OpenGL 4.0 Shading Language Cookbook

Chapter No. 2
"The Basics of GLSL Shaders"
In this package, you will find:
A Biography of the author of the book
A preview chapter from the book, Chapter NO.2 "The Basics of GLSL Shaders"
A synopsis of the book’s content
Information on where to buy this book

About the Author

David Wolff is an associate professor in the Computer Science and Computer Engineering Department at Pacific Lutheran University (PLU). He received his PhD in Physics from Oregon State University. He has a passion for computer graphics and the intersection between art and science. He has been teaching computer graphics to undergraduates at PLU for over 10 years, using OpenGL.

Special thanks to Brandon Whitley for interesting discussions and helpful insights during the writing of this book. His help has been incredibly valuable. Thanks also to all of the reviewers and editors for their help.

I’d also like to thank my parents for a lifetime of support, love and encouragement.

For More Information:
www.packtpub.com/opengl-4-0-shading-language-cookbook/book
The OpenGL Shading Language (GLSL) Version 4.0 brings unprecedented power and flexibility to programmers interested in creating modern, interactive, graphical programs. It allows us to harness the power of modern Graphics Processing Units (GPUs) in a straightforward way by providing a simple, yet powerful, language and API.

The OpenGL 4.0 Shading Language Cookbook will provide easy-to-follow examples that start by walking you through the theory and background behind each technique. It then goes on to provide and explain the GLSL and OpenGL code needed to implement them. Beginning through to advanced techniques are presented, including topics such as texturing, screen-space techniques, lighting, shading, tessellation shaders, geometry shaders, and shadows.

What This Book Covers

Chapter 1, Getting Started with GLSL 4.0, provides tips and tricks for setting up your OpenGL development environment to take advantage of the latest OpenGL and GLSL language features. It also teaches the basic techniques for communicating with shader programs.

Chapter 2, The Basics of GLSL Shaders, provides examples of basic shading techniques such as diffuse shading, two-sided shading, and flat shading. It also discuses an example of a new 4.0 language feature: subroutines.

Chapter 3, Lighting and Shading Effects and Optimizations, provides examples of more complex lighting and shading such as multiple lights, per-fragment shading, spotlights, cartoon shading, and fog. It moves further to explain how to gain a slight increase in execution speed by using the halfway vector or a directional light source.

Chapter 4, Using Textures, provides a variety of examples illustrating how textures can be used in GLSL shaders. It also explores examples involving simple 2D textures, multiple textures, normal maps, alpha maps, cube maps, and projected textures. It also discusses how to render to a texture using framebuffer objects.

Chapter 5, Image Processing and Screen Space Techniques, discusses various techniques to apply post-processing effects such as bloom, blur, and edge detection. It also discusses an example of a very popular rendering technique known as deferred shading.

For More Information:
www.packtpub.com/opengl-4-0-shading-language-cookbook/book
Chapter 6, *Using Geometry and Tessellation Shaders*, provides a series of examples to introduce you to the new and powerful segments of the shader pipeline. It provides some examples of geometry shaders, and discusses how to use tessellation shaders to dynamically render geometry at different levels of detail.

Chapter 7, *Shadows*, provides several recipes surrounding the shadow-mapping algorithm. It also discusses some basic and advanced techniques for producing shadows, focusing mainly on texture-based shadow maps.

Chapter 8, *Using Noise in Shaders*, provides recipes that demonstrate how to make use of a pre-computed noise texture to create a variety of effects. The first two recipes demonstrate how to generate a noise texture using the free, open-source library libnoise. Then, it moves on to explain several examples that use noise textures to produce natural and artificial effects such as wood grain, clouds, electrical interference, splattering, and erosion.

Chapter 9, *Animation and Particles*, discusses several examples of animation within shaders, focusing mostly on particle systems. It also provides an example illustrating how to use OpenGL’s transform feedback functionality within a particle system. The last two recipes in the chapter demonstrate some particle systems for simulating complex real systems, such as smoke and fire.

For More Information:
www.packtpub.com/opengl-4-0-shading-language-cookbook/book
The Basics of GLSL Shaders

In this chapter, we will cover:

- Implementing diffuse, per-vertex shading with a single point light source
- Implementing per-vertex ambient, diffuse, and specular (ADS) shading
- Using functions in shaders
- Implementing two sided shading
- Implementing flat shading
- Using subroutines to select shader functionality
- Discarding fragments to create a perforated look

Introduction

Shaders were first introduced into OpenGL in version 2.0, introducing programmability into the formerly fixed-function OpenGL pipeline. Shaders give us the power to implement alternative rendering algorithms and a greater degree of flexibility in the implementation of those techniques. With shaders, we can run custom code directly on the GPU, providing us with the opportunity to leverage the high degree of parallelism available with modern GPUs.

For More Information:
www.packtpub.com/opengl-4-0-shading-language-cookbook/book
Shaders are implemented using the OpenGL Shading Language (GLSL). The GLSL is syntactically similar to C, which should make it easier for experienced OpenGL programmers to learn. Due to the nature of this text, I won't present a thorough introduction to GLSL here. Instead, if you're new to GLSL, reading through these recipes should help you to learn the language by example. If you are already comfortable with GLSL, but don't have experience with version 4.0, you'll see how to implement these techniques utilizing the newer API. However, before we jump into GLSL programming, let's take a quick look at how vertex and fragment shaders fit within the OpenGL pipeline.

**Vertex and fragment shaders**

In OpenGL version 4.0, there are five shader stages: vertex, geometry, tessellation control, tessellation evaluation, and fragment. In this chapter we'll focus only on the vertex and fragment stages. In Chapter 6, I'll provide some recipes for working with the geometry and tessellation shaders.

Shaders replace parts of the OpenGL pipeline. More specifically, they make those parts of the pipeline programmable. The following block diagram shows a simplified view of the OpenGL pipeline with only the vertex and fragment shaders installed.

Vertex data is sent down the pipeline and arrives at the vertex shader via shader input variables. The vertex shader's input variables correspond to vertex attributes (see Chapter 1, *Sending data to a shader using per-vertex attributes and vertex buffer objects*). In general, a shader receives its input via programmer-defined input variables, and the data for those variables comes either from the main OpenGL application or previous pipeline stages (other shaders). For example, a fragment shader's input variables might be fed from the output variables of the vertex shader. Data can also be provided to any shader stage using uniform variables (see Chapter 1: *Sending data to a shader using uniform variables*). These are used for information that changes less often than vertex attributes (for example, matrices, light position, and other settings). The following figure shows a simplified view of the relationships between input and output variables when there are two shaders active (vertex and fragment).
The vertex shader is executed once for each vertex, possibly in parallel. The data corresponding to vertex position must be transformed into clip coordinates and assigned to the output variable `gl_Position` before the vertex shader finishes execution. The vertex shader can send other information down the pipeline using shader output variables. For example, the vertex shader might also compute the color associated with the vertex. That color would be passed to later stages via an appropriate output variable.

Between the vertex and fragment shader, the vertices are assembled into primitives, clipping takes place, and the viewport transformation is applied (among other operations). The rasterization process then takes place and the polygon is filled (if necessary). The fragment shader is executed once for each fragment (pixel) of the polygon being rendered (typically in parallel). Data provided from the vertex shader is (by default) interpolated in a perspective correct manner, and provided to the fragment shader via shader input variables. The fragment shader determines the appropriate color for the pixel and sends it to the frame buffer using output variables. The depth information is handled automatically.

**Replicating the old fixed functionality**

Programmable shaders give us tremendous power and flexibility. However, in some cases we might just want to re-implement the basic shading techniques that were used in the default fixed-function pipeline, or perhaps use them as a basis for other shading techniques. Studying the basic shading algorithm of the old fixed-function pipeline can also be a good way to get started when learning about shader programming.

In this chapter, we'll look at the basic techniques for implementing shading similar to that of the old fixed-function pipeline. We'll cover the standard ambient, diffuse, and specular (ADS) shading algorithm, the implementation of two-sided rendering, and flat shading. Along the way, we'll also see some examples of other GLSL features such as functions, subroutines, and the `discard` keyword.

For More Information:

www.packtpub.com/opengl-4-0-shading-language-cookbook/book
The algorithms presented within this chapter are largely unoptimized. I present them this way to avoid additional confusion for someone who is learning the techniques for the first time. We'll look at a few optimization techniques at the end of some recipes, and some more in the next chapter.

Implementing diffuse, per-vertex shading with a single point light source

One of the simplest shading techniques is to assume that the surface exhibits purely diffuse reflection. That is to say that the surface is one that appears to scatter light in all directions equally, regardless of direction. Incoming light strikes the surface and penetrates slightly before being re-radiated in all directions. Of course, the incoming light interacts with the surface before it is scattered, causing some wavelengths to be fully or partially absorbed and others to be scattered. A typical example of a diffuse surface is a surface that has been painted with a matte paint. The surface has a dull look with no shine at all.

The following image shows a torus rendered with diffuse shading.

The mathematical model for diffuse reflection involves two vectors: the direction from the surface point to the light source (s), and the normal vector at the surface point (n). The vectors are represented in the following diagram.

For More Information:
www.packtpub.com/opengl-4-0-shading-language-cookbook/book
The amount of incoming light (or radiance) that reaches the surface is partially dependent on the orientation of the surface with respect to the light source. The physics of the situation tells us that the amount of radiation that reaches a point on a surface is maximal when the light arrives along the direction of the normal vector, and zero when the light is perpendicular to the normal. In between, it is proportional to the cosine of the angle between the direction towards the light source and the normal vector. So, since the dot product is proportional to the cosine of the angle between two vectors, we can express the amount of radiation striking the surface as the product of the light intensity and the dot product of $s$ and $n$.

$$ L_d \ s \cdot n $$

Where $L_d$ is the intensity of the light source, and the vectors $s$ and $n$ are assumed to be normalized. You may recall that the dot product of two unit vectors is equal to the cosine of the angle between them.

As stated previously, some of the incoming light is absorbed before it is re-emitted. We can model this interaction by using a reflection coefficient ($K_d$), which represents the fraction of the incoming light that is scattered. This is sometimes referred to as the diffuse reflectivity, or the diffuse reflection coefficient. The diffuse reflectivity becomes a scaling factor for the incoming radiation, so the intensity of the outgoing light can be expressed as follows:

$$ L = L_d \ K_d \ s \cdot n $$

Because this model depends only on the direction towards the light source and the normal to the surface, not on the direction towards the viewer, we have a model that represents uniform (omnidirectional) scattering.

In this recipe, we'll evaluate this equation at each vertex in the vertex shader and interpolate the resulting color across the face.

For More Information:  
www.packtpub.com/opengl-4-0-shading-language-cookbook/book
In this and the following recipes, light intensities and material reflectivity coefficients are represented by 3-component (RGB) vectors. Therefore, the equations should be treated as component-wise operations, applied to each of the three components separately. Luckily, the GLSL will make this nearly transparent because the needed operators will operate component-wise on vector variables.

**Getting ready**

Start with an OpenGL application that provides the vertex position in attribute location 0, and the vertex normal in attribute location 1 (see *Chapter 1, Sending data to a shader using per-vertex attributes and vertex buffer objects*). The OpenGL application also should provide the standard transformation matrices (projection, modelview, and normal) via uniform variables.

The light position (in eye coordinates), $K_d$, and $L_d$ should also be provided by the OpenGL application via uniform variables. Note that $K_d$ and $L_d$ are type `vec3`. We can use a `vec3` to store an RGB color as well as a vector or point.

**How to do it...**

To create a shader pair that implements diffuse shading, use the following code:

1. Use the following code for the vertex shader.

```cpp
#version 400
layout (location = 0) in vec3 VertexPosition;
layout (location = 1) in vec3 VertexNormal;
out vec3 LightIntensity;
uniform vec4 LightPosition; // Light position in eye coords.
uniform vec3 Kd;           // Diffuse reflectivity
uniform vec3 Ld;            // Light source intensity
uniform mat4 ModelViewMatrix;
uniform mat3 NormalMatrix;
uniform mat4 ProjectionMatrix;
uniform mat4 MVP;           // Projection * ModelView
void main()
{
    // Convert normal and position to eye coords
    vec3 tnorm = normalize( NormalMatrix * VertexNormal);
    vec4 eyeCoords = ModelViewMatrix *
        vec4(VertexPosition,1.0));
```

For More Information:

www.packtpub.com/opengl-4-0-shading-language-cookbook/book
vec3 s = normalize(vec3(LightPosition - eyeCoords));

// The diffuse shading equation
LightIntensity = Ld * Kd * max( dot( s, tnorm ), 0.0 );

// Convert position to clip coordinates and pass along
gl_Position = MVP * vec4(VertexPosition, 1.0);
}

2. Use the following code for the fragment shader.

```glsl
#version 400
in vec3 LightIntensity;
layout( location = 0 ) out vec4 FragColor;
void main() {
  FragColor = vec4(LightIntensity, 1.0);
}
```

3. Compile and link both shaders within the OpenGL application, and install the shader program prior to rendering. See Chapter 1 for details about compiling, linking, and installing shaders.

**How it works...**

The vertex shader does all of the work in this example. The diffuse reflection is computed in eye coordinates by first transforming the normal vector using the normal matrix, normalizing, and storing the result in $tnorm$. Note that the normalization here may not be necessary if your normal vectors are already normalized and the normal matrix does not do any scaling.

The next step converts the vertex position to eye (camera) coordinates by transforming it via the model-view matrix. Then we compute the direction towards the light source by subtracting the vertex position from the light position and storing the result in $s$.

For More Information:
www.packtpub.com/opengl-4-0-shading-language-cookbook/book
Next, we compute the scattered light intensity using the equation described above and store the result in the output variable `LightIntensity`. Note the use of the `max` function here. If the dot product is less than zero, then the angle between the normal vector and the light direction is greater than 90 degrees. This means that the incoming light is coming from inside the surface. Since such a situation is not physically possible (for a closed mesh), we use a value of 0.0. However, you may decide that you want to properly light both sides of your surface, in which case the normal vector needs to be reversed for those situations where the light is striking the back side of the surface (see Implementing two-sided shading).

Finally, we convert the vertex position to clip coordinates by multiplying with the model-view projection matrix, (which is: `projection * view * model`) and store the result in the built-in output variable `gl_Position`.

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

The subsequent stage of the OpenGL pipeline expects that the vertex position will be provided in clip coordinates in the output variable `gl_Position`. This variable does not directly correspond to any input variable in the fragment shader, but is used by the OpenGL pipeline in the primitive assembly, clipping, and rasterization stages that follow the vertex shader. It is important that we always provide a valid value for this variable.

Since `LightIntensity` is an output variable from the vertex shader, its value is interpolated across the face and passed into the fragment shader. The fragment shader then simply assigns the value to the output fragment.

**There's more...**

Diffuse shading is a technique that models only a very limited range of surfaces. It is best used for surfaces that have a "matte" appearance. Additionally, with the technique used above, the dark areas may look a bit too dark. In fact, those areas that are not directly illuminated are completely black. In real scenes, there is typically some light that has been reflected about the room that brightens these surfaces. In the following recipes, we'll look at ways to model more surface types, as well as provide some light for those dark parts of the surface.

**See also**

- Chapter 1, Sending data to a shader using uniform variables
- Chapter 1, Compiling a shader
- Chapter 1, Linking a shader program
- Chapter 1, Sending data to a shader using per-vertex attributes and vertex buffer objects

**For More Information:**

www.packtpub.com/opengl-4-0-shading-language-cookbook/book
Implementing per-vertex ambient, diffuse, and specular (ADS) shading

The OpenGL fixed function pipeline implemented a default shading technique which is very similar to the one presented here. It models the light-surface interaction as a combination of three components: ambient, diffuse, and specular. The **ambient** component is intended to model light that has been reflected so many times that it appears to be emanating uniformly from all directions. The **diffuse** component was discussed in the previous recipe, and represents omnidirectional reflection. The **specular** component models the shininess of the surface and represents reflection around a preferred direction. Combining these three components together can model a nice (but limited) variety of surface types. This shading model is also sometimes called the **Phong reflection model** (or **Phong shading model**), after Bui Tuong Phong.

An example of a torus rendered with the ADS shading model is shown in the following screenshot:

![Torus rendered with ADS shading](image)

The ADS model is implemented as the sum of the three components: ambient, diffuse, and specular. The ambient component represents light that illuminates all surfaces equally and reflects equally in all directions. It is often used to help brighten some of the darker areas within a scene. Since it does not depend on the incoming or outgoing directions of the light, it can be modeled simply by multiplying the light source intensity ($L_a$) by the surface reflectivity ($K_a$).

$$I_a = L_a K_a$$

For More Information:
www.packtpub.com/opengl-4-0-shading-language-cookbook/book
The Basics of GLSL Shaders

The diffuse component models a rough surface that scatters light in all directions (see *Implementing diffuse per-vertex shading with a single point light source*). The intensity of the outgoing light depends on the angle between the surface normal and the vector towards the light source.

\[ I_d = L_d K_d (s \cdot n) \]

The specular component is used for modeling the shininess of a surface. When a surface has a glossy shine to it, the light is reflected off of the surface in a mirror-like fashion. The reflected light is strongest in the direction of perfect (mirror-like) reflection. The physics of the situation tells us that for perfect reflection, the angle of incidence is the same as the angle of reflection and that the vectors are coplanar with the surface normal, as shown in the following diagram:

![Diagram showing diffuse and specular components](image.png)

In the preceding diagram, \( r \) represents the vector of pure-reflection corresponding to the incoming light vector \((-s)\), and \( n \) is the surface normal. We can compute \( r \) by using the following equation:

\[ r = -s + 2(s \cdot n)n \]

To model specular reflection, we need to compute the following (normalized) vectors: the direction towards the light source \((s)\), the vector of perfect reflection \((r)\), the vector towards the viewer \((v)\), and the surface normal \((n)\). These vectors are represented in the following image:

For More Information:

We would like the reflection to be maximal when the viewer is aligned with the vector $r$, and to fall off quickly as the viewer moves further away from alignment with $r$. This can be modeled using the cosine of the angle between $v$ and $r$ raised to some power ($f$).

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

(Recall that the dot product is proportional to the cosine of the angle between the vectors involved.) The larger the power, the faster the value drops towards zero as the angle between $v$ and $r$ increases. Again, similar to the other components, we also introduce a specular light intensity term ($L_s$) and reflectivity term ($K_s$).

The specular component creates **specular highlights** (bright spots) that are typical of glossy surfaces. The larger the power of $f$ in the equation, the smaller the specular highlight and the shinier the surface appears. The value for $f$ is typically chosen to be somewhere between 1 and 200.

Putting all of this together, we have the following shading equation:

$$I = I_a + I_d + I_s$$
$$= L_a K_a + L_d K_d (s \cdot n) + L_s K_s (r \cdot v)^f$$

For more details about how this shading model was implemented in the fixed function pipeline, take a look at Chapter 5, *Image Processing and Screen Space Techniques*.

In the following code, we'll evaluate this equation in the vertex shader, and interpolate the color across the polygon.

For More Information:
www.packtpub.com/opengl-4-0-shading-language-cookbook/book
Getting ready

In the OpenGL application, provide the vertex position in location 0 and the vertex normal in location 1. The light position and the other configurable terms for our lighting equation are uniform variables in the vertex shader and their values must be set from the OpenGL application.

How to do it...

To create a shader pair that implements ADS shading, use the following code:

1. Use the following code for the vertex shader:

```glsl
#version 400
layout (location = 0) in vec3 VertexPosition;
layout (location = 1) in vec3 VertexNormal;
out vec3 LightIntensity;
struct LightInfo {
    vec4 Position; // Light position in eye coords.
    vec3 La;       // Ambient light intensity
    vec3 Ld;       // Diffuse light intensity
    vec3 Ls;       // Specular light intensity
};
uniform LightInfo Light;
struct MaterialInfo {
    vec3 Ka;            // Ambient reflectivity
    vec3 Kd;            // Diffuse reflectivity
    vec3 Ks;            // Specular reflectivity
    float Shininess;    // Specular shininess factor
};
uniform MaterialInfo Material;
uniform mat4 ModelViewMatrix;
uniform mat3 NormalMatrix;
uniform mat4 ProjectionMatrix;
uniform mat4 MVP;
void main()
{
    vec3 tnorm = normalize( NormalMatrix * VertexNormal);
    vec4 eyeCoords = ModelViewMatrix *
        vec4(VertexPosition,1.0);
    vec3 s = normalize(vec3(Light.Position - eyeCoords));
    vec3 v = normalize(-eyeCoords.xyz);
```
vec3 r = reflect( -s, tnorm );
vec3 ambient = Light.La * Material.Ka;
float sDotN = max( dot(s,tnorm), 0.0 );
vec3 diffuse = Light.Ld * Material.Kd * sDotN;
vec3 spec = vec3(0.0);
if( sDotN > 0.0 )
    spec = Light.Ls * Material.Ks * 
    pow( max( dot(r,v), 0.0 ), Material.Shininess );
LightIntensity = ambient + diffuse + spec;
gl_Position = MVP * vec4(VertexPosition,1.0);
}

2. Use the following code for the fragment shader:

```glsl
#version 400
in vec3 LightIntensity;
layout( location = 0 ) out vec4 FragColor;
void main() {
    FragColor = vec4(LightIntensity, 1.0);
}
```

3. Compile and link both shaders within the OpenGL application, and install the shader program prior to rendering.

---

**How it works...**

The vertex shader computes the shading equation in eye coordinates. It begins by transforming the vertex normal into eye coordinates and normalizing, then storing the result in `tnorm`. The vertex position is then transformed into eye coordinates and stored in `eyeCoords`.

Next, we compute the normalized direction towards the light source (`s`). This is done by subtracting the vertex position in eye coordinates from the light position and normalizing the result.

The direction towards the viewer (`v`) is the negation of the position (normalized) because in eye coordinates the viewer is at the origin.

We compute the direction of pure reflection by calling the GLSL built-in function `reflect`, which reflects the first argument about the second. We don't need to normalize the result because the two vectors involved are already normalized.

---

For More Information:

www.packtpub.com/opengl-4-0-shading-language-cookbook/book
The ambient component is computed and stored in the variable \textit{ambient}. The dot product of \textit{s} and \textit{n} is computed next. As in the preceding recipe, we use the built-in function \textit{max} to limit the range of values to between one and zero. The result is stored in the variable named \textit{sDotN}, and is used to compute the diffuse component. The resulting value for the diffuse component is stored in the variable \textit{diffuse}. Before computing the specular component, we check the value of \textit{sDotN}. If \textit{sDotN} is zero, then there is no light reaching the surface, so there is no point in computing the specular component, as its value must be zero. Otherwise, if \textit{sDotN} is greater than zero, we compute the specular component using the equation presented earlier. Again, we use the built-in function \textit{max} to limit the range of values of the dot product to between one and zero, and the function \textit{pow} raises the dot product to the power of the \textit{Shininess} exponent (corresponding to \textit{f} in our lighting equation).

If we did not check \textit{sDotN} before computing the specular component, it is possible that some specular highlights could appear on faces that are facing away from the light source. This is clearly a non-realistic and undesirable result. Some people solve this problem by multiplying the specular component by the diffuse component, which would decrease the specular component substantially and alter its color. The solution presented here avoids this, at the cost of a branch statement (the \textit{if} statement).

The sum of the three components is then stored in the output variable \textit{LightIntensity}. This value will be associated with the vertex and passed down the pipeline. Before reaching the fragment shader, its value will be interpolated in a perspective correct manner across the face of the polygon.

Finally, the vertex shader transforms the position into clip coordinates, and assigns the result to the built-in output variable \textit{gl\_Position} (see \textit{Implementing diffuse, per-vertex shading with a single point light source}).

The fragment shader simply applies the interpolated value of \textit{LightIntensity} to the output fragment by storing it in the shader output variable \textit{FragColor}.

\textbf{There's more...}

This version of the ADS (Ambient, Diffuse, and Specular) reflection model is by no means optimal. There are several improvements that could be made. For example, the computation of the vector of pure reflection can be avoided via the use of the so-called "halfway vector". This is discussed in Chapter 3, \textit{Using the halfway vector for improved performance}.

\textbf{For More Information:}

www.packtpub.com/opengl-4-0-shading-language-cookbook/book
Using a non-local viewer

We can avoid the extra normalization needed to compute the vector towards the viewer (v), by using a so-called non-local viewer. Instead of computing the direction towards the origin, we simply use the constant vector (0, 0, 1) for all vertices. This is similar to assuming that the viewer is located infinitely far away in the z direction. Of course, it is not accurate, but in practice the visual results are very similar, often visually indistinguishable, saving us normalization.

In the old fixed-function pipeline, the non-local viewer was the default, and could be adjusted (turned on or off) using the function glLightModel.

Per-vertex vs. Per-fragment

Since the shading equation is computed within the vertex shader, we refer to this as per-vertex lighting. One of the disadvantages of per-vertex lighting is that specular highlights can be warped or lost, due to the fact that the shading equation is not evaluated at each point across the face. For example, a specular highlight that should appear in the middle of a polygon might not appear at all when per-vertex lighting is used, because of the fact that the shading equation is only computed at the vertices where the specular component is near zero. In Chapter 3, Per-fragment shading, we'll look at the changes needed to move the shading computation into the fragment shader, producing more realistic results.

Directional lights

We can also avoid the need to compute a light direction (s), for each vertex if we assume a directional light. A directional light source is one that can be thought of as located infinitely far away in a given direction. Instead of computing the direction towards the source for each vertex, a constant vector is used, which represents the direction towards the remote light source. We'll look at an example of this in Chapter 3, Using a directional light source.

Light attenuation with distance

You might think that this shading model is missing one important component. It doesn't take into account the effect of the distance to the light source. In fact, it is known that the intensity of radiation from a source falls off in proportion to the inverse square of the distance from the source. So why not include this in our model?

It would be fairly simple to do so, however, the visual results are often less than appealing. It tends to exaggerate the distance effects and create unrealistic looking images. Remember, our equation is just an approximation of the physics involved and is not a truly realistic model, so it is not surprising that adding a term based on a strict physical law produces unrealistic results.

In the OpenGL fixed-function pipeline, it was possible to turn on distance attenuation using the glLight function. If desired, it would be straightforward to add a few uniform variables to our shader to produce the same effect.
The Basics of GLSL Shaders

See also
- Chapter 3, Using a directional light source
- Chapter 3, Per-fragment shading
- Chapter 3, Using the halfway vector for improved performance

Using functions in shaders

The GLSL supports functions that are syntactically similar to C functions. However, the calling conventions are somewhat different. In this example, we'll revisit the ADS shader using functions to help provide abstractions for the major steps.

Getting ready

As with previous recipes, provide the vertex position at attribute location 0 and the vertex normal at attribute location 1. Uniform variables for all of the ADS coefficients should be set from the OpenGL side, as well as the light position and the standard matrices.

How to do it...

To implement ADS shading using functions, use the following code:

1. Use the following vertex shader:

```glsl
#version 400
layout (location = 0) in vec3 VertexPosition;
layout (location = 1) in vec3 VertexNormal;
out vec3 LightIntensity;
struct LightInfo {
    vec4 Position; // Light position in eye coords.
    vec3 La;       // Ambient light intensity
    vec3 Ld;       // Diffuse light intensity
    vec3 Ls;       // Specular light intensity
};
uniform LightInfo Light;
struct MaterialInfo {
    vec3 Ka;            // Ambient reflectivity
    vec3 Kd;            // Diffuse reflectivity
    vec3 Ks;            // Specular reflectivity
    float Shininess;    // Specular shininess factor
};
uniform MaterialInfo Material;
```

For More Information:
www.packtpub.com/opengl-4-0-shading-language-cookbook/book
uniform mat4 ModelViewMatrix;
uniform mat3 NormalMatrix;
uniform mat4 ProjectionMatrix;
uniform mat4 MVP;

void getEyeSpace( out vec3 norm, out vec4 position )
{
    norm = normalize( NormalMatrix * VertexNormal);
    position = ModelViewMatrix * vec4(VertexPosition,1.0);
}

vec3 phongModel( vec4 position, vec3 norm )
{
    vec3 s = normalize(vec3(Light.Position - position));
    vec3 v = normalize(-position.xyz);
    vec3 r = reflect( -s, norm );
    vec3 ambient = Light.La * Material.Ka;
    float sDotN = max( dot(s,norm), 0.0 );
    vec3 diffuse = Light.Ld * Material.Kd * sDotN;
    vec3 spec = vec3(0.0);
    if( sDotN > 0.0 )
        spec = Light.Ls * Material.Ks *
            pow( max( dot(r,v), 0.0 ), Material.Shininess );
    return ambient + diffuse + spec;
}

void main()
{
    vec3 eyeNorm;
    vec4 eyePosition;
    // Get the position and normal in eye space
    getEyeSpace(eyeNorm, eyePosition);
    // Evaluate the lighting equation.
    LightIntensity = phongModel( eyePosition, eyeNorm );
    gl_Position = MVP * vec4(VertexPosition,1.0);
}

2. Use the following fragment shader:

```glsl
#version 400

in vec3 LightIntensity;
layout( location = 0 ) out vec4 FragColor;

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

For More Information:
www.packtpub.com/opengl-4-0-shading-language-cookbook/book
3. Compile and link both shaders within the OpenGL application, and install the shader program prior to rendering.

**How it works...**

In GLSL functions, the evaluation strategy is "call by value-return" (also called "call by copy-restore" or "call by value-result"). Parameter variables can be qualified with `in`, `out`, or `inout`. Arguments corresponding to input parameters (those qualified with `in` or `inout`) are copied into the parameter variable at call time, and output parameters (those qualified with `out` or `inout`) are copied back to the corresponding argument before the function returns. If a parameter variable does not have any of the three qualifiers, the default qualifier is `in`.

We've created two functions in the vertex shader. The first, named `getEyeSpace`, transforms the vertex position and vertex normal into eye space, and returns them via output parameters. In the main function, we create two uninitialized variables (`eyeNorm` and `eyePosition`) to store the results, and then call the function with the variables as the function's arguments. The function stores the results into the parameter variables (`norm` and `position`) which are copied into the arguments before the function returns.

The second function, `phongModel`, uses only input parameters. The function receives the eye-space position and normal, and computes the result of the ADS shading equation. The result is returned by the function and stored in the shader output variable `LightIntensity`.

**There's more...**

Since it makes no sense to read from an output parameter variable, output parameters should only be written to within the function. Their value is undefined.

Within a function, writing to an input only parameter (qualified with `in`) is allowed. The function's copy of the argument is modified, and changes are not reflected in the argument.

**The const qualifier**

The additional qualifier `const` can be used with input-only parameters (not with `out` or `inout`). This qualifier makes the input parameter read-only, so it cannot be written to within the function.

**Function overloading**

Functions can be overloaded by creating multiple functions with the same name, but with different number and/or type of parameters. As with many languages, two overloaded functions may not differ in return type only.

---

For More Information:

Passing arrays or structures to a function

It should be noted that when passing arrays or structures to functions, they are passed by value. If a large array or structure is passed, it can incur a large copy operation which may not be desired. It would be a better choice to declare these variables in the global scope.

See also

- Implementing per-vertex ambient, diffuse, and specular (ADS) shading

Implementing two-sided shading

When rendering a mesh that is completely closed, the back faces of polygons are hidden. However, if a mesh contains holes, it might be the case that the back faces would become visible. In this case, the polygons may be shaded incorrectly due to the fact that the normal vector is pointing in the wrong direction. To properly shade those back faces, one needs to invert the normal vector and compute the lighting equations based on the inverted normal.

The following image shows a teapot with the lid removed. On the left, the ADS lighting model is used. On the right, the ADS model is augmented with the two-sided rendering technique discussed in this recipe.

In this recipe, we'll look at an example that uses the ADS model discussed in the previous recipes, augmented with the ability to correctly shade back faces.

Getting ready

The vertex position should be provided in attribute location 0 and the vertex normal in attribute location 1. As in previous examples, the lighting parameters must be provided to the shader via uniform variables.

For More Information:

www.packtpub.com/opengl-4-0-shading-language-cookbook/book
The Basics of GLSL Shaders

How to do it...

To implement a shader pair that uses the ADS shading model with two-sided lighting, use the following code:

1. Use the following code for the vertex shader:

```glsl
#version 400
layout (location = 0) in vec3 VertexPosition;
layout (location = 1) in vec3 VertexNormal;

out vec3 FrontColor;
out vec3 BackColor;

struct LightInfo {
    vec4 Position; // Light position in eye coords.
    vec3 La;       // Ambient light intensity
    vec3 Ld;       // Diffuse light intensity
    vec3 Ls;       // Specular light intensity
};

uniform LightInfo Light;

struct MaterialInfo {
    vec3 Ka;            // Ambient reflectivity
    vec3 Kd;            // Diffuse reflectivity
    vec3 Ks;            // Specular reflectivity
    float Shininess;    // Specular shininess factor
};

uniform MaterialInfo Material;

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

vec3 phongModel( vec4 position, vec3 normal ) {
    // The ADS shading calculations go here (see: "Using
    // functions in shaders," and "Implementing
    // per-vertex ambient, diffuse and specular (ADS) shading")
    ...
}

void main()
{
    vec3 tnorm = normalize( NormalMatrix * VertexNormal);
    vec4 eyeCoords = ModelViewMatrix *
        vec4(VertexPosition, 1.0);

    FrontColor = phongModel( eyeCoords, tnorm );
}
```

For More Information:

www.packtpub.com/opengl-4-0-shading-language-cookbook/book
BackColor = phongModel( eyeCoords, -tnorm );

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

2. Use the following for the fragment shader:

```glsl
#version 400

in vec3 FrontColor;
in vec3 BackColor;
layout( location = 0 ) out vec4 FragColor;

void main() {
    if( gl_FrontFacing ) {
        FragColor = vec4(FrontColor, 1.0);
    } else {
        FragColor = vec4(BackColor, 1.0);
    }
}
```

3. Compile and link both shaders within the OpenGL application, and install the shader program prior to rendering.

### How it works...

In the vertex shader, we compute the lighting equation using both the vertex normal and the inverted version, and pass each resultant color to the fragment shader. The fragment shader chooses and applies the appropriate color depending on the orientation of the face.

The vertex shader is a slightly modified version of the vertex shader presented in the recipe Implementing per-vertex ambient, diffuse, and specular (ADS) shading. The evaluation of the shading model is placed within a function named phongModel. The function is called twice, first using the normal vector (transformed into eye coordinates), and second using the inverted normal vector. The combined results are stored in FrontColor and BackColor, respectively.

Note that there are a few aspects of the shading model that are independent of the orientation of the normal vector (such as the ambient component). One could optimize this code by rewriting it so that the redundant calculations are only done once. However, in this recipe we compute the entire shading model twice in the interest of making things clear and readable.

For More Information:
www.packtpub.com/opengl-4-0-shading-language-cookbook/book
In the fragment shader, we determine which color to apply based on the value of the built-in variable `gl_FrontFacing`. This is a Boolean value that indicates whether the fragment is part of a front or back facing polygon. Note that this determination is based on the **winding** of the polygon, and not the normal vector. (A polygon is said to have counter-clockwise winding if the vertices are specified in counter-clockwise order as viewed from the front side of the polygon.) By default when rendering, if the order of the vertices appear on the screen in a counter-clockwise order, it indicates a front facing polygon, however, we can change this by calling `glFrontFace` from the OpenGL program.

### There's more...

In the vertex shader we determine the front side of the polygon by the direction of the normal vector, and in the fragment shader, the determination is based on the polygon's winding. For this to work properly, the normal vector must be defined appropriately for the face determined by the setting of `glFrontFace`.

**Using two-sided rendering for debugging**

It can sometimes be useful to visually determine which faces are front facing and which are back facing. For example, when working with arbitrary meshes, polygons may not be specified using the appropriate winding. As another example, when developing a mesh procedurally, it can sometimes be helpful to determine which faces are oriented in the proper direction in order to help with debugging. We can easily tweak our fragment shader to help us solve these kinds of problems by mixing a solid color with all back (or front) faces. For example, we could change the `else` clause within our fragment shader to the following:

```
FragColor = mix( vec4(BackColor,1.0),
    vec4(1.0,0.0,0.0,1.0), 0.7 );
```

This would mix a solid red color with all back faces, helping them to stand out, as shown in the following image. In the image, back faces are mixed with 70% red as shown in the preceding code.

---

For More Information:

See also

- Implementing per-vertex ambient, diffuse, and specular (ADS) shading

**Implementing flat shading**

Per-vertex shading involves computation of the shading model at each vertex and associating the result (a color) with that vertex. The colors are then interpolated across the face of the polygon to produce a smooth shading effect. This is also referred to as Gouraud shading. In earlier versions of OpenGL, this per-vertex shading with color interpolation was the default shading technique.

It is sometimes desirable to use a single color for each polygon so that there is no variation of color across the face of the polygon, causing each polygon to have a flat appearance. This can be useful in situations where the shape of the object warrants such a technique, perhaps because the faces really are intended to look flat, or to help visualize the locations of the polygons in a complex mesh. Using a single color for each polygon is commonly called flat shading.

The images below show a mesh rendered with the ADS shading model. On the left, Gouraud shading is used. On the right, flat shading is used.

In earlier versions of OpenGL, flat shading was enabled by calling the function `glShadeModel` with the argument `GL_FLAT`. In which case, the computed color of the last vertex of each polygon was used across the entire face.

In OpenGL 4.0, flat shading is facilitated by the interpolation qualifiers available for shader input/output variables.

For More Information:

www.packtpub.com/opengl-4-0-shading-language-cookbook/book
How to do it...

To modify the ADS shading model to implement flat shading, use the following steps:

1. Use the same vertex shader as in the ADS example provided earlier. Change the output variable `LightIntensity` as follows:
   ```glsl
   #version 400
   layout (location = 0) in vec3 VertexPosition;
   layout (location = 1) in vec3 VertexNormal;
   flat out vec3 LightIntensity;
   // the rest is identical to the ADS shader...
   ```

2. Use the following code for the fragment shader:
   ```glsl
   #version 400
   flat in vec3 LightIntensity;
   layout( location = 0 ) out vec4 FragColor;
   void main() {
       FragColor = vec4(LightIntensity, 1.0);
   }
   ```

3. Compile and link both shaders within the OpenGL application, and install the shader program prior to rendering.

How it works...

Flat shading is enabled by qualifying the vertex output variable (and its corresponding fragment input variable) with the `flat` qualifier. This qualifier indicates that no interpolation of the value is to be done before it reaches the fragment shader. The value presented to the fragment shader will be the one corresponding to the result of the invocation of the vertex shader for either the first or last vertex of the polygon. This vertex is called the **provoking vertex**, and can be configured using the OpenGL function `glProvokingVertex`. For example, the call:

   ```glsl
   glProvokingVertex(GL_FIRST_VERTEX_CONVENTION);
   ```

This indicates that the first vertex should be used as the value for the flat shaded variable. The argument `GL_LAST_VERTEX_CONVENTION` indicates that the last vertex should be used.

For More Information:

See also

- Implementing per-vertex ambient, diffuse, and specular (ADS) shading

Using subroutines to select shader functionality

In GLSL, a subroutine is a mechanism for binding a function call to one of a set of possible function definitions based on the value of a variable. In many ways it is similar to function pointers in C. A uniform variable serves as the pointer and is used to invoke the function. The value of this variable can be set from the OpenGL side, thereby binding it to one of a few possible definitions. The subroutine's function definitions need not have the same name, but must have the same number and type of parameters and the same return type.

Subroutines therefore provide a way to select alternate implementations at runtime without swapping shader programs and/or recompiling, or using if statements along with a uniform variable. For example, a single shader could be written to provide several shading algorithms intended for use on different objects within the scene. When rendering the scene, rather than swapping shader programs (or using a conditional statement), we can simply change the subroutine's uniform variable to choose the appropriate shading algorithm as each object is rendered.

Since performance is crucial in shader programs, avoiding a conditional statement or a shader swap can be very valuable. With subroutines, we can implement the functionality of a conditional statement or shader swap without the computational overhead.

In this example, we'll demonstrate the use of subroutines by rendering a teapot twice. The first teapot will be rendered with the full ADS shading model described earlier. The second teapot will be rendered with diffuse shading only. A subroutine uniform will be used to choose between the two shading techniques.

For More Information:
www.packtpub.com/opengl-4-0-shading-language-cookbook/book
In the following image, we see an example of a rendering that was created using subroutines. The teapot on the left is rendered with the full ADS shading model, and the teapot on the right is rendered with diffuse shading only. A subroutine is used to switch between shader functionality.

**Getting ready**

As with previous recipes, provide the vertex position at attribute location 0 and the vertex normal at attribute location 1. Uniform variables for all of the ADS coefficients should be set from the OpenGL side, as well as the light position and the standard matrices.

We'll assume that, in the OpenGL application, the variable `programHandle` contains the handle to the shader program object.

**How to do it...**

To create a shader program that uses a subroutine to switch between pure-diffuse and ADS shading, use the following code:

1. Use the following code for the vertex shader:

   ```glsl
   #version 400
   subroutine vec3 shadeModelType( vec4 position, vec3 normal);
   subroutine uniform shadeModelType shadeModel;
   layout (location = 0) in vec3 VertexPosition;
   layout (location = 1) in vec3 VertexNormal;
   out vec3 LightIntensity;
   struct LightInfo {
   ```
vec4 Position; // Light position in eye coords.
vec3 La;       // Ambient light intensity
vec3 Ld;       // Diffuse light intensity
vec3 Ls;       // Specular light intensity
};
uniform LightInfo Light;
struct MaterialInfo {
    vec3 Ka;            // Ambient reflectivity
    vec3 Kd;            // Diffuse reflectivity
    vec3 Ks;            // Specular reflectivity
    float Shininess;    // Specular shininess factor
};
uniform MaterialInfo Material;
uniform mat4 ModelViewMatrix;
uniform mat3 NormalMatrix;
uniform mat4 ProjectionMatrix;
uniform mat4 MVP;
void getEyeSpace( out vec3 norm, out vec4 position )
{
    norm = normalize( NormalMatrix * VertexNormal);
    position = ModelViewMatrix * vec4(VertexPosition,1.0);
}
subroutine( shadeModelType )
vec3 phongModel( vec4 position, vec3 norm )
{
    // The ADS shading calculations go here (see: "Using
    // functions in shaders," and "Implementing
    // per-vertex ambient, diffuse and specular (ADS) shading")
    ...
}
subroutine( shadeModelType )
vec3 diffuseOnly( vec4 position, vec3 norm )
{
    vec3 s = normalize( vec3(Light.Position - position) );
    return
        Light.Ld * Material.Kd * max( dot(s, norm), 0.0 );
}
void main()
{
    vec3 eyeNorm;
    vec4 eyePosition;
    // Get the position and normal in eye space

For More Information:
www.packtpub.com/opengl-4-0-shading-language-cookbook/book
The Basics of GLSL Shaders

```cpp
getEyeSpace(eyeNorm, eyePosition);
// Evaluate the shading equation. This will call one of
// the functions: diffuseOnly or phongModel.
LightIntensity = shadeModel(eyePosition, eyeNorm);
gl_Position = MVP * vec4(VertexPosition,1.0);
}
```

2. Use the following code for the fragment shader:

```cpp
#version 400
in vec3 LightIntensity;
layout( location = 0 ) out vec4 FragColor;
void main() {
    FragColor = vec4(LightIntensity, 1.0);
}
```

3. In the OpenGL application, compile and link the above shaders into a shader program, and install the program into the OpenGL pipeline.

4. Within the render function of the OpenGL application, use the following code:

```cpp
GLuint adsIndex =
    glGetUniformLocation(programHandle,
                        GL_VERTEX_SHADER,"phongModel");
GLuint diffuseIndex =
    glGetUniformLocation(programHandle,
                        GL_VERTEX_SHADER, "diffuseOnly");
glUniformSubroutinesuiv(GL_VERTEX_SHADER, 1, &adsIndex);
... // Render the left teapot
glUniformSubroutinesuiv(GL_VERTEX_SHADER, 1, &diffuseIndex);
... // Render the right teapot
```

How it works...

In this example, the subroutine is defined within the vertex shader. The first step involves declaring the subroutine type.

```cpp
subroutine vec3 shadeModelType(vec4 position,
                            vec3 normal);
```

This defines a new subroutine type with the name `shadeModelType`. The syntax is very similar to a function prototype, in that it defines a name, a parameter list, and a return type. As with function prototypes, the parameter names are optional.
After creating the new subroutine type, we declare a uniform variable of that type named `shadeModel`.

    subroutine uniform shadeModelType shadeModel;

This variable serves as our function pointer and will be assigned to one of the two possible functions in the OpenGL application.

We declare two functions to be part of the subroutine by prefixing their definition with the subroutine qualifier:

    subroutine ( shadeModelType )

This indicates that the function matches the subroutine type, and therefore its header must match the one in the subroutine type definition. We use this prefix for the definition of the functions `phongModel` and `diffuseOnly`. The `diffuseOnly` function computes the diffuse shading equation, and the `phongModel` function computes the complete ADS shading equation.

We call one of the two subroutine functions by utilizing the subroutine uniform `shadeModel` within the main function.

    LightIntensity = shadeModel( eyePosition, eyeNorm );

Again, this call will be bound to one of the two functions depending on the value of the subroutine uniform `shadeModel`, which we will set within the OpenGL application.

Within the render function of the OpenGL application, we assign a value to the subroutine uniform with the following steps. First, we query for the index of each subroutine function using `glGetSubroutineIndex`. The first argument is the program handle. The second is the shader stage. In this case, the subroutine is defined within the vertex shader, so we use `GL_VERTEX_SHADER` here. The third argument is the name of the subroutine. We query for each function individually and store the indexes in the variables `adsIndex` and `diffuseIndex`.

To select the appropriate subroutine function, we need to set the value of the subroutine uniform `shadeModel`. To do so, we call `glUniformSubroutinesuiv`. This function is designed for setting multiple subroutine uniforms at once. In our case, of course, we are setting only a single uniform. The first argument is the shader stage (`GL_VERTEX_SHADER`), the second is the number of uniforms being set, and the third is a pointer to an array of subroutine function indexes. Since we are setting a single uniform, we simply provide the address of the `GLuint` variable containing the index, rather than a true array of values. Of course, we would use an array if multiple uniforms were being set. In general, the array of values provided as the third argument is assigned to subroutine uniform variables in the following way. The `i`th element of the array is assigned to the subroutine uniform variable with index `i`. Since we have provided only a single value, we are setting the subroutine uniform at index zero.

For More Information:
www.packtpub.com/opengl-4-0-shading-language-cookbook/book
You may be wondering, "How do we know that our subroutine uniform is located at index zero? We didn't query for the index before calling `glUniformSubroutinesuiv`!" The reason that this code works is that we are relying on the fact that OpenGL will always number the indexes of the subroutines consecutively starting at zero. If we had multiple subroutine uniforms, we could (and should) query for their indexes using `glGetSubroutineUniformLocation`, and then order our array appropriately.

Finally, we select the `phongModel` function by setting the uniform to `adsIndex` and then render the left teapot. We then select the `diffuseOnly` function by setting the uniform to `diffuseIndex` and render the right teapot.

**There's more...**

A subroutine function defined in a shader can match multiple subroutine types. In that case, the subroutine qualifier can contain a comma-separated list of subroutine types. For example, if a subroutine matched the types `type1` and `type2`, we could use the following qualifier:

```
subroutine( type1, type2 )
```

This would allow us to use subroutine uniforms of differing types to refer to the same subroutine function.

**See also**

- Implementing per-vertex ambient, diffuse, and specular (ADS) shading
- Implementing diffuse, per-vertex shading with a single point light source

**Discarding fragments to create a perforated look**

Fragment shaders can make use of the `discard` keyword to "throw away" fragments. Use of this keyword causes the fragment shader to stop execution, without writing anything (including depth) to the output buffer. This provides a way to create holes in polygons without using blending. In fact, since fragments are completely discarded, there is no dependence on the order in which objects are drawn, saving us the trouble of doing any depth sorting that might have been necessary if blending was used.

In this recipe, we'll draw a teapot, and use the `discard` keyword to remove fragments selectively based on texture coordinates. The result will look like the following image:
Getting ready

The vertex position, normal, and texture coordinates must be provided to the vertex shader from the OpenGL application. The position should be provided at location 0, the normal at location 1, and the texture coordinates at location 2. As in previous examples, the lighting parameters must be set from the OpenGL application via the appropriate uniform variables.

How to do it...

To create a shader program that discards fragments based on a square lattice (as in the preceding image), use the following code:

1. Use the following code for the vertex shader:

```glsl
#version 400
layout (location = 0) in vec3 VertexPosition;
layout (location = 1) in vec3 VertexNormal;
layout (location = 2) in vec2 VertexTexCoord;
out vec3 FrontColor;
out vec3 BackColor;
out vec2 TexCoord;
struct LightInfo {
    vec4 Position; // Light position in eye coords.
    vec3 La;       // Ambient light intensity
    vec3 Ld;       // Diffuse light intensity
    vec3 Ls;       // Specular light intensity
}
```

For More Information:

www.packtpub.com/opengl-4-0-shading-language-cookbook/book
The Basics of GLSL Shaders

```glsl
struct MaterialInfo {
    vec3 Ka;            // Ambient reflectivity
    vec3 Kd;            // Diffuse reflectivity
    vec3 Ks;            // Specular reflectivity
    float Shininess;    // Specular shininess factor
};

uniform LightInfo Light;
uniform MaterialInfo Material;
uniform mat4 ModelViewMatrix;
uniform mat3 NormalMatrix;
uniform mat4 ProjectionMatrix;
uniform mat4 MVP;

void getEyeSpace( out vec3 norm, out vec4 position )
{
    norm = normalize( NormalMatrix * VertexNormal);
    position = ModelViewMatrix * vec4(VertexPosition,1.0);
}

vec3 phongModel( vec4 position, vec3 norm )
{
    // The ADS shading calculations go here (see: "Using
    // functions in shaders," and "Implementing
    // per-vertex ambient, diffuse and specular (ADS) shading")
    ...
}

void main()
{
    vec3 eyeNorm;
    vec4 eyePosition;

    TexCoord = VertexTexCoord;
    // Get the position and normal in eye space
    getEyeSpace(eyeNorm, eyePosition);

    FrontColor = phongModel( eyePosition, eyeNorm );
    BackColor = phongModel( eyePosition, -eyeNorm );
    gl_Position = MVP * vec4(VertexPosition,1.0);
}
```

2. Use the following code for the fragment shader:

```glsl
#version 400
in vec3 FrontColor;
in vec3 BackColor;
```

For More Information:
www.packtpub.com/opengl-4-0-shading-language-cookbook/book
in vec2 TexCoord;
layout( location = 0 ) out vec4 FragColor;
void main() {
    const float scale = 15.0;
    bvec2 toDiscard = greaterThan( fract(TexCoord * scale),
                                    vec2(0.2, 0.2) );
    if( all(toDiscard) )
        discard;
    if( gl_FrontFacing )
        FragColor = vec4(FrontColor, 1.0);
    else
        FragColor = vec4(BackColor, 1.0);
}

3. Compile and link both shaders within the OpenGL application, and install the shader program prior to rendering.

How it works...

Since we will be discarding some parts of the teapot, we will be able to see through the teapot to the other side. This will cause the back sides of some polygons to become visible. Therefore, we need to compute the lighting equation appropriately for both sides of each face. We'll use the same technique presented earlier in the two-sided shading recipe.

The vertex shader is essentially the same as in the two-sided shading recipe, with the main difference being the addition of the texture coordinate. The differences are highlighted in the above listing. To manage the texture coordinate, we have an additional input variable, VertexTexCoord, that corresponds to attribute location 2. The value of this input variable is passed directly on to the fragment shader unchanged via the output variable TexCoord. The ADS shading model is calculated twice, once using the given normal vector, storing the result in FrontColor, and again using the reversed normal, storing that result in BackColor.

In the fragment shader, we calculate whether or not the fragment should be discarded based on a simple technique designed to produce the lattice-like pattern shown in the preceding image. We first scale the texture coordinate by the arbitrary scaling factor scale. This corresponds to the number of lattice rectangles per unit (scaled) texture coordinate. We then compute the fractional part of each component of the scaled texture coordinate using the built-in function fract. Each component is compared to 0.2 using the built-in function greaterThan, and the result is stored in the bool vector toDiscard. The greaterThan function compares the two vectors component-wise, and stores the Boolean results in the corresponding components of the return value.

For More Information:
www.packtpub.com/opengl-4-0-shading-language-cookbook/book
If both components of the vector toDiscard are true, then the fragment lies within the inside of each lattice frame, and therefore we wish to discard this fragment. We can use the built-in function all to help with this check. The function all will return true if all of the components of the parameter vector are true. If the function returns true, we execute the discard statement to reject the fragment.

In the else branch, we color the fragment based on the orientation of the polygon, as in the two-sided shading recipe presented earlier.

See also

- *Implementing two-sided shading*
Where to buy this book

You can buy OpenGL 4.0 Shading Language Cookbook from the Packt Publishing website: http://www.packtpub.com/opengl-4-0-shading-language-cookbook/book

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.