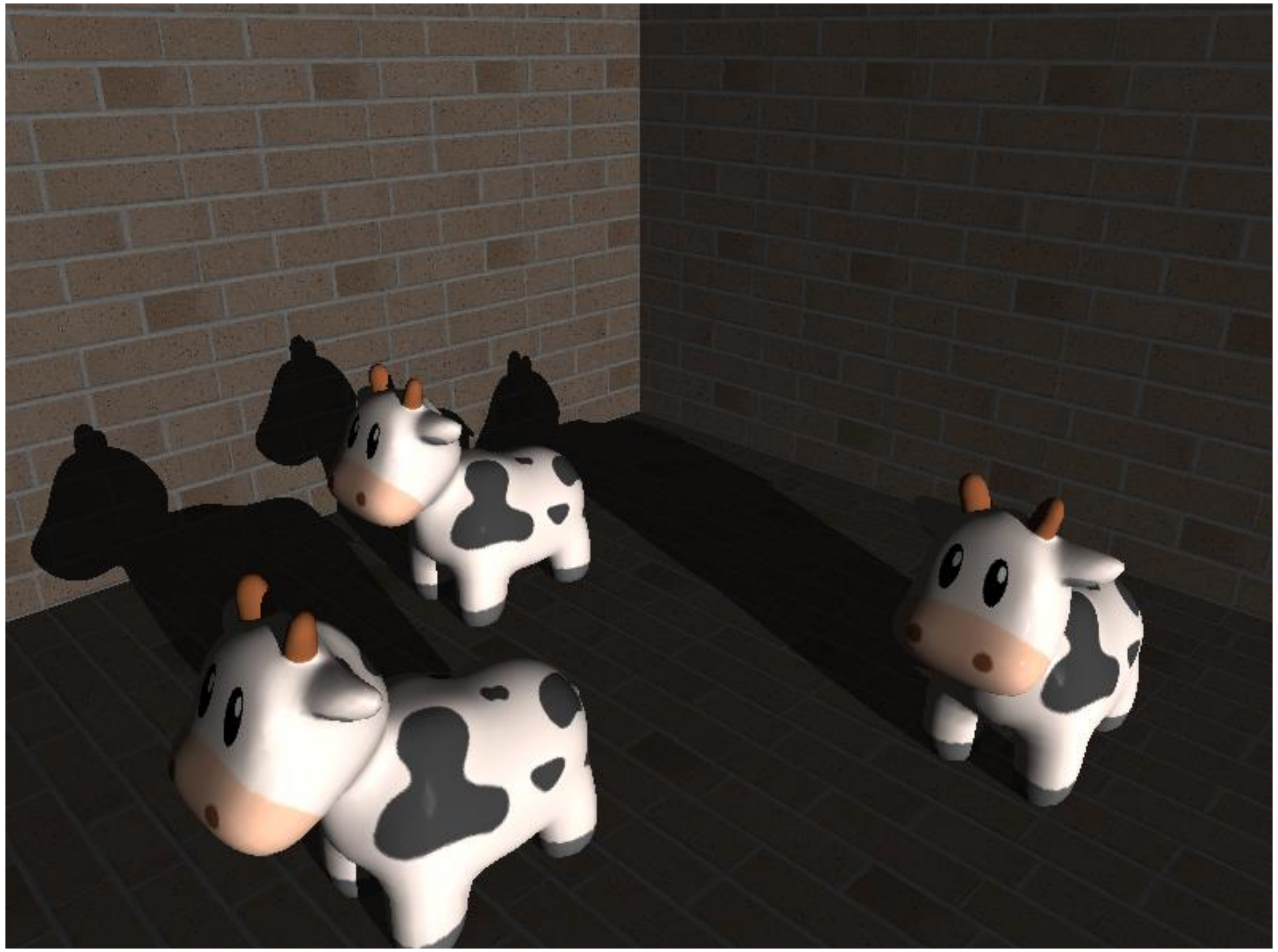
```
layout(binding = 0) uniform sampler2D DiffSpecTex;
layout(location = 0) out vec4 FragColor;

void main() {
        vec4 diffSpec = texelFetch(DiffSpecTex, ivec2(gl_FragCoord), 0);
        FragColor = vec4(diffSpec.xyz, 1);
}
```

Shadow volumes implementation

- The technique described here is often referred to as the **z-pass** technique. One issue, if the camera is located within a shadow volume, this technique breaks down because the counts in the stencil buffer will be off by at least one.
- A common solution is to basically invert the problem and trace a ray from infinity toward the view point. This is called the **z-fail** technique or **Carmack's reverse**.
- With z-fail, it is important to draw the caps of the shadow volumes
- The technique is very similar to z-pass. Instead of incrementing/decrementing when the depth test passes, we do so when the depth test fails. This effectively **traces** a ray from infinity back toward the view point.

Shadow volumes



Shadow Volume vs Shadow Map

Shadow Map	Shadow Volume
Scalable – adjust resolution of shadow maps or change filtering.	Shadows match the shape of the casting mesh
Textured shadows possible.	Can't do accurate soft shadows
Transparent shadows possible.	Can't do shadows on transparent objects

Shadow map is industry standard due to above

Useful links

- Normalized device coordinates and clips space: https://www.youtube.com/watch?v=pThwoS8MR7w&ab_channel=BrianWill
- To read – Chapter 8 Buffer Objects: Storage is now in your hand (OpenGL Superbible – see link on the DLE)
- To read – Shadow mapping: https://learnopengl.com/Advanced-Lighting/Shadows/Shadow-Mapping
- To read: Shadows (OpenGL 4 Shading Language Cookbook).
- glDepthRange: https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glDepthRange.xhtml
- glPolygonOffset: https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glPolygonOffset.xhtml
- Shadow mapping tutorial: http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-16-shadow-mapping/
- How shadow mapping works: https://www.youtube.com/watch?v=EsccgeUpdsM&ab_channel=thebennybox
- Improve shadow mapping (Microsoft): https://docs.microsoft.com/en-gb/windows/win32/dxtecharts/common-techniques-to-improve-shadow-depth-maps?redirectedfrom=MSDN
- textureProj: https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/textureProj.xhtml
- textureProjOffset: https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/textureProjOffset.xhtml
- gl_FragCoord: https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/gl_FragCoord.xhtml
- Stencil buffer: https://subscription.packtpub.com/book/game_development/9781789342253/8/ch08lvl1sec80/creating-shadows-using-shadow-volumes-and-the-geometry-shader