7
Shadows

In this chapter, we will cover:

- Rendering shadows with shadow maps
- Anti-aliasing shadow edges with PCF
- Creating soft shadow edges with random sampling
- Creating shadows using shadow volumes and the geometry shader

Introduction

Shadows add a great deal of realism to a scene. Without shadows, it can be easy to misjudge the relative location of objects, and the lighting can appear unrealistic, as light rays seem to pass right through objects.

Shadows are important visual cues for realistic scenes, but can be challenging to produce in an efficient manner in interactive applications. One of the most popular techniques for creating shadows in real-time graphics is the **shadow mapping** algorithm (also called depth shadows). In this chapter, we'll look at several recipes surrounding the shadow mapping algorithm. We'll start with the basic algorithm, and discuss it in detail in the first recipe. Then we'll look at a couple of techniques for improving the look of the shadows produced by the basic algorithm.

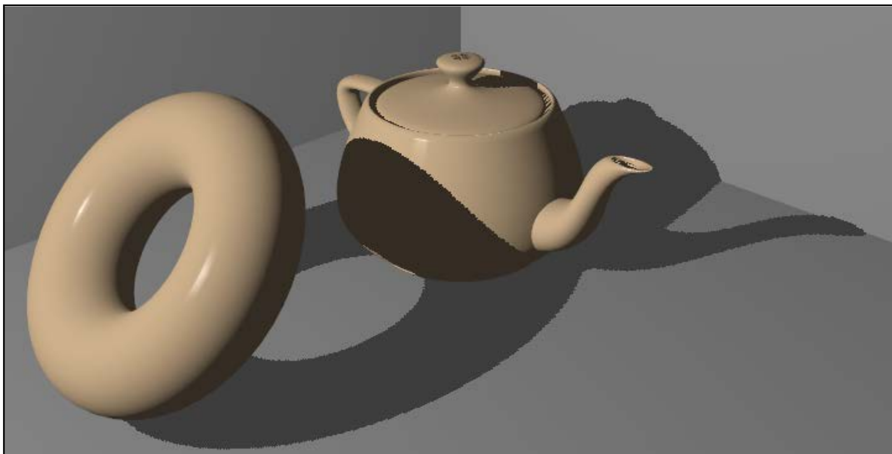
We'll also look at an alternative technique for shadows called shadow volumes. Shadow volumes produce near perfect hard-edged shadows, but are not well suited for creating shadows with soft edges.

Rendering shadows with shadow maps

One of the most common and popular techniques for producing shadows is called shadow mapping. In its basic form, the algorithm involves two passes. In the 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 the shadow map. This map will help provide information about the visibility of objects from the light's perspective. In other words, the shadow map stores the distance (actually the pseudo-depth) from the light to whatever the light can "see". Anything that is closer to the light than the corresponding depth stored in the map is lit; anything else must be in shadow.

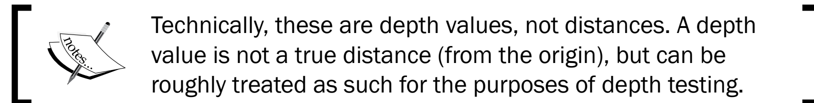
In the second pass, the scene is rendered normally, but each fragment's depth (from the light's perspective) is first tested against the shadow map to determine whether or not the fragment is in shadow. The fragment is then shaded differently depending on the result of this test. If the fragment is in shadow, it is shaded with ambient lighting only; otherwise, it is shaded normally.

The following figure shows an example of shadows produced by the basic shadow mapping technique:



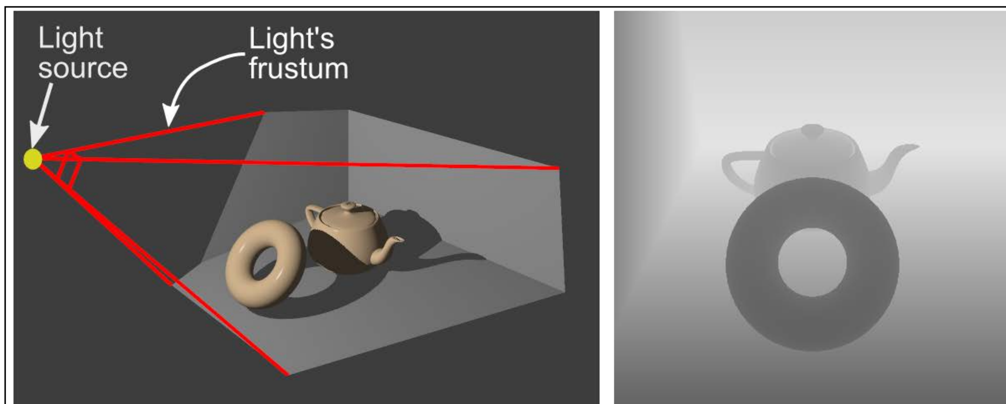
Let's look at each step of the algorithm in detail.

The first step is the creation of the shadow map. We set up our view matrix so that we are rendering the scene as if the camera is located at the position of the light source, and is oriented towards the shadow-casting objects. We set up a projection matrix such that the view frustum encloses all objects that may cast shadows as well as the area where the shadows will appear. We then render the scene normally and store the information from the depth buffer in a texture. This texture is called the shadow map (or simply depth map). We can think of it (roughly) as a set of distances from the light source to various surface locations.



Technically, these are depth values, not distances. A depth value is not a true distance (from the origin), but can be roughly treated as such for the purposes of depth testing.

The following figures represent an example of the basic shadow mapping setup. The left figure shows the light's position and its associated perspective frustum. The right-hand figure shows the corresponding shadow map. The grey scale intensities in the shadow map correspond to the depth values (darker is closer).



Once we have created the shadow map and stored the map to a texture, we render the scene again from the point of view of the camera. This time, 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. Therefore, 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.

The key aspect here is the conversion of the fragment's 3D coordinates to the coordinates appropriate for a lookup into the shadow map. As the shadow map is just a 2D texture, we need coordinates that range from zero to one for points that lie within the light's frustum. The light's view matrix will transform points in world coordinates to points within the light's coordinate system. The light's projection matrix will transform points that are within the light's frustum to **homogeneous clip coordinates**.

These are called clip coordinates because the built-in clipping functionality takes place when the position is defined in these coordinates. Points within the perspective (or orthographic) frustum are transformed by the projection matrix to the (homogeneous) space that is contained within a cube centered at the origin, with each side of length two. This space is called the **canonical viewing volume**. The term "homogeneous" means that these coordinates should not necessarily be considered to be true Cartesian positions until they are divided by their fourth coordinate. For full details about homogeneous coordinates, refer to your favorite textbook on computer graphics.

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. However, 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** (more on this later).

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. Therefore, if we are to use our z coordinate to accurately compare with this depth buffer, we need to scale and translate it appropriately.

In clip coordinates (after perspective division) the z coordinate ranges from -1 to 1. It is the viewport transformation that (among other things) converts the depth to a range between zero and one. Incidentally, if so desired, we can configure the viewport transformation to use some other range for the depth values (say between 0 and 100) via the `glDepthRange` function.

Of course, the x and y components also need to be biased between zero and one because that is the appropriate range for texture access.

We can use the following "bias" matrix to alter our clip coordinates.

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

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. Now, combining the bias matrix with the light's view (V_l) and projection (P_l) matrices, we have the following equation for converting positions in world coordinates (**W**) to homogeneous positions that can be used for shadow map access (**Q**).

$$\mathbf{Q} = \mathbf{B}\mathbf{P}_l\mathbf{V}_l\mathbf{W}$$

Finally, before we can use the value of **Q** directly, we need to divide by the fourth (**w**) component. This step is sometimes called "perspective division". This converts the position from a homogeneous value to a true Cartesian position, and is always required when using a perspective projection matrix.

In the following equation, we'll define a shadow matrix (**S**) that also includes the model matrix (**M**), so that we can convert directly from the modeling coordinates (C). (Note that **W = MC**, because the model matrix takes modeling coordinates to world coordinates.)

$$\mathbf{Q} = \mathbf{S}\mathbf{C}$$

Here, **S** is the shadow matrix, the product of the model matrix with all of the preceding matrices.

$$\mathbf{S} = \mathbf{B}\mathbf{P}_l\mathbf{V}_l\mathbf{M}$$

In this recipe, in order to keep things simple and clear, we'll cover only the basic shadow mapping algorithm, without any of the usual improvements. We'll build upon this basic algorithm in the following recipes. Before we get into the code, we should note that the results will likely be less than satisfying. This is because the basic shadow mapping algorithm suffers from significant aliasing artifacts. Nevertheless, it is still an effective technique when combined with one of many techniques for anti-aliasing. We'll look at some of those techniques in the recipes that follow.

Getting ready

The position should be supplied in vertex attribute zero and the normal in vertex attribute one. Uniform variables for the ADS shading model should be declared and assigned, as well as uniforms for the standard transformation matrices. The `ShadowMatrix` variable should be set to the matrix for converting from modeling coordinates to shadow map coordinates (**S** in the preceding equation).

The uniform variable `ShadowMap` is a handle to the shadow map texture, and should be assigned to texture unit zero.

How to do it...

To create an OpenGL application that creates shadows using the shadow mapping technique, use the following steps. We'll start by setting up a **Framebuffer Object** (**FBO**) to contain the shadow map texture, and then move on to the required shader code:

1. In the main OpenGL program, set up a FBO with a depth buffer only. Declare a `GLuint` variable named `shadowFBO` to store the handle to this `framebuffer`. The depth buffer storage should be a texture object. You can use something similar to the following code to accomplish this:

```
GLfloat border[]={1.0f,0.0f,0.0f,0.0f};

//The shadowmap texture
GLuint depthTex;
glGenTextures(1,&depthTex);
glBindTexture(GL_TEXTURE_2D,depthTex);
glTexStorage2D(GL_TEXTURE_2D, 1, GL_DEPTH_COMPONENT24,
               shadowMapWidth, shadowMapHeight);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,
                GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,
                GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S,
                GL_CLAMP_TO_BORDER);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T,
                GL_CLAMP_TO_BORDER);
glTexParameterfv(GL_TEXTURE_2D,GL_TEXTURE_BORDER_COLOR,
                 border);

glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_COMPARE_MODE,
                GL_COMPARE_REF_TO_TEXTURE);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_COMPARE_FUNC,
                GL_LESS);

//Assign the shadow map to texture unit 0
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D,depthTex);

//Create and set up the FBO
glGenFramebuffers(1,&shadowFBO);
glBindFramebuffer(GL_FRAMEBUFFER,shadowFBO);
glFramebufferTexture2D(GL_FRAMEBUFFER,GL_DEPTH_ATTACHMENT,
                       GL_TEXTURE_2D,depthTex,0);
GLenum drawBuffers[]={GL_NONE};
glDrawBuffers(1,drawBuffers);
// Revert to the default framebuffer for now
glBindFramebuffer(GL_FRAMEBUFFER,0);
```

2. Use the following code for the vertex shader:

```
layout (location=0) in vec3 VertexPosition;
layout (location=1) in vec3 VertexNormal;

out vec3 Normal;
out vec3 Position;

// Coordinate to be used for shadow map lookup
out vec4 ShadowCoord;

uniform mat4 ModelViewMatrix;
uniform mat3 NormalMatrix;
uniform mat4 MVP;
uniform mat4 ShadowMatrix;

void main()
{
    Position = (ModelViewMatrix *
                vec4(VertexPosition,1.0)).xyz;
    Normal = normalize( NormalMatrix * VertexNormal );

    // ShadowMatrix converts from modeling coordinates
    // to shadow map coordinates.
    ShadowCoord =ShadowMatrix * vec4(VertexPosition,1.0);

    gl_Position = MVP * vec4(VertexPosition,1.0);
}
```

3. Use the following code for the fragment shader:

```
// Declare any uniforms needed for your shading model
uniform sampler2DShadow ShadowMap;

in vec3 Position;
in vec3 Normal;
in vec4 ShadowCoord;

layout (location = 0) out vec4 FragColor;

vec3 diffAndSpec()
{
   // Compute only the diffuse and specular components of
   // the shading model.
}
```

```
subroutine void RenderPassType();
subroutine uniform RenderPassType RenderPass;

subroutine (RenderPassType)
void shadeWithShadow()
{
  vec3 ambient = …;// compute ambient component here
  vec3 diffSpec = diffAndSpec();

  // Do the shadow-map look-up
  float shadow = textureProj(ShadowMap, ShadowCoord);

  // If the fragment is in shadow, use ambient light only.
  FragColor = vec4(diffSpec * shadow + ambient, 1.0);
}
subroutine (RenderPassType)
void recordDepth()
{
    // Do nothing, depth will be written automatically
}

void main() {
  // This will call either shadeWithShadow or recordDepth
  RenderPass();
}
```

Within the main OpenGL program, perform the following steps when rendering:

Pass 1

1. Set the viewport, view, and projection matrices to those that are appropriate for the light source.
2. Bind to the `framebuffer` containing the shadow map (`shadowFBO`).
3. Clear the depth buffer.
4. Select the subroutine function `recordDepth`.
5. Enable front-face culling.
6. Draw the scene.

Pass 2

1. Select the viewport, view, and projection matrices appropriate for the scene.
2. Bind to the default framebuffer.
3. Disable culling (or switch to back-face culling).
4. Select the subroutine function `shadeWithShadow`.
5. Draw the scene.

How it works...

The first block of the preceding code demonstrates how to create a FBO for our shadow map texture. The FBO contains only a single texture connected to its depth buffer attachment. The first few lines of code create the shadow map texture. The texture is allocated using the `glTexStorage2D` function with an internal format of `GL_DEPTH_COMPONENT24`.

We use `GL_NEAREST` for `GL_TEXTURE_MAG_FILTER` and `GL_TEXTURE_MIN_FILTER` here, although `GL_LINEAR` could also be used, and might provide slightly better-looking results. We use `GL_NEAREST` here so that we can see the aliasing artifacts clearly, and the performance will be slightly better.

Next, the `GL_TEXTURE_WRAP_*` modes are set to `GL_CLAMP_TO_BORDER`. When a fragment is found to lie completely outside of the shadow map (outside of the light's frustum), then the texture coordinates for that fragment will be greater than one or less than zero. When that happens, we need to make sure that those points are not treated as being in shadow. When `GL_CLAMP_TO_BORDER` is used, the value that is returned from a texture lookup (for coordinates outside the 0..1 range) will be the border value. The default border value is `(0,0,0,0)`. When the texture contains depth components, the first component is treated as the depth value. A value of zero will not work for us here because a depth of zero corresponds to points on the near plane. Therefore all points outside of the light's frustum will be treated as being in shadow! Instead, we set the border color to `(1,0,0,0)` using the `glTexParameterfv` function, which corresponds to the maximum possible depth.

The next two calls to `glTexParameteri` affect settings that are specific to depth textures. The first call sets `GL_TEXTURE_COMPARE_MODE` to `GL_COMPARE_REF_TO_TEXTURE`. When this setting is enabled, the result of a texture access is the result of a comparison, rather than a color value retrieved from the texture. The third component of the texture coordinate (the `p` component) is compared against the value in the texture at location (s,t). The result of the comparison is returned as a single floating-point value. The comparison function that is used is determined by the value of `GL_TEXTURE_COMPARE_FUNC`, which is set on the next line. In this case, we set it to `GL_LESS`, which means that the result will be `1.0` if the `p` value of the texture coordinate is less than the value stored at (s,t). (Other options include `GL_LEQUAL`, `GL_ALWAYS`, `GL_GEQUAL`, and so on.)

The next few lines create and set up the FBO. The shadow map texture is attached to the FBO as the depth attachment with the `glFramebufferTexture2D` function. For more details about FBOs, check out the *Rendering to a texture* recipe in *Chapter 4*, *Using Textures*.

The vertex shader is fairly simple. It converts the vertex position and normal to camera coordinates and passes them along to the fragment shader via the output variables `Position` and `Normal`. The vertex position is also converted into shadow map coordinates using `ShadowMatrix`. This is the matrix S that we referred to in the previous section. It converts a position from modeling coordinates to shadow coordinates. The result is sent to the fragment shader via the output variable `ShadowCoord`.

As usual, the position is also converted to clip coordinates and assigned to the built-in output variable `gl_Position`.

In the fragment shader, we provide different functionality for each pass. In the main function, we call `RenderPass`, which is a subroutine uniform that will call either `recordDepth` or `shadeWithShadow`. For the first pass (shadow map generation), the subroutine function `recordDepth` is executed. This function does nothing at all! This is because we only need to write the depth to the depth buffer. OpenGL will do this automatically (assuming that `gl_Position` was set correctly by the vertex shader), so there is nothing for the fragment shader to do.

During the second pass, the `shadeWithShadow` function is executed. We compute the ambient component of the shading model and store the result in the `ambient` variable. We then compute the diffuse and specular components and store those in the `diffuseAndSpec` variable.

The next step is the key to the shadow mapping algorithm. We use the built-in texture access function `textureProj`, to access the shadow map texture `ShadowMap`. Before using the texture coordinate to access the texture, the `textureProj` function will divide the first three components of the texture coordinate by the fourth component. Remember that this is exactly what is needed to convert the homogeneous position (`ShadowCoord`) to a true Cartesian position.

After this perspective division, the `textureProj` function will use the result to access the texture. As this texture's sampler type is `sampler2DShadow`, it is treated as texture containing depth values, and rather than returning a value from the texture, it returns the result of a comparison. The first two components of `ShadowCoord` are used to access a depth value within the texture. That value is then compared against the value of the third component of `ShadowCoord`. When `GL_NEAREST` is the interpolation mode (as it is in our case) the result will be `1.0` or `0.0`. As we set the comparison function to `GL_LESS`, this will return `1.0`, if the value of the third component of `ShadowCoord` is less than the value within the depth texture at the sampled location. This result is then stored in the variable `shadow`. Finally, we assign a value to the output variable `FragColor`. The result of the shadow map comparison (`shadow`) is multiplied by the diffuse and specular components, and the result is added to the ambient component. If `shadow` is `0.0`, that means that the comparison failed, meaning that there is something between the fragment and the light source. Therefore, the fragment is only shaded with ambient light. Otherwise, `shadow` is 1.0, and the fragment is shaded with all three shading components.

When rendering the shadow map, note that we culled the front faces. This is to avoid the z-fighting that can occur when front faces are included in the shadow map. Note that this only works if our mesh is completely closed. If back faces are exposed, you may need to use another technique (that uses `glPolygonOffset`) to avoid this. I'll talk a bit more about this in the next section.

There's more...

There's a number of challenging issues with the shadow mapping technique. Let's look at just a few of the most immediate ones.

Aliasing

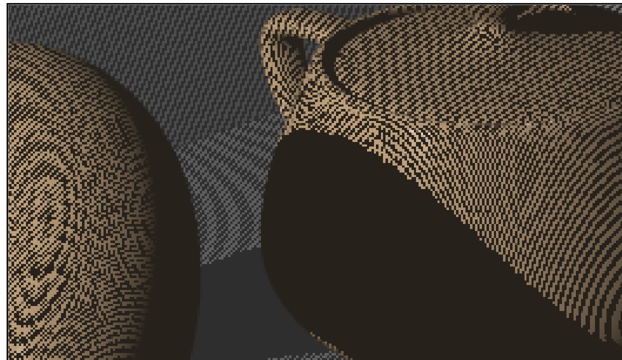
As mentioned earlier, this algorithm often suffers from severe aliasing artifacts at the shadow's edges. This is due to the fact that the shadow map is essentially projected onto the scene when the depth comparison is made. If the projection causes the map to be magnified, aliasing artifacts appear.

The following figure shows the aliasing of the shadow's edges:

The easiest solution is to simply increase the size of the shadow map. However, that may not be possible due to memory, CPU speed, or other constraints. There is a large number of techniques for improving the quality of the shadows produced by the shadow mapping algorithm such as resolution-matched shadow maps, cascaded shadow maps, variance shadow maps, perspective shadow maps and many others. In the following recipes, we'll look at some ways to help soften and anti-alias the edges of the shadows.

Rendering back faces only for the shadow map

When creating the shadow map, we only rendered back faces. This is because of the fact that if we were to render front faces, points on certain faces would have nearly the same depth as the shadow map's depth, which can cause fluctuations between light and shadow across faces that should be completely lit. The following figure shows an example of this effect:



Since the majority of faces that cause this issue are those that are facing the light source, we avoid much of the problem by only rendering back faces during the shadow map pass. This of course will only work correctly if your meshes are completely closed. If that is not the case, `glPolygonOffset` can be used to help the situation by offsetting the depth of the geometry from that in the shadow map. In fact, even when back faces are only rendered when generating the shadow map, similar artifacts can appear on faces that are facing away from the light (back faces in the shadow map, but front from the camera's perspective). Therefore, it is quite often the case that a combination of front-face culling and `glPolygonOffset` is used when generating the shadow map.

See also

- The *Rendering to a texture* recipe in *Chapter 4*, *Using Textures*
- The *Anti-aliasing shadow edges with PCF* recipe
- The *Creating soft shadow edges with random sampling* recipe

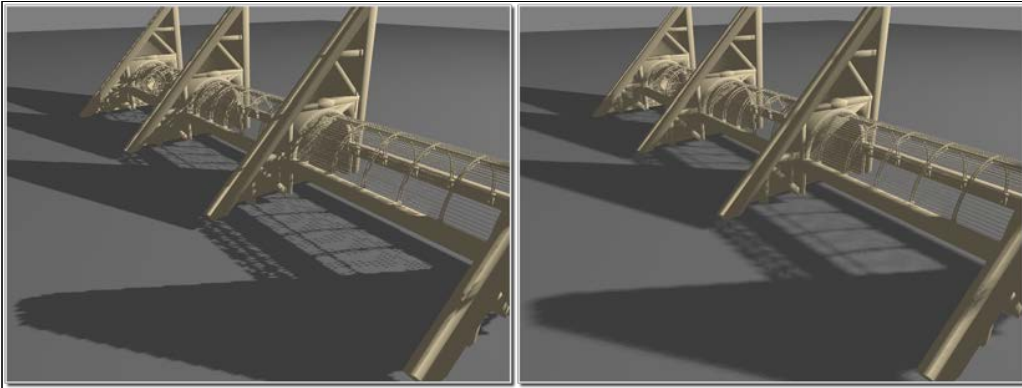
Anti-aliasing shadow edges with PCF

One of the simplest and 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 (diffuse and specular) shading that the fragment receives. The overall effect is a blurring of the shadow's edges.

The basic technique was first published by *Reeves* et al in a 1987 paper (*SIGGRAPH Proceedings, Volume 21, Number 4, July 1987*). The concept involved 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.

A common variant of the PCF algorithm 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. While the result may not be physically accurate, the result is not objectionable to the eye.

The following figures show shadows rendered with PCF (right) and without PCF (left). Note that the shadows in the right-hand image have fuzzier edges and the aliasing is less visible.



In this recipe, we'll use the latter technique, and sample a constant number of texels around the fragment's position in the shadow map. We'll calculate an average of the resulting comparisons and use that result to scale the diffuse and specular components.

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 (the details of this are implementation dependent). Therefore, when linear filtering is enabled, the result of the `textureProj` function can be somewhere between 0.0 and 1.0.

We'll also make use of the built-in functions for texture accesses with offsets. OpenGL provides the texture access function `textureProjOffset`, which has a third parameter (the offset) that is added to the texel coordinates before the lookup/comparison.

Getting ready

Start with the shaders and FBO presented in the previous recipe, *Rendering shadows with shadow maps*. We'll just make a few minor changes to the code presented there.

How to do it...

To add the PCF technique to the shadow mapping algorithm, use the following steps:

1. When setting up the FBO for the shadow map, make sure to use linear filtering on the depth texture. Replace the corresponding lines with the following code:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                GL_LINEAR);
```

2. Use the following code for the `shadeWithShadow` function within the fragment shader:

```
subroutine (RenderPassType)
void shadeWithShadow()
{
  vec3 ambient = vec3(0.2);
  vec3 diffSpec = diffAndSpec();

  // The sum of the comparisons with nearby texels
  float sum = 0;

  // Sum contributions from texels around ShadowCoord
  sum += textureProjOffset(ShadowMap, ShadowCoord,
                           ivec2(-1,-1));
  sum += textureProjOffset(ShadowMap, ShadowCoord,
                           ivec2(-1,1));
  sum += textureProjOffset(ShadowMap, ShadowCoord,
                           ivec2(1,1));
  sum += textureProjOffset(ShadowMap, ShadowCoord,
                           ivec2(1,-1));
  float shadow = sum * 0.25;


  FragColor = vec4(ambient + diffSpec * shadow,1.0);
}
```

How it works...

The first step enables linear filtering on the shadow map texture. When this is enabled, the OpenGL driver can repeat the depth comparison on the four nearby texels within the texture. The results of the four comparisons will be averaged and returned.

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.

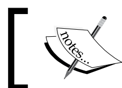
As linear filtering is enabled, each lookup will sample an additional four texels, for a total of 16 texels. The results are then averaged together and stored within the variable shadow.

As before, the value of `shadow` is used to attenuate the diffuse and specular components of the lighting model.

There's more...

An excellent survey of the PCF technique was written by *Fabio Pellacini* of *Pixar*, and can be found in *Chapter 11*, *Shadow Map Anti-aliasing* of *GPU Gems*, edited by *Randima Fernando*, *Addison-Wesley Professional*, *2004*. If more details are desired, I highly recommend reading this short, but informative, chapter.

Because of its simplicity and efficiency, the PCF technique is an extremely common method for anti-aliasing the edges of shadows produced by shadow mapping. Since it has the effect of blurring the edges, it can also be used to simulate soft shadows. However, the number of samples must be increased with the size of the blurred edge (the penumbra) to avoid certain artifacts. This can, of course, be a computational roadblock. In the next recipe, we'll look at a technique for producing soft shadows by randomly sampling a larger region.



The penumbra is the region of a shadow where only a portion of the light source is obscured.

See also

- The *Rendering shadows with shadow maps* recipe

Creating soft shadow edges with random sampling

The basic shadow mapping algorithm combined with PCF can produce shadows with soft edges. However, if we desire blurred edges that are substantially wide (to approximate true soft shadows) then a large number of samples is required. Additionally, there is a good deal of wasted effort when shading fragments that are in the center of large shadows, or completely outside of the shadow. For those fragments, all of the nearby shadow map texels will evaluate to the same value. Therefore, the work of accessing and averaging those texels is essentially a wasted effort.

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*). It provides an approach that can address both of the preceding issues to create shadows with soft edges of various widths, while avoiding unnecessary texture accesses in areas inside and outside of the shadow.

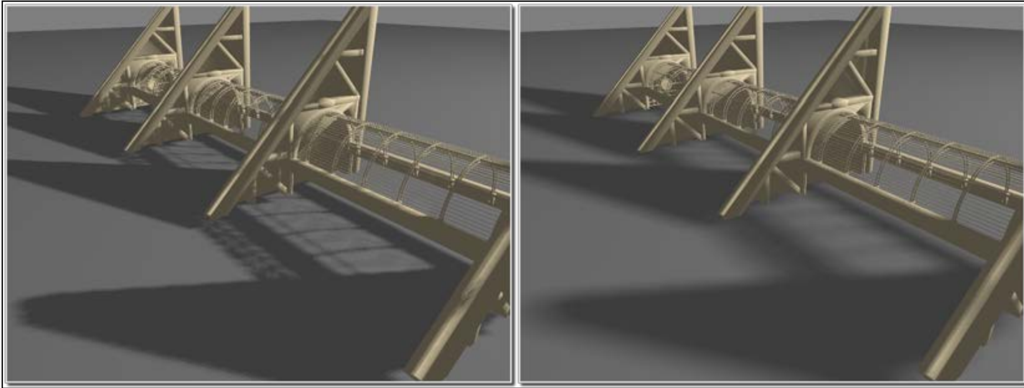
The basic idea is as follows:

- 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

The following figure is a visualization of a possible set of shadow map samples. The center of the cross-hairs is the fragment's location in the shadow map, and each **x** is a sample. The samples are distributed randomly within a circular grid around the fragment's location (one sample per grid cell).

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 use 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.

The following figures show the difference between shadows using the PCF algorithm (left), and the random sampling technique described in this recipe (right).



We'll store the offsets in a three-dimensional texture (n x n x d). The first two dimensions are of arbitrary size, and the third dimension contains the offsets. Each (s,t) location contains a list (size d) of random offsets packed into an RGBA color. Each RGBA color 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 (s,t) location contains a total of `2*d` offsets. For example, location (1, 1, 3) contains the sixth and seventh offset at location (1,1). The entire set of values at a given (s,t) comprise a full set of offsets.

We'll rotate through the texture based on the fragment's screen coordinates. The location within the offset texture will be determined by taking the remainder of the screen coordinates divided by the texture's size. For example, if the fragment's coordinates are (10.0,10.0) and the texture's size is (4,4), then we use the set of offsets located in the offset texture at location (2,2).

Getting ready

Start with the code presented in the *Rendering shadows with shadow maps* recipe.

There are three additional uniforms that need to be set. They are as follows:

- `OffsetTexSize`: This gives the width, height, and depth of the offset texture. Note that the depth is same as the number of samples per fragment divided by two.
- `OffsetTex`: This is a handle to the texture unit containing the offset texture.

- `Radius`: This is the blur radius in pixels divided by the size of the shadow map texture (assuming a square shadow map). This could be considered as the softness of the shadow.

How to do it...

To modify the shadow mapping algorithm and to use this random sampling technique, use the following steps. We'll build the offset texture within the main OpenGL program, and make use of it within the fragment shader:

1. Use the following code within the main OpenGL program to create the offset texture. This only needs to be executed once during the program initialization:

```
void buildOffsetTex(int size, int samplesU, int samplesV)
{
  int samples = samplesU * samplesV;
  int bufSize = size * size * samples * 2;
  float *data = new float[bufSize];

  for( int i = 0; i< size; i++ ) {
    for(int j = 0; j < size; j++ ) {
      for( int k = 0; k < samples; k += 2 ) {
        int x1,y1,x2,y2;
        x1 = k % (samplesU);
        y1 = (samples - 1 - k) / samplesU;
        x2 = (k+1) % samplesU;
        y2 = (samples - 1 - k - 1) / samplesU;

        vec4 v;
        // Center on grid and jitter
        v.x = (x1 + 0.5f) + jitter();
        v.y = (y1 + 0.5f) + jitter();
        v.z = (x2 + 0.5f) + jitter();
        v.w = (y2 + 0.5f) + jitter();
        // Scale between 0 and 1
        v.x /= samplesU;
        v.y /= samplesV;
        v.z /= samplesU;
        v.w /= samplesV;
        // Warp to disk
        int cell = ((k/2) * size * size + j *
                    size + i) * 4;
        data[cell+0] = sqrtf(v.y) * cosf(TWOPI*v.x);
        data[cell+1] = sqrtf(v.y) * sinf(TWOPI*v.x);
        data[cell+2] = sqrtf(v.w) * cosf(TWOPI*v.z);
        data[cell+3] = sqrtf(v.w) * sinf(TWOPI*v.z);
```

```
      }
    }
  }

  glActiveTexture(GL_TEXTURE1);
  GLuint texID;
  glGenTextures(1, &texID);

  glBindTexture(GL_TEXTURE_3D, texID);
  glTexStorage3D(GL_TEXTURE_3D, 1, GL_RGBA32F, size, size,
                 samples/2);
  glTexSubImage3D(GL_TEXTURE_3D, 0, 0, 0, 0, size, size,
                 samples/2, GL_RGBA, GL_FLOAT, data);
  glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_MAG_FILTER,
                   GL_NEAREST);
  glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_MIN_FILTER,
                   GL_NEAREST);

  delete [] data;
}

// Return random float between -0.5 and 0.5
float jitter() {
  return ((float)rand() / RAND_MAX) - 0.5f;
}
```

2. Add the following uniform variables to the fragment shader:

```
uniform sampler3D OffsetTex;
uniform vec3 OffsetTexSize; // (width, height, depth)
uniform float Radius;
```

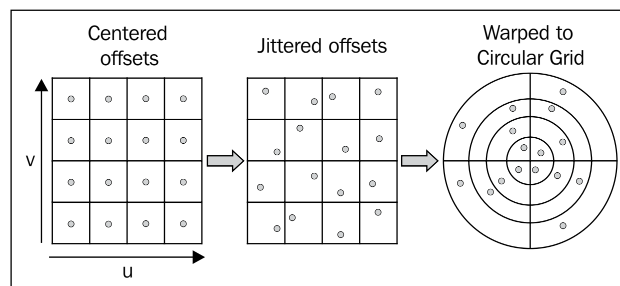
3. Use the following code for the `shadeWithShadow` function in the fragment shader:

```
subroutine (RenderPassType)
void shadeWithShadow()
{
  vec3 ambient = vec3(0.2);
  vec3 diffSpec = diffAndSpec();

  ivec3 offsetCoord;
  offsetCoord.xy = ivec2( mod( gl_FragCoord.xy,
                         OffsetTexSize.xy ) );

  float sum = 0.0;
  int samplesDiv2 = int(OffsetTexSize.z);
  vec4 sc = ShadowCoord;

  for( int i = 0 ; i< 4; i++ ) {
    offsetCoord.z = i;
```

```
      vec4 offsets = texelFetch(OffsetTex,offsetCoord,0) *
                         Radius * ShadowCoord.w;

      sc.xy = ShadowCoord.xy + offsets.xy;
      sum += textureProj(ShadowMap, sc);
      sc.xy = ShadowCoord.xy + offsets.zw;
      sum += textureProj(ShadowMap, sc);
    }
    float shadow = sum / 8.0;

    if( shadow != 1.0 && shadow != 0.0 ) {
      for( int i = 4; i< samplesDiv2; i++ ) {
        offsetCoord.z = i;
        vec4 offsets =
          texelFetch(OffsetTex, offsetCoord,0) *
                     Radius * ShadowCoord.w;

        sc.xy = ShadowCoord.xy + offsets.xy;
        sum += textureProj(ShadowMap, sc);
        sc.xy = ShadowCoord.xy + offsets.zw;
        sum += textureProj(ShadowMap, sc);
      }
      shadow = sum / float(samplesDiv2 * 2.0);
    }
    FragColor = vec4(diffSpec * shadow + ambient, 1.0);
  }
```

How it works...

The `buildOffsetTex` function creates our three-dimensional texture of random offsets. The first parameter, `texSize`, defines the width and height of the texture. To create the preceding images, I used a value of 8. The second and third parameters, `samplesU` and `samplesV`, define the number of samples in the `u` and `v` directions. I used a value of 4 and 8, respectively, for a total of 32 samples. The **u** and **v** directions are arbitrary axes that are used to define a grid of offsets. To help understand this, take a look at the following figure:



The offsets are initially defined to be centered on a grid of size `samplesU` x `samplesV` (4 x 4 in the preceding figure). 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 such that they surround the origin and lie within the circular grid shown on the right.

The last step can be accomplished by using the v coordinate as the distance from the origin and the u coordinate as the angle scaled from 0 to 360. The following equations should do the trick:

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

Here, 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 towards the center. This will help us avoid taking too many samples when we are working completely inside or outside of the shadow.

Of course, 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.

Within the fragment shader, we start by computing the ambient component of the shading model separately from the diffuse and specular components. We access the offset texture at a location based on the fragment's screen coordinates (`gl_FragCoord`). We do so by taking the modulus of the fragment's position and the size of the offset texture. The result is stored in the first two components of `offsetCoord`. This will give us a different set of offsets for each nearby pixel. The third components of `offsetCoord` will be used to access a pair of samples. The number of samples is the depth of the texture divided by two. This is stored in `samplesDiv2`. We access the sample using the `texelFetch` function. This function allows us to access a texel using the integer texel coordinates rather than the usual normalized texture coordinates in the range 0...1.

The offset is retrieved and multiplied by `Radius` and the w component of `ShadowCoord`. Multiplying by `Radius` simply scales the offsets so that they range from 0.0 to `Radius`. We multiply by the w component because `ShadowCoord` is still a homogeneous coordinate, and our goal is to use offsets to translate the `ShadowCoord`. In order to do so properly, we need to multiply the offset by the w component. Another way of thinking of this is that the w component will be cancelled when perspective division takes place.

Next, we use offsets to translate `ShadowCoord` and access the shadow map to do the depth comparison using `textureProj`. We do so for each of the two samples stored in the texel, once for the first two components of offsets and again for the last two. The result is added to `sum`.

The first loop repeats this for the first eight samples. If the first eight samples are all 0.0 or 1.0, then we assume that all of the samples will be the same (the sample area is completely in or out of the shadow). In that case, we skip the evaluation of the rest of the samples. Otherwise, we evaluate the following samples and compute the overall average.

Finally the resulting average (shadow) is used to attenuate the diffuse and specular components of the lighting model.

There's more...

The use of a small texture containing a set of random offsets helps to blur the edges of the shadow better than what we might achieve with the standard PCF technique that uses a constant set of offsets. However, artifacts can still appear as repeated patterns within the shadow edges because the texture is finite and offsets are repeated every few pixels. We could improve this by also using a random rotation of the offsets within the fragment shader, or simply compute the offsets randomly within the shader.

It should also be noted that this blurring of the edges may not be desired for all shadow edges. For example, edges that are directly adjacent to the occluder, that is, creating the shadow, should not be blurred. These may not always be visible, but can become so in certain situations, such as when the occluder is a narrow object. The effect is to make the object appear as if it is hovering above the surface. Unfortunately, there isn't an easy fix for this one.

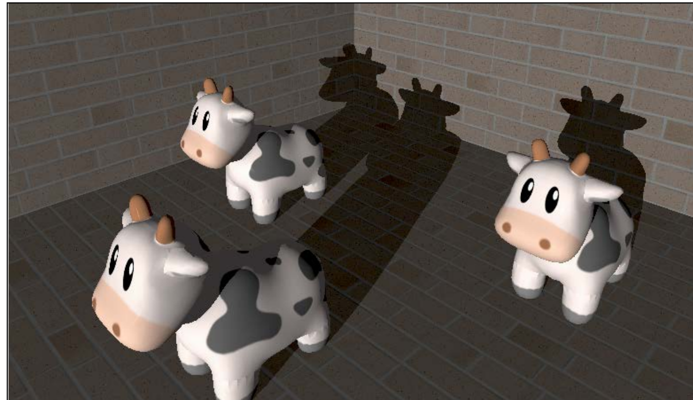
See also

- The *Rendering shadows with shadow maps* recipe

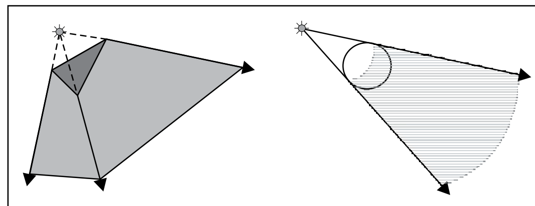
Creating shadows using shadow volumes and the geometry shader

As we discovered in the previous recipes, one of the main problems with shadow maps is aliasing. The problem essentially boils down to the fact that we are sampling the shadow map(s) at a different frequency (resolution) than we are using when rendering the scene. To minimize the aliasing we can blur the shadow edges (as in the previous recipes), or try to sample the shadow map at a frequency that is closer to the corresponding resolution in projected screen space. There are many techniques that help with the latter; for more details, I recommend the book *Real-Time Shadows*.

An alternate technique for shadow generation is called *shadow volumes*. The shadow volume method completely avoids the aliasing problem that plagues shadow maps. With shadow volumes, you get pixel-perfect hard shadows, without the aliasing artifacts of shadow maps. The following figure shows a scene with shadows that are produced using the shadow volume technique.

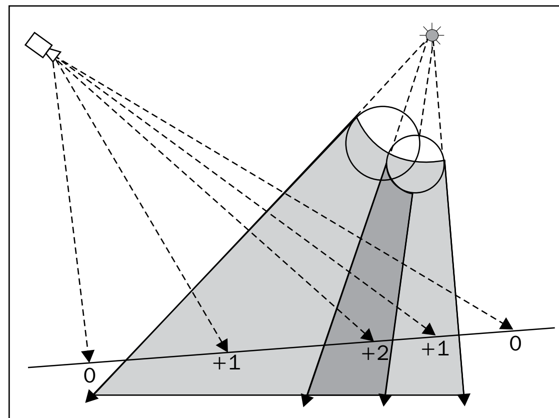


The shadow volume technique works by making use of the stencil buffer to mask out areas that are in shadow. We do this by drawing the boundaries of the actual shadow volumes (more on this below). A shadow volume is the region of space where the light source is occluded by an object. For example, the following figures show a representation of the shadow volumes of a triangle (left) and a sphere (right).



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. For an object that consists of many triangles, such as the sphere above, 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, one would 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.

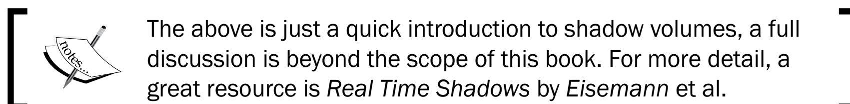
The shadow volume technique works in the following way. Imagine a ray that originates at the camera position and extends through a pixel on the near plane. Suppose that we follow that ray and keep track of a counter that is incremented every time that it enters a shadow volume and decremented each time that it exits a shadow volume. If we stop counting when we hit a surface, that point on the surface is occluded (in shadow) if our count is non-zero, otherwise, the surface is lit by the light source. The following figure shows an example of this idea:



The roughly horizontal line represents a surface that is receiving a shadow. The numbers represent the counter for each camera ray. For example, the rightmost ray with value **+1** has that value because the ray entered two volumes and exited one along the way from the camera to the surface: 1 + 1 - 1 = 1. The rightmost ray has a value of zero at the surface because it entered and exited both shadow volumes: 1 + 1 - 1 - 1 = 0.

This all sounds fine in theory, but how can we trace rays in OpenGL? The good news is that we don't have to. The stencil buffer provides just what we need. 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. So 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 (as in the above diagram). 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.



The above is just a quick introduction to shadow volumes, a full discussion is beyond the scope of this book. For more detail, a great resource is *Real Time Shadows* by *Eisemann* et al.

In this recipe, we'll draw our shadow volumes with the help of the geometry shader. 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. In the *Drawing silhouette lines using the geometry shader* recipe in *Chapter 6*, *Using Geometry and Tessellation Shaders*, we saw how 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 neighboring 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.

The entire process is done in three passes. They are as follows:

- Render the scene normally, but write the shaded color to two separate buffers. We'll store the ambient component in one and the diffuse and specular components in another.
- 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.
- Set up the stencil buffer so 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.

That's the high-level view, and there are many details. Let's go through them in the next sections.

Getting ready

We'll start by creating our buffers. We'll use a framebuffer object with a depth attachment and two color 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 ambient buffer (`ambBuf`), a depth buffer (`depthBuf`), and a texture (`diffSpecTex`), then set up the FBO.

```
glGenFramebuffers(1, &colorDepthFBO);
glBindFramebuffer(GL_FRAMEBUFFER, colorDepthFBO);
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
                          GL_RENDERBUFFER, depthBuf);
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
                          GL_RENDERBUFFER, ambBuf);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT1,
                       GL_TEXTURE_2D, diffSpecTex, 0);
```

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);
```

How to do it...

For the first pass, enable the framebuffer object that we set up above, 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(); }
```

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);
```

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-1,height-1,0,0,width-1,height-1,
           GL_DEPTH_BUFFER_BIT|GL_COLOR_BUFFER_BIT, GL_NEAREST);
```

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);
```

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 )
{
  vec3 n = cross( b - a, c - a );
  vec3 da = LightPosition.xyz - a;
  vec3 db = LightPosition.xyz - b;
  vec3 dc = LightPosition.xyz - c;
  return dot(n, da) > 0 || dot(n, db) > 0 || dot(n, dc) > 0;
}

void emitEdgeQuad( vec3 a, vec3 b ) {
  gl_Position = ProjMatrix * vec4(a, 1);
  EmitVertex();
  gl_Position = ProjMatrix * vec4(a - LightPosition.xyz, 0);
  EmitVertex();
  gl_Position = ProjMatrix * vec4(b, 1);
  EmitVertex();
  gl_Position = ProjMatrix * vec4(b - LightPosition.xyz, 0);
  EmitVertex();
  EndPrimitive();
}

void main()
{
  if( facesLight(VPosition[0], VPosition[2], VPosition[4]) ) {
    if( ! facesLight(VPosition[0],VPosition[1],VPosition[2]) )
       emitEdgeQuad(VPosition[0],VPosition[2]);
    if( ! facesLight(VPosition[2],VPosition[3],VPosition[4]) )
       emitEdgeQuad(VPosition[2],VPosition[4]);
    if( ! facesLight(VPosition[4],VPosition[5],VPosition[0]) )
       emitEdgeQuad(VPosition[4],VPosition[0]);
  }
}
```

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 the diffuse + specular when the stencil test succeeds.

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

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 (we copied it over earlier using `glBlitFramebuffer`).

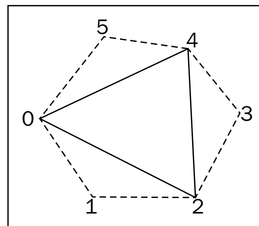
```
layout(binding = 0) uniform sampler2D DiffSpecTex;
layout(location = 0) out vec4 FragColor;

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

How it works...

The first pass is fairly straightforward. We draw the entire scene normally, except we separate the ambient color from the diffuse and specular color, and send the results to different buffers.

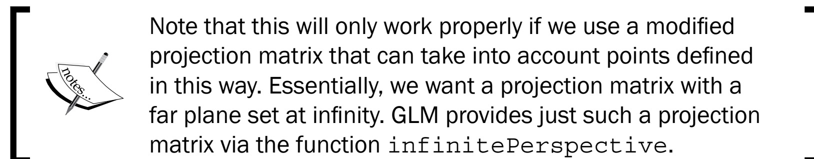
The second pass is the core of the algorithm. Here we render only the objects that cast shadows and let the geometry shader produce the shadow volumes. Thanks to the geometry shader, we don't actually end up rendering the shadow-casting objects at all, only the shadow volumes. However, before this pass, we need to do a bit of setup. We set up the stencil test so that it increments when a front face is rendered and decrements for back faces using `glStencilOpSeparate`, and the stencil test is configured to always succeed using `glStencilFunc`. We also use `glBlitFramebuffer` to copy over the depth buffer and (ambient) color buffer from the FBO used in the first pass. Since we want to only render shadow volumes that are not obscured by geometry, we make the depth buffer read-only using `glDepthMask`. Lastly, we disable writing to the color buffer using `glColorMask` because we don't want to mistakenly overwrite anything in this pass.

The geometry shader does the work of producing the silhouette shadow volumes. Since we are rendering using adjacency information (see the *Drawing silhouette lines using the geometry shader* recipe in *Chapter 6*, *Using Geometry and Tessellation Shaders*), the geometry shader has access to six vertices that define the current triangle being rendered and the three neighboring triangles. The vertices are numbered from 0 to 5, and are available via the input array named `VPosition` in this example. Vertices 0, 2, and 4 define the current triangle and the others define the adjacent triangles as shown in the following figure:



The geometry shader starts by testing the main triangle (0, 2, 4) to see if it faces the light source. We do so by computing the normal to the triangle (`n`) and the vector from each vertex to the light source. Then we compute the dot product of n and each of the three light source direction vectors (`da`, `db`, and `dc`). If any of the three are positive, then the triangle faces the light source. If we find that triangle (0, 2, 4) faces the light, then we test each neighboring triangle in the same way. If a neighboring triangle does not face the light source, then the edge between them is a silhouette edge and can be used as an edge of a face of the shadow volume.

We create a shadow volume face in the `emitEdgeQuad` function. The points `a` and `b` define the silhouette edge, one edge of the shadow volume face. The other two vertices of the face are determined by extending `a` and `b` away from the light source. Here, we use a mathematical trick that is enabled by homogeneous coordinates. We extend the face out to infinity by using a zero in the w coordinate of the extended vertices. This effectively defines a homogeneous vector, sometimes called a point at infinity. The x, y and z coordinates define a vector in the direction away from the light source, and the w value is set to zero. The end result is that we get a quad that extends out to infinity, away from the light source.



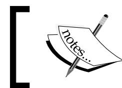
Note that this will only work properly if we use a modified projection matrix that can take into account points defined in this way. Essentially, we want a projection matrix with a far plane set at infinity. GLM provides just such a projection matrix via the function `infinitePerspective`.

We don't worry about drawing the caps of the shadow volume here. We don't need a cap at the far end, because we've used the homogeneous trick described above, and the object itself will serve as the cap on the near end.

In the third and final pass, we reset our stencil test to pass when the value in the stencil buffer is equal to zero using `glStencilFunc`. Here we want to sum the ambient with the diffuse + specular color when the stencil test succeeds, so we enable blending, and set the source and destination blend functions to `GL_ONE`. We render just a single screen-filling quad, and output the value from the texture that contains our diffuse + specular color. The stencil test will take care of discarding fragments that are in shadow, and OpenGL's blending support will blend the output with the ambient color for fragments that pass the test. (Remember that we copied over the ambient color using `glBlitFramebuffer` earlier.)

There's more...

The technique described above is often referred to as the **z-pass** technique. It has one fatal flaw. 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. The common solution is to basically invert the problem and trace a ray from infinity towards the view point. This is called the z-fail technique or Carmack's reverse.



> The "fail" and "pass" here refers to whether or not we are counting when the depth test passes or fails.

Care must be taken when using z-fail because it is important to draw the caps of the shadow volumes. However, 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 towards the view point.

I should also note that the preceding code is not robust to degenerate triangles (triangles that have sides that are nearly parallel), or non-closed meshes. One might need to take care in such situations. For example, to better deal with degenerate triangles we could use another technique for determining the normal to the triangle. We could also add additional code to handle edges of meshes, or simply always use closed meshes.

See also

- The *Drawing silhouette lines using the geometry shader* recipe in *Chapter 6*, *Using Geometry and Tessellation Shaders*