3
Lighting, Shading, and Optimization

In this chapter, we will cover:

- Shading with multiple positional lights
- Shading with a directional light source
- Using per-fragment shading for improved realism
- Using the halfway vector for improved performance
- Simulating a spotlight
- Creating a cartoon shading effect
- Simulating fog
- Configuring the depth test

Introduction

In *Chapter 2*, *The Basics of GLSL Shaders*, we covered a number of techniques for implementing some of the shading effects that were produced by the former fixed-function pipeline. We also looked at some basic features of GLSL such as functions and subroutines. In this chapter, we'll move beyond the shading model introduced in *Chapter 2*, *The Basics of GLSL Shaders*, and see how to produce shading effects such as spotlights, fog, and cartoon style shading. We'll cover how to use multiple light sources, and how to improve the realism of the results with a technique called per-fragment shading.

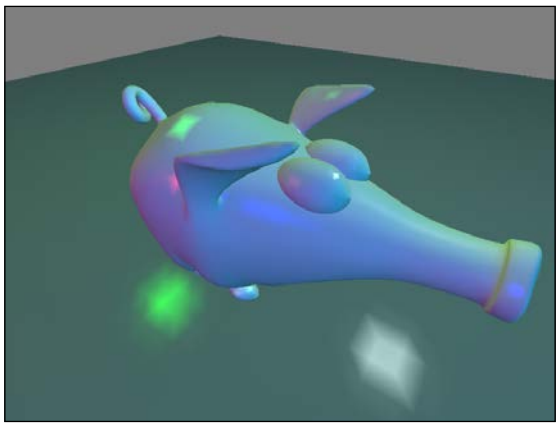
We'll also see techniques for improving the efficiency of the shading calculations by using the so-called "halfway vector" and directional light sources.

Finally, we'll cover how to fine-tune the depth test by configuring the early depth test optimization.

Shading with multiple positional lights

When shading with multiple light sources, we need to evaluate the shading equation for each light and sum the results to determine the total light intensity reflected by a surface location. The natural choice is to create uniform arrays to store the position and intensity of each light. We'll use an array of structures so that we can store the values for multiple lights within a single uniform variable.

The following figure shows a "pig" mesh rendered with five light sources of different colors. Note the multiple specular highlights.



Getting ready

Set up your OpenGL program with the vertex position in attribute location zero, and the normal in location one.

How to do it...

To create a shader program that renders using the ADS (Phong) shading model with multiple light sources, use the following steps:

1. Use the following vertex shader:

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

```
out vec3 Color;

struct LightInfo {
  vec4 Position;  // Light position in eye coords.
  vec3 Intensity; // Light intensity
};
uniform LightInfo lights[5];

// Material parameters
uniform vec3 Kd;            // Diffuse reflectivity
uniform vec3 Ka;            // Ambient reflectivity
uniform vec3 Ks;            // Specular reflectivity
uniform float Shininess;    // Specular shininess factor

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

vec3 ads( int lightIndex, vec4 position, vec3 norm )
{
  vec3 s = normalize( vec3(lights[lightIndex].Position -
                           position) );
  vec3 v = normalize(vec3(-position));
  vec3 r = reflect( -s, norm );
  vec3 I = lights[lightIndex].Intensity;
  return
        I * ( Ka +
        Kd * max( dot(s, norm), 0.0 ) +
        Ks * pow( max( dot(r,v), 0.0 ), Shininess ) );
}
void main()
{
  vec3 eyeNorm = normalize( NormalMatrix * VertexNormal);
  vec4 eyePosition = ModelViewMatrix *
                     vec4(VertexPosition,1.0);

  // Evaluate the lighting equation for each light
  Color = vec3(0.0);
  for( int i = 0; i < 5; i++ )
    Color += ads( i, eyePosition, eyeNorm );

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

2. Use the following simple fragment shader:

```
in vec3 Color;

layout( location = 0 ) out vec4 FragColor;

void main() {
  FragColor = vec4(Color, 1.0);
}
```

3. In the OpenGL application, set the values for the `lights` array in the vertex shader. For each light, use something similar to the following code. This example uses the C++ shader program class (`prog` is a `GLSLProgram` object).

```
prog.setUniform("lights[0].Intensity",
                vec3(0.0f,0.8f,0.8f) );
prog.setUniform("lights[0].Position", position );
```

   Update the array index as appropriate for each light.

How it works...

Within the vertex shader, the lighting parameters are stored in the uniform array lights. Each element of the array is a struct of type `LightInfo`. This example uses five lights. The light intensity is stored in the `Intensity` field, and the position in eye coordinates is stored in the `Position` field.

The rest of the uniform variables are essentially the same as in the ADS (ambient, diffuse, and specular) shader presented in *Chapter 2*, *The Basics of GLSL Shaders*.

The `ads` function is responsible for computing the shading equation for a given light source. The index of the light is provided as the first parameter `lightIndex`. The equation is computed based on the values in the `lights` array at that index.

In the `main` function, a `for` loop is used to compute the shading equation for each light, and the results are summed into the shader output variable `Color`.

The fragment shader simply applies the interpolated color to the fragment.

See also

- The *Implementing per-vertex ambient, diffuse, and specular (ADS) shading* recipe in *Chapter 2, The Basics of GLSL shaders*
- The *Shading with a directional light source* recipe

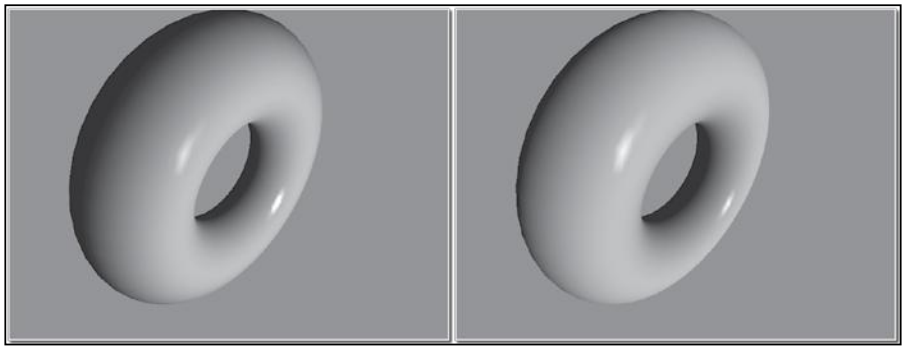
Shading with a directional light source

A core component of a shading equation is the vector that points from the surface location towards the light source (s in previous examples). For lights that are extremely far away, there is very little variation in this vector over the surface of an object. In fact, for very distant light sources, the vector is essentially the same for all points on a surface. (Another way of thinking about this is that the light rays are nearly parallel.) Such a model would be appropriate for a distant, but powerful, light source such as the sun. Such a light source is commonly called a **directional light source** because it does not have a specific position, only a direction.

Of course, we are ignoring the fact that, in reality, the intensity of the light decreases with the square of the distance from the source. However, it is not uncommon to ignore this aspect for directional light sources.

If we are using a directional light source, the direction towards the source is the same for all points in the scene. Therefore, we can increase the efficiency of our shading calculations because we no longer need to recompute the direction towards the light source for each location on the surface.

Of course, there is a visual difference between a positional light source and a directional one. The following figures show a torus rendered with a positional light (left) and a directional light (right). In the left figure, the light is located somewhat close to the torus. The directional light covers more of the surface of the torus due to the fact that all of the rays are parallel.



In previous versions of OpenGL, the fourth component of the light position was used to determine whether or not a light was considered directional. A zero in the fourth component indicated that the light source was directional and the position was to be treated as a direction towards the source (a vector). Otherwise, the position was treated as the actual location of the light source. In this example, we'll emulate the same functionality.

Getting ready

Set up your OpenGL program with the vertex position in attribute location zero, and the vertex normal in location one.

How to do it...

To create a shader program that implements ADS shading using a directional light source, use the following code:

1. Use the following vertex shader:

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

out vec3 Color;

uniform vec4 LightPosition;
uniform vec3 LightIntensity;

uniform vec3 Kd;            // Diffuse reflectivity
uniform vec3 Ka;            // Ambient reflectivity
uniform vec3 Ks;            // Specular reflectivity
uniform float Shininess;    // Specular shininess factor

uniform mat4 ModelViewMatrix;
uniform mat3 NormalMatrix;
uniform mat4 ProjectionMatrix;
uniform mat4 MVP;

vec3 ads( vec4 position, vec3 norm )
{
  vec3 s;
  if( LightPosition.w == 0.0 )
    s = normalize(vec3(LightPosition));
  else
    s = normalize(vec3(LightPosition - position));
  vec3 v = normalize(vec3(-position));
  vec3 r = reflect( -s, norm );
  return
    LightIntensity * ( Ka +
                       Kd * max( dot(s, norm), 0.0 ) +
                       Ks * pow( max( dot(r,v), 0.0 ),
                        Shininess ) );
}
```

```
void main()
{
  vec3 eyeNorm = normalize( NormalMatrix * VertexNormal);
  vec4 eyePosition = ModelViewMatrix *
                     vec4(VertexPosition,1.0);

  // Evaluate the lighting equation
  Color = ads( eyePosition, eyeNorm );
  gl_Position = MVP * vec4(VertexPosition,1.0);
}
```

2. Use the same simple fragment shader from the previous recipe:

```
in vec3 Color;

layout( location = 0 ) out vec4 FragColor;

void main() {
  FragColor = vec4(Color, 1.0);
}
```

How it works...

Within the vertex shader, the fourth coordinate of the uniform variable LightPosition is used to determine whether or not the light is to be treated as a directional light. Inside the ads function, which is responsible for computing the shading equation, the value of the vector s is determined based on whether or not the fourth coordinate of LightPosition is zero. If the value is zero, LightPosition is normalized and used as the direction towards the light source. Otherwise, LightPosition is treated as a location in eye coordinates, and we compute the direction towards the light source by subtracting the vertex position from LightPosition and normalizing the result.

There's more...

There is a slight efficiency gain when using directional lights due to the fact that there is no need to recompute the light direction for each vertex. This saves a subtraction operation, which is a small gain, but could accumulate when there are several lights, or when the lighting is computed per-fragment.

See also

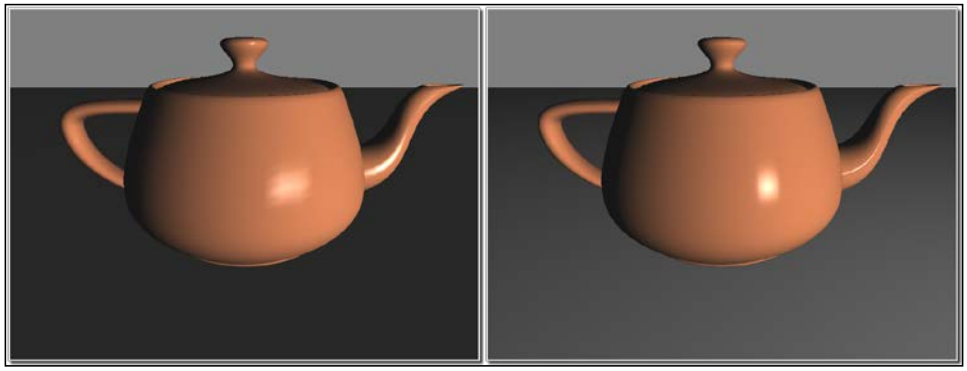
- The *Implementing per-vertex ambient, diffuse, and specular (ADS) shading* recipe in *Chapter 2*, *The Basics of GLSL Shaders*.
- The *Using per-fragment shading for improved realism* recipe

Using per-fragment shading for improved realism

When the shading equation is evaluated within the vertex shader (as we have done in previous recipes), we end up with a color associated with each vertex. That color is then interpolated across the face, and the fragment shader assigns that interpolated color to the output fragment. As mentioned previously (the *Implementing flat shading* recipe in *Chapter 2*, *The Basics of GLSL Shaders*), this technique is often called **Gouraud shading**. Gouraud shading (like all shading techniques) is an approximation, and can lead to some less than desirable results when; for example, the reflection characteristics at the vertices have little resemblance to those in the center of the polygon. For example, a bright specular highlight may reside in the center of a polygon, but not at its vertices. Simply evaluating the shading equation at the vertices would prevent the specular highlight from appearing in the rendered result. Other undesirable artifacts, such as edges of polygons, may also appear when Gouraud shading is used, due to the fact that color interpolation is less physically accurate.

To improve the accuracy of our results, we can move the computation of the shading equation from the vertex shader to the fragment shader. Instead of interpolating color across the polygon, we interpolate the position and normal vector, and use these values to evaluate the shading equation at each fragment. This technique is often called **Phong shading** or **Phong interpolation**. The results from Phong shading are much more accurate and provide more pleasing results, but some undesirable artifacts may still appear.

The following figure shows the difference between Gouraud and Phong shading. The scene on the left is rendered with Gouraud (per-vertex) shading, and on the right is the same scene rendered using Phong (per-fragment) shading. Underneath the teapot is a partial plane, drawn with a single quad. Note the difference in the specular highlight on the teapot, as well as the variation in the color of the plane beneath the teapot.



In this example, we'll implement Phong shading by passing the position and normal from the vertex shader to the fragment shader, and then evaluate the ADS shading model within the fragment shader.

Getting ready

Set up your OpenGL program with the vertex position in attribute location zero, and the normal in location one. Your OpenGL application must also provide the values for the uniform variables `Ka`, `Kd`, `Ks`, `Shininess`, `LightPosition`, and `LightIntensity`, the first four of which are the standard material properties (reflectivities) of the ADS shading model. The latter two are the position of the light in eye coordinates, and the intensity of the light source, respectively. Finally, the OpenGL application must also provide the values for the uniforms `ModelViewMatrix`, `NormalMatrix`, `ProjectionMatrix`, and `MVP`.

How to do it...

To create a shader program that can be used for implementing per-fragment (or Phong) shading using the ADS shading model, use the following steps:

1. Use the following code for the vertex shader:

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

out vec3 Position;
out vec3 Normal;

uniform mat4 ModelViewMatrix;
uniform mat3 NormalMatrix;
uniform mat4 ProjectionMatrix;
uniform mat4 MVP;

void main()
{
    Normal = normalize( NormalMatrix * VertexNormal);
    Position = vec3( ModelViewMatrix *
                     vec4(VertexPosition,1.0) );
    gl_Position = MVP * vec4(VertexPosition,1.0);
}
```

2. Use the following code for the fragment shader:

```
in vec3 Position;
in vec3 Normal;

uniform vec4 LightPosition;
uniform vec3 LightIntensity;
uniform vec3 Kd;            // Diffuse reflectivity
uniform vec3 Ka;            // Ambient reflectivity
uniform vec3 Ks;            // Specular reflectivity
uniform float Shininess;    // Specular shininess factor
```

```
layout( location = 0 ) out vec4 FragColor;

vec3 ads( )
{
  vec3 n = normalize( Normal );
  vec3 s = normalize( vec3(LightPosition) - Position );
  vec3 v = normalize(vec3(-Position));
  vec3 r = reflect( -s, n );
  return
    LightIntensity *
      ( Ka +
      Kd * max( dot(s, n), 0.0 ) +
      Ks * pow( max( dot(r,v), 0.0 ), Shininess ) );
}

void main() {
  FragColor = vec4(ads(), 1.0);
}
```

How it works...

The vertex shader has two output variables: `Position` and `Normal`. In the `main` function, we convert the vertex normal to eye coordinates by transforming with the normal matrix, and then store the converted value in `Normal`. Similarly, the vertex position is converted to eye coordinates by transforming it by the model-view matrix, and the converted value is stored in `Position`.

The values of `Position` and `Normal` are automatically interpolated and provided to the fragment shader via the corresponding input variables. The fragment shader then computes the standard ADS shading equation using the values provided. The result is then stored in the output variable, `FragColor`.

There's more...

Evaluating the shading equation within the fragment shader produces more accurate renderings. However, the price we pay is in the evaluation of the shading model for each pixel of the polygon, rather than at each vertex. The good news is that with modern graphics cards, there may be enough processing power to evaluate all of the fragments for a polygon in parallel. This can essentially provide nearly equivalent performance for either per-fragment or per-vertex shading.

See also

- The *Implementing per-vertex ambient, diffuse, and specular (ADS) shading* recipe in *Chapter 2*, *The Basics of GLSL Shaders*

Using the halfway vector for improved performance

As covered in the *Implementing per-vertex ambient, diffuse, and specular (ADS) shading* recipe in *Chapter 2*, *The Basics of GLSL Shaders*, the specular term in the ADS shading equation involves the dot product of the vector of pure reflection (**r**), and the direction towards the viewer (**v**).

$$I_s = L_s K_s (\mathbf{r} \cdot \mathbf{v})^f$$

In order to evaluate the above equation, we need to find the vector of pure reflection (**r**), which is the reflection of the vector towards the light source (**s**) about the normal vector (**n**).

$$\mathbf{r} = -\mathbf{s} + 2(\mathbf{s} \cdot \mathbf{n})\mathbf{n}$$

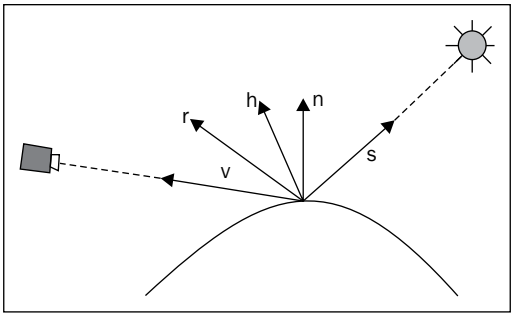
This equation is implemented by the GLSL function: `reflect`.

The above equation requires a dot product, an addition, and a couple of multiplication operations. We can gain a slight improvement in the efficiency of the specular calculation by making use of the following observation. When **v** is aligned with **r**, the normal vector (**n**) must be halfway between **v** and **s**.

Let's define the halfway vector (**h**) as the vector that is halfway between **v** and **s**, where h is normalized after the addition:

$$\mathbf{h} = \mathbf{v} + \mathbf{s}$$

The following diagram shows the relative positions of the halfway vector and the others:



We can then replace the dot product in the equation for the specular component, with the dot product of **h** and **n**.

$$I_s = L_s K_s (\mathbf{h} \cdot \mathbf{n})^f$$

Computing **h** requires fewer operations than it takes to compute **r**, so we should expect some efficiency gain by using the halfway vector. The angle between the halfway vector and the normal vector is proportional to the angle between the vector of pure reflection (**r**) and the vector towards the viewer (**v**) when all vectors are coplanar. Therefore, we expect that the visual results will be similar, although not exactly the same.

Getting ready

Start by utilizing the same shader program that was presented in the recipe, *Using per-fragment shading for improved realism*, and set up your OpenGL program as described there.

How to do it...

Using the same shader pair as in the recipe, *Using per-fragment shading for improved realism*, replace the `ads` function in the fragment shader with the following code:

```
vec3 ads( )
{
    vec3 n = normalize( Normal );
    vec3 s = normalize( vec3(LightPosition) - Position );
    vec3 v = normalize(vec3(-Position));
    vec3 h = normalize( v + s );

    return
        LightIntensity *
            (Ka +
             Kd * max( dot(s, Normal), 0.0 ) +
             Ks * pow(max(dot(h,n),0.0), Shininess ) );
}
```

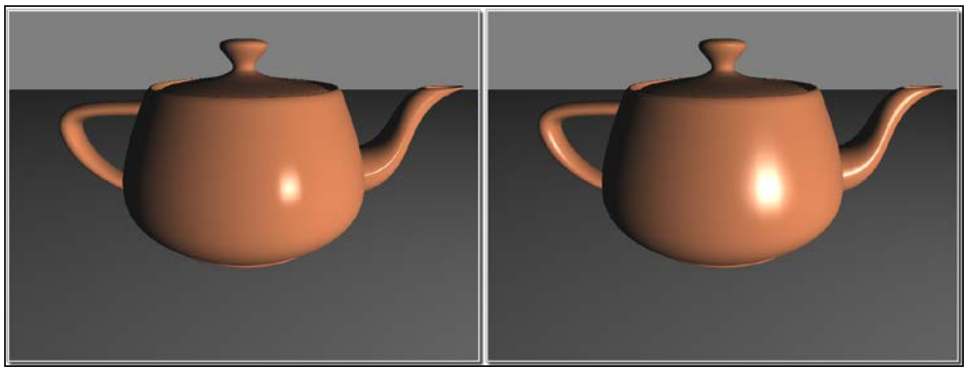
How it works...

We compute the halfway vector by summing the direction towards the viewer (`v`), and the direction towards the light source (`s`), and normalizing the result. The value for the halfway vector is then stored in `h`.

The specular calculation is then modified to use the dot product between h and the normal vector (`Normal`). The rest of the calculation is unchanged.

There's more...

The halfway vector provides a slight improvement in the efficiency of our specular calculation, and the visual results are quite similar. The following figure shows the teapot rendered using the halfway vector (right), versus the same rendering using the equation provided in the *Implementing per-vertex ambient, diffuse, and specular (ADS) shading* recipe in *Chapter 2*, *The Basics of GLSL Shaders* (left). The halfway vector produces a larger specular highlight, but the visual impact is not substantially different. If desired, we could compensate for the difference in the size of the specular highlight by increasing the value of the exponent `Shininess`.



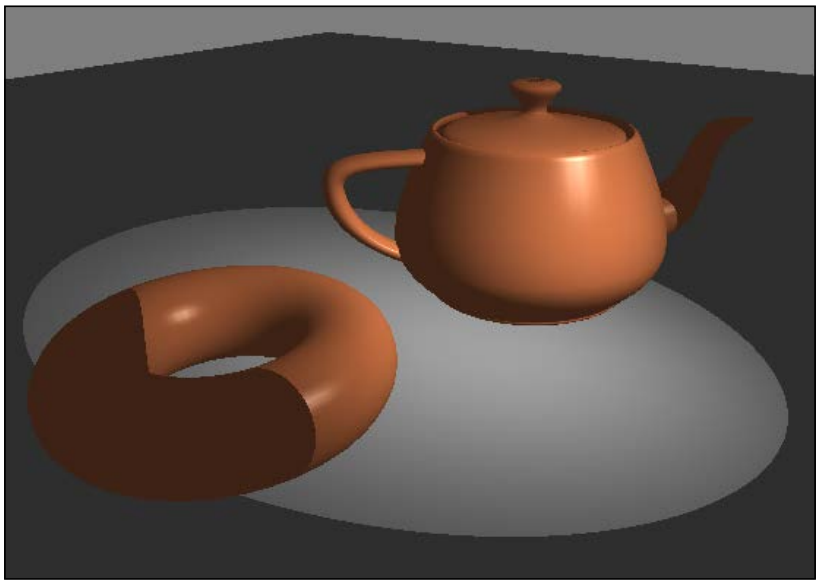
See also

- The *Using per-fragment shading for improved realism* recipe

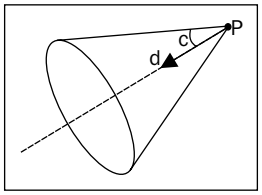
Simulating a spotlight

The fixed function pipeline had the ability to define light sources as spotlights. In such a configuration, the light source was considered to be one that only radiated light within a cone, the apex of which was located at the light source. Additionally, the light was attenuated so that it was maximal along the axis of the cone and decreased towards the outside edges. This allowed us to create light sources that had a similar visual effect to a real spotlight.

The following figure shows a teapot and a torus rendered with a single spotlight. Note the slight decrease in the intensity of the spotlight from the center towards the outside edge.



In this recipe, we'll use a shader to implement a spotlight effect similar to that produced by the fixed-function pipeline.



The spotlight's cone is defined by a spotlight direction (**d** in the preceding figure), a cutoff angle (**c** in the preceding figure), and a position (**P** in the preceding figure). The intensity of the spotlight is considered to be strongest along the axis of the cone, and decreases as you move towards the edges.

Getting ready

Start with the same vertex shader from the recipe, *Using per-fragment shading for improved realism*. Your OpenGL program must set the values for all uniform variables defined in that vertex shader as well as the fragment shader shown below.

How to do it...

To create a shader program that uses the ADS shading model with a spotlight, use the following code for the fragment shader:

```
in vec3 Position;
in vec3 Normal;

struct SpotLightInfo {
  vec4 position;  // Position in eye coords.
  vec3 intensity; // Amb., Diff., and Specular intensity
  vec3 direction; // Normalized direction of the spotlight
  float exponent; // Angular attenuation exponent
  float cutoff;   // Cutoff angle (between 0 and 90)
};
uniform SpotLightInfo Spot;

uniform vec3 Kd;            // Diffuse reflectivity
uniform vec3 Ka;            // Ambient reflectivity
uniform vec3 Ks;            // Specular reflectivity
uniform float Shininess;    // Specular shininess factor

layout( location = 0 ) out vec4 FragColor;

vec3 adsWithSpotlight( )
{
  vec3 s = normalize( vec3( Spot.position) - Position );
  float angle = acos( dot(-s, Spot.direction) );
  float cutoff = radians( clamp( Spot.cutoff, 0.0, 90.0 ) );
  vec3 ambient = Spot.intensity * Ka;

  if( angle < cutoff ) {
    float spotFactor = pow( dot(-s, Spot.direction),
                            Spot.exponent );
  vec3 v = normalize(vec3(-Position));
  vec3 h = normalize( v + s );
  return
    ambient +
    spotFactor * Spot.intensity * (
      Kd * max( dot(s, Normal), 0.0 ) +
      Ks * pow(max(dot(h,Normal), 0.0),Shininess));
  } else {
    return ambient;
    }
}

void main() {
  FragColor = vec4(adsWithSpotlight(), 1.0);
}
```

How it works...

The structure `SpotLightInfo` defines all of the configuration options for the spotlight. We declare a single uniform variable named `Spot` to store the data for our spotlight. The `position` field defines the location of the spotlight in eye coordinates. The `intensity` field is the intensity (ambient, diffuse, and specular) of the spotlight. If desired, you could break this into three variables. The `direction` field will contain the direction that the spotlight is pointing, which defines the center axis of the spotlight's cone. This vector should be specified in eye coordinates. Within the OpenGL program it should be transformed by the normal matrix in the same way that normal vectors would be transformed. We could do so within the shader; however, within the shader, the normal matrix would be specified for the object being rendered. This may not be the appropriate transform for the spotlight's direction.

The `exponent` field defines the exponent that is used when calculating the angular attenuation of the spotlight. The intensity of the spotlight is decreased in proportion to the cosine of the angle between the vector from the light to the surface location (the negation of the variable s) and the direction of the spotlight. That cosine term is then raised to the power of the variable exponent. The larger the value of this variable, the faster the intensity of the spotlight is decreased. This is quite similar to the exponent in the specular shading term.

The `cutoff` field defines the angle between the central axis and the outer edge of the spotlight's cone of light. We specify this angle in degrees, and clamp its value between 0 and 90.

The function, `adsWithSpotlight`, computes the standard ambient, diffuse, and specular (ADS) shading equation, using a spotlight as the light source. The first line computes the vector from the surface location to the spotlight's position (`s`). Next, the spotlight's direction is normalized and stored within `spotDir`. The angle between `spotDir` and the negation of `s` is then computed and stored in the variable `angle`. The `cutoff` variable stores the value of `Spot.cutoff` after it has been clamped between 0 and 90, and converted from degrees to radians. Next, the ambient lighting component is computed and stored in the `ambient` variable.

We then compare the value of the `angle` variable with that of the `cutoff` variable. If `angle` is less than `cutoff`, then the surface point is within the spotlight's cone. Otherwise the surface point only receives ambient light, so we return only the ambient component.

If `angle` is less than `cutoff`, we compute the `spotFactor` variable by raising the dot product of `-s` and `spotDir` to the power of `Spot.exponent`. The value of `spotFactor` is used to scale the intensity of the light so that the light is maximal in the center of the cone, and decreases as you move towards the edges. Finally, the ADS shading equation is computed as usual, but the diffuse and specular terms are scaled by `spotFactor`.

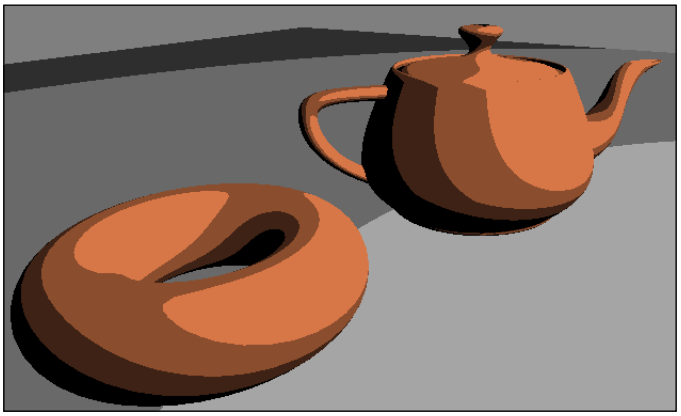
See also

- The *Using per-fragment shading for improved realism* recipe
- The *Implementing per-vertex ambient, diffuse, and specular shading (ADS)* recipe in *Chapter 2*, *The Basics of GLSL Shaders*

Creating a cartoon shading effect

Toon shading (also called **Celshading**) is a non-photorealistic technique that is intended to mimic the style of shading often used in hand-drawn animation. There are many different techniques that are used to produce this effect. In this recipe, we'll use a very simple technique that involves a slight modification to the ambient and diffuse shading model.

The basic effect is to have large areas of constant color with sharp transitions between them. This simulates the way that an artist might shade an object using strokes of a pen or brush. The following figure shows an example of a teapot and torus rendered with toon shading.



The technique presented here involves computing only the ambient and diffuse components of the typical ADS shading model, and quantizing the cosine term of the diffuse component. In other words, the value of the dot product normally used in the diffuse term is restricted to a fixed number of possible values. The following table illustrates the concept for four levels:

Cosine of the Angle between s and n	Value used
Between 1 and 0.75	0.75
Between 0.75 and 0.5	0.5
Between 0.5 and 0.25	0.25
Between 0.25 and 0.0	0.0

In the preceding table, **s** is the vector towards the light source and **n** is the normal vector at the surface. By restricting the value of the cosine term in this way, the shading displays strong discontinuities from one level to another (see the preceding figure), simulating the pen strokes of hand-drawn cel animation.

Getting ready

Start with the same vertex shader from the *Using per-fragment shading for improved realism* recipe. Your OpenGL program must set the values for all uniform variables defined in that vertex shader as well as the fragment shader code described below.

How to do it...

To create a shader program that produces a toon shading effect, use the following fragment shader:

```
in vec3 Position;
in vec3 Normal;

struct LightInfo {
  vec4 position;
  vec3 intensity;
};
uniform LightInfo Light;

uniform vec3 Kd;            // Diffuse reflectivity
uniform vec3 Ka;            // Ambient reflectivity

const int levels = 3;
const float scaleFactor = 1.0 / levels;
layout( location = 0 ) out vec4 FragColor;
vec3 toonShade( )
{
  vec3 s = normalize( Light.position.xyz - Position.xyz );
  float cosine = max( 0.0, dot( s, Normal ) );
  vec3 diffuse = Kd * floor( cosine * levels ) *
                 scaleFactor;

  return Light.intensity * (Ka + diffuse);
}

void main() {
  FragColor = vec4(toonShade(), 1.0);
}
```

How it works...

The constant variable, `levels`, defines how many distinct values will be used in the diffuse calculation. This could also be defined as a uniform variable to allow for configuration from the main OpenGL application. We will use this variable to quantize the value of the cosine term in the diffuse calculation.

The `toonShade` function is the most significant part of this shader. We start by computing `s`, the vector towards the light source. Next, we compute the cosine term of the diffuse component by evaluating the dot product of `s` and `Normal`. The next line quantizes that value in the following way. Since the two vectors are normalized, and we have removed negative values with the max function, we are sure that the value of cosine is between zero and one. By multiplying this value by levels and taking the floor, the result will be an integer between 0 and levels –1. When we divide that value by levels (by multiplying by `scaleFactor`), we scale these integral values to be between zero and one again. The result is a value that can be one of `levels` possible values spaced between zero and one. This result is then multiplied by `Kd`, the diffuse reflectivity term.

Finally, we combine the diffuse and ambient components together to get the final color for the fragment.

There's more...

When quantizing the cosine term, we could have used `ceil` instead of `floor`. Doing so would have simply shifted each of the possible values up by one level. This would make the levels of shading slightly brighter.

The typical cartoon style seen in most cel animation includes black outlines around the silhouettes and along other edges of a shape. The shading model presented here does not produce those black outlines. There are several techniques for producing them, and we'll look at one later on in this book.

See also

- The *Using per-fragment shading for improved realism* recipe
- The *Implementing per-vertex ambient, diffuse, and specular (ADS) shading* recipe in *Chapter 2*, *The Basics of GLSL Shaders*
- The *Drawing silhouette lines using the geometry shader* recipe in *Chapter 6*, *Using Geometry and Tessellation Shaders*

Simulating fog

A simple fog effect can be achieved by mixing the color of each fragment with a constant fog color. The amount of influence of the fog color is determined by the distance from the camera. We could use either a linear relationship between the distance and the amount of fog color, or we could use a non-linear relationship such as an exponential one.

The following figure shows four teapots rendered with a fog effect produced by mixing the fog color in a linear relationship with distance.

To define this linear relationship we can use the following equation:

$$f = \frac{d_{\max} - |z|}{d_{\max} - d_{\min}}$$

In the preceding equation, d_{min} is the distance from the eye where the fog is minimal (no fog contribution), and d_{max} is the distance where the fog color obscures all other colors in the scene. The variable `z` represents the distance from the eye. The value `f` is the fog factor. A fog factor of zero represents 100 percent fog, and a factor of one represents no fog. Since fog typically looks thickest at large distances, the fog factor is minimal when `|z|` is equal to d_{max}, and maximal when `|z|` is equal to d_{min}.

Since the fog is applied by the fragment shader, the effect will only be visible on the objects that are rendered. It will not appear on any "empty" space in the scene (the background). To help make the fog effect consistent, you should use a background color that matches the maximum fog color.

Getting ready

Start with the same vertex shader from the *Using per-fragment shading for improved realism* recipe. Your OpenGL program must set the values for all uniform variables defined in that vertex shader as well as the fragment shader shown in the following section.

How to do it...

To create a shader that produces a fog-like effect, use the following code for the fragment shader:

```
in vec3 Position;
in vec3 Normal;

struc tLightInfo {
  vec4 position;
  vec3 intensity;
};
uniform LightInfo Light;

struct FogInfo {
  float maxDist;
  float minDist;
  vec3 color;
};
uniform FogInfo Fog;

uniform vec3 Kd;            // Diffuse reflectivity
uniform vec3 Ka;            // Ambient reflectivity
uniform vec3 Ks;            // Specular reflectivity
uniform float Shininess;    // Specular shininess factor

layout( location = 0 ) out vec4 FragColor;

vec3 ads( )
{
   // … The ADS shading algorithm
}

void main() {
  float dist = abs( Position.z );
  float fogFactor = (Fog.maxDist - dist) /
                    (Fog.maxDist - Fog.minDist);
  fogFactor = clamp( fogFactor, 0.0, 1.0 );
  vec3 shadeColor = ads();
  vec3 color = mix( Fog.color, shadeColor, fogFactor );

  FragColor = vec4(color, 1.0);
}
```

How it works...

In this shader, the `ads` function is exactly the same as the one used in the recipe *Using the halfway vector for improved performance*. The part of this shader that deals with the fog effect lies within the `main` function.

The uniform variable `Fog` contains the parameters that define the extent and color of the fog. The `minDist` field is the distance from the eye to the fog's starting point, and `maxDist` is the distance to the point where the fog is maximal. The `color` field is the color of the fog.

The `dist` variable is used to store the distance from the surface point to the eye position. The `z` coordinate of the position is used as an estimate of the actual distance. The `fogFactor` variable is computed using the preceding equation. Since `dist` may not be between `minDist` and `maxDist`, we clamp the value of `fogFactor` to be between zero and one.

We then call the `ads` function to evaluate the basic ADS shading model. The result of this is stored in the `shadeColor` variable.

Finally, we mix `shadeColor` and `Fog.color` together based on the value of `fogFactor`, and the result is used as the fragment color.

There's more...

In this recipe, we used a linear relationship between the amount of fog color and the distance from the eye. Another choice would be to use an exponential relationship. For example, the following equation could be used:

$$f = e^{-d|z|}$$

In the above equation, **d** represents the density of the fog. Larger values would create "thicker" fog. We could also square the exponent to create a slightly different relationship (a faster increase in the fog with distance).

$$f = e^{-(dz)^2}$$

Computing distance from the eye

In the above code, we used the absolute value of the z coordinate as the distance from the camera. This may cause the fog to look a bit unrealistic in certain situations. To compute a more precise distance, we could replace the line:

```
float dist = abs( Position.z );
```

with the following:

```
float dist = length( Position.xyz );
```

Of course, the latter version requires a square root, and therefore would be a bit slower in practice.

See also

- The *Using per-fragment shading for improved realism* recipe
- The *Implementing per-vertex ambient, diffuse, and specular (ADS) shading* recipe in *Chapter 2*, *The Basics of GLSL Shaders*

Configuring the depth test

GLSL 4 provides the ability to configure how the depth test is performed. This gives us additional control over how and when fragments are tested against the depth buffer.

Many OpenGL implementations automatically provide an optimization known as the early depth test or early fragment test. With this optimization, the depth test is performed before the fragment shader is executed. Since fragments that fail the depth test will not appear on the screen (or the framebuffer), there is no point in executing the fragment shader at all for those fragments and we can save some time by avoiding the execution.

The OpenGL specification, however, states that the depth test is performed *after* the fragment shader. This means that if an implementation wishes to use the early depth test optimization, it must be careful. The implementation must make sure that if anything within the fragment shader might change the results of the depth test, then it should avoid using the early depth test.

For example, a fragment shader can change the depth of a fragment by writing to the output variable, `gl_FragDepth`. If it does so, then the early depth test cannot be performed because, of course, the final depth of the fragment is not known prior to the execution of the fragment shader. However, the GLSL provides ways to notify the pipeline roughly how the depth will be modified, so that the implementation may determine when it might be ok to use the early depth test.

Another possibility is that the fragment shader might conditionally discard the fragment using the `discard` keyword. If there is any possibility that the fragment may be discarded, some implementations may not perform the early depth test.

There are also certain situations where we want to rely on the early depth test. For example, if the fragment shader writes to memory other than the framebuffer (with image load/store, shader storage buffers, or other incoherent memory writing), we might not want the fragment shader to execute for fragments that fail the depth test. This would help us to avoid writing data for fragments that fail. The GLSL provides a technique for forcing the early depth test optimization.

How to do it...

To ask the OpenGL pipeline to always perform the early depth test optimization, use the following layout qualifier in your fragment shader:

```
layout(early_fragment_tests) in;
```

If your fragment shader will modify the fragment's depth, but you still would like to take advantage of the early depth test when possible, use the following layout qualifier in a declaration of `gl_FragDepth` within your fragment shader:

```
layout (depth_*) out float gl_FragDepth;
```

Where, `depth_*` is one of the following: `depth_any`, `depth_greater`, `depth_less`, or `depth_unchanged`.

How it works...

The following statement forces the OpenGL implementation to always perform the early depth test:

```
layout(early_fragment_tests) in;
```

We must keep in mind that if we attempt to modify the depth anywhere within the shader by writing to `gl_FragDepth`, the value that is written will be ignored.

If your fragment shader needs to modify the depth value, then we can't force early fragment tests. However, we can help the pipeline to determine when it can still apply the early test. We do so by using one of the layout qualifiers for `gl_FragDepth` as shown above. This places some limits on how the value will be modified. The OpenGL implementation can then determine if the fragment shader can be skipped. If it can be determined that the depth will not be changed in such a way that it would cause the result of the test to change, the implementation can still use the optimization.

The layout qualifier for the output variable `gl_FragDepth` tells the OpenGL implementation specifically how the depth might change within the fragment shader. The qualifier `depth_any` indicates that it could change in any way. This is the default.

The other qualifiers describe how the value may change with respect to `gl_FragCoord.z`.

- `depth_greater`: This fragment shader promises to only increase the depth.
- `depth_less`: This fragment shader promises to only decrease the depth.
- `depth_unchanged`: This fragment shader promises not to change the depth. If it writes to `gl_FragDepth`, the value will be equal to `gl_FragCoord.z`.

If you use one of these qualifiers, but then go on to modify the depth in an incompatible way, the results are undefined. For example, if you declare `gl_FragDepth` with `depth_greater`, but decrease the depth of the fragment, the code will compile and execute, but you shouldn't expect to see accurate results.

If your fragment shader writes to `gl_FragDepth`, then it must be sure to write a value in all circumstances. In other words, it must write a value no matter which branches are taken within the code.

See also

- The *Implementing order-independent transparency* recipe in *Chapter 5*, *Image Processing and Screen Space Techniques*