CHAPTER 8

Implementing Illumination and Shadow

After completing this chapter, you will be able to:

- Understand the parameters of simple illumination models
- Define infrastructure supports for working with multiple light sources
- Understand the basics of diffuse reflection and normal mapping
- Understand the basics of specular reflection and the Phong illumination model
- Implement GLSL shaders to simulate diffuse and specular reflection and the Phong illumination model
- Create and manipulate point, directional, and spotlights
- Simulate shadows with the WebGL stencil buffer

Introduction

Up to now in your game engine you have implemented mostly functional modules in order to provide the core fundamentals required for many types of 2D games (that is, modules that serve to provide functionality directly to the end gameplay of a game created with your engine). This is a great approach because it allows you to systematically expand the capabilities of your engine to allow more types of games and gameplay. For instance, with the topics covered thus far, you can implement a variety of different games including puzzle games, top-down space shooters, and even simple platform games.

An illumination model, or a lighting model, is a mathematic formulation that describes the color and brightness of a scene based on simulating light energy reflecting off the surfaces in the scene. In this chapter, you will implement an illumination model that indirectly affects the types of gameplay your game engine can support and the visual fidelity that can be achieved. This is because illumination within a game engine can be more than a simple aesthetic effect. When used creatively, illumination can enhance gameplay or provide a dramatic setting for your game. For example, you could have a scene with a torch light that illuminates an otherwise dark pathway for the hero, with the torch flickering to communicate a sense of unease or danger to the player. Additionally, while the lighting model is based on light behaviors within the real world, in your game implementation the lighting model allows surreal or physically impossible settings, such as an oversaturated light source that displays bright or iridescent colors or even a negative light intensity that seemingly absorbs the light around it.
When implementing illumination models commonly present in game engines, you will need to venture into concepts in 3D space to properly simulate light within your scenes. You will need to define depth values for the light sources to cast light energy upon the game objects, or renderables, which are flat 2D geometries. Once you consider concepts in 3D, the task of implementing a lighting model becomes much more straightforward, and you can apply knowledge from computer graphics to properly illuminate a scene.

A variation of the Phong illumination model will be derived and implemented in your game engine. While there are many versions of the Phong illumination model, you will be implementing a simplified version that caters to the 2D aspect of your game engine. However, the principles of the illumination model remain the same. If you desire more information or a further in-depth analysis of the Phong illumination model, please refer to the discussion in Chapter 1.

Overview of Illumination and GLSL Implementation

In general, an illumination model is one or a set of mathematical equations describing how humans observe the interaction of light with object materials in the environment. As you can imagine, an accurate illumination model that is based on the physical world can be highly complex and computationally intensive. The Phong illumination model captures many of the interesting aspects of light/material interactions with a relatively simple equation that can be implemented efficiently. The projects in this chapter guide you in understanding the fundamental elements of the Phong illumination model.

- **Ambient Light**: Reviews the effects of lights in the absence of explicit light sources
- **Light Source**: Examines the effect of illumination from a single light source
- **Multiple Light Sources**: Develops game engine infrastructure to support multiple light sources
- **Diffuse Reflection and Normal Maps**: Simulates diffuse light reflection in 2D
- **Specular Light and Material**: Models light reflecting off surfaces and reaching the camera
- **Light Source types**: Introduces illumination based on different types of light sources
- **Shadow**: Approximates the results from light occlusion

Together, the projects in this chapter build a powerful tool for adding visual intricacy into your games.

ambient Light

Ambient light, often referred to as background light, allows you to see objects in the environment when there are no explicit light sources. For example, in the dark of night, you can see objects in a room even though all lights are switched off. In the real world, light coming from the window, from underneath the door, or from the background illuminates the room for you. A realistic simulation of the background light illumination, often referred to as *indirect illumination*, is complex and computationally too expensive to simulate in real time. Instead, in computer graphics and most 2D games, ambient lighting is approximated by adding a constant ambient light color to every object within the current scene or world. It is important to note that while ambient lighting can provide desired results, it is not meant to mimic real-world lighting exactly. For your specific engine implementation, each object within the scene needs access to an ambient color and an ambient intensity before it is drawn in order to take into account the ambient lighting of the scene.
The Global Ambient Project

This project demonstrates how to implement ambient lighting within your scenes by providing a global ambient color and a global ambient intensity that each renderable object references before being drawn. You can see an example of this project running in Figure 8-1. The source code of this project is located in the Chapter8/8.1.GlobalAmbient folder.

![Running the Global Ambient project](image)

**Figure 8-1. Running the Global Ambient project**

The controls of the project are as follows:

- *Left mouse button*: Increases the global red ambient
- *Middle mouse button*: Decreases the global red ambient
- *Left/right arrow keys*: Decrease/increase the global ambient intensity

The goals of the project are as follows:

- To experience the effects of ambient lighting
- To understand how to implement a simple global ambient across a scene
- To refamiliarize yourself with the Shader/Renderable pair structure to interface to GLSL shaders and the game engine
You can find the following external resources in the assets folder: the fonts folder that contains the default system fonts and two texture images (minion_sprite.png, which defines the sprite elements for the hero and the minions, and bg.png, which defines the background).

### Modifying the GLSL Shaders

A good place to start when implementing new shaders for the game engine is the GLSL shader. This is because it allows you to implement the shading technique, which in turn provides the outline for how your engine must be modified in order to support this new shader. Thus, to start, implement the global ambient into your SimpleFS.glsl.

1. Modify the fragment shader SimpleFS.glsl by adding the uniform variables `uGlobalAmbientColor` and `uGlobalAmbientIntensity`. Then utilize them by multiplying them by `uPixelColor` to get the final color for each fragment. You can see this implemented in the following code:

   ```c
   precision mediump float;
   // Color of the object
   uniform vec4 uPixelColor;
   uniform vec4 uGlobalAmbientColor; // this is shared globally
   uniform float uGlobalAmbientIntensity; // this is shared globally

   void main(void) {
       // for every pixel called sets to the user specified color
       gl_FragColor = uPixelColor * uGlobalAmbientIntensity * uGlobalAmbientColor;
   }
   ```

2. Similarly modify the texture fragment shader TextureFS.glsl by adding the uniform variables `uGlobalAmbientColor` and `uGlobalAmbientIntensity`. Then utilize them by multiplying them by `c` (the fragment color sampled from the texture) to get the final color for each fragment. Remember that the color for `c` was obtained by using the interpolated `vTexCoord` variable from the vertex shader to sample a fragment from the passed-in texture. You can see this implemented in the following code:

   ```c
   precision mediump float;
   uniform sampler2D uSampler;
   // Color of the object
   uniform vec4 uPixelColor;
   uniform vec4 uGlobalAmbientColor; // this is shared globally
   uniform float uGlobalAmbientIntensity;
   varying vec2 vTexCoord;

   void main(void) {
       // texel color look up based on interpolated UV value in vTexCoord
       vec4 c = texture2D(uSampler, vec2(vTexCoord.s, vTexCoord.t));

       c = c * uGlobalAmbientIntensity * uGlobalAmbientColor;
   }
   ```
// tint the textured area, and
// leave transparent area as defined by the texture
vec3 r = vec3(c) * (1.0-uPixelColor.a) + vec3(uPixelColor) * uPixelColor.a;
vec4 result = vec4(r, c.a);

gl_FragColor = result;

Modifying SimpleShader

With global ambient color and intensity now implemented within the shader, you need to modify the simple shader in order to accommodate the new variables by passing the values onto the GLSL shader.

1. Modify the `SimpleShader.js` file in the `src/Engine/Shaders` folder to hold two new variables in the constructor for storing the references or locations of the ambient color and intensity variables within the GLSL shader.

```javascript
this.mGlobalAmbientColor = null;
this.mGlobalAmbientIntensity = null;
```

2. In step E of the `SimpleShader` constructor, get the locations of the ambient color and intensity within the shader by using WebGL's `getUniformLocation()` function, as shown in the following code:

```javascript
// Step E: references: uniforms:
//       uPixelColor, uModelTransform, and uViewProjTransform
this.mPixelColor = gl.getUniformLocation(this.mCompiledShader, "uPixelColor");
this.mModelTransform = gl.getUniformLocation(this.mCompiledShader, "uModelTransform");
this.mViewProjTransform = gl.getUniformLocation(this.mCompiledShader, "uViewProjTransform");
this.mGlobalAmbientColor = gl.getUniformLocation(this.mCompiledShader, "uGlobalAmbientColor");
this.mGlobalAmbientIntensity = gl.getUniformLocation(this.mCompiledShader, "uGlobalAmbientIntensity");
```

3. In the `activateShader()` function, pass the ambient color and intensity values to the shader by utilizing the global values and the locations you obtained in the previous step along with the GLSL uniform set functions provided by WebGL. Notice that the data type used by the set functions for GLSL variables state which data type is used explicitly. As you can probably guess, `uniform4fv` corresponds to `vec4`, which is used to hold the color, and to `uniform1f`, which corresponds to a float and is used to hold the intensity.

```javascript
// Activate the shader for rendering
SimpleShader.prototype.activateShader = function(pixelColor, aCamera) {
    var gl = gEngine.Core.getGL();
    gl.useProgram(this.mCompiledShader);
    gl.uniformMatrix4fv(this.mViewProjTransform, false, aCamera.getVPMatrix());
    gl.bindBuffer(gl.ARRAY_BUFFER, gEngine.VertexBuffer.getGLVertexRef());
    gl.vertexAttribPointer(this.mShaderVertexPositionAttribute,
```
Chapter 8 ■ Implementing Illumination and Shadow

```javascript
3, // each element is a 3-float (x,y,z)
gl.FLOAT, // data type is FLOAT
false, // if the content is normalized vectors
0, // number of bytes to skip in between elements
0); // offsets to the first element
gl.enableVertexAttribArray(this.mShaderVertexPositionAttribute);
gl.uniform4fv(this.mPixelColor, pixelColor);
gl.uniform4fv(this.mGlobalAmbientColor,
gEngine.DefaultResources.getGlobalAmbientColor());
gl.uniform1f(this.mGlobalAmbientIntensity,
gEngine.DefaultResources.getGlobalAmbientIntensity());
}
```

Modifying the Engine

Now that the shader and the corresponding shader object for ambient color and intensity are properly integrated, you can modify the engine in order to support the variables for global ambient.

1. Begin by adding a global ambient color and a global ambient intensity variable in the Engine_DefaultResources.js file, located in the src/Engine/Core/Resources folder, as shown here:

    ```javascript
    // Global Ambient color
    var mGlobalAmbientColor = [0.3, 0.3, 0.3, 1];
    var mGlobalAmbientIntensity = 1;
    ```

2. Define basic get and set accessors to allow for the modification of the ambient color and intensity, as shown in the following code:

    ```javascript
    var getGlobalAmbientIntensity = function() { return mGlobalAmbientIntensity; }
    var setGlobalAmbientIntensity = function(v) { mGlobalAmbientIntensity = v; }
    var getGlobalAmbientColor = function() { return mGlobalAmbientColor; }
    var setGlobalAmbientColor = function(v) {
        mGlobalAmbientColor = vec4.fromValues(v[0], v[1], v[2], v[3]); }
    }
    ```

3. Remember to include the accessors in your mPublic() function list.

    ```javascript
    getGlobalAmbientColor: getGlobalAmbientColor,
    setGlobalAmbientColor: setGlobalAmbientColor,
    getGlobalAmbientIntensity: getGlobalAmbientIntensity,
    setGlobalAmbientIntensity: setGlobalAmbientIntensity,
    ```

Testing the Ambient Illumination

Now that the engine supports ambient lighting for a scene, all that is left is to verify the correctness by utilizing it within MyGame.js and observe the shaded results. To get a better picture of what exactly is happening and how exactly the ambient lighting can be utilized, you can begin with a clean MyGame.js file and re-implement the core functions.
1. Start by adding variables to the constructor for a camera, a background, a hero, and some minions. Additionally, remember to inherit from the Scene class in order to utilize its functionality.

"use strict";
function MyGame() {
  this.kMinionSprite = "assets/minion_sprite.png";
  this.kBg = "assets/bg.png";

  // The camera to view the rectangles
  this.mCamera = null;
  this.mBg = null;

  this.mMsg = null;

  // the hero and the support objects
  this.mHero = null;
  this.mLMinion = null;
  this.mRMinion = null;
}
gEngine.Core.inheritPrototype(MyGame, Scene);

2. Next, practice proper implementation by remembering to load and unload the background and the minions. You can see this in the following code:

MyGame.prototype.loadScene = function() {
  gEngine.Textures.LoadTexture(this.kMinionSprite);
  gEngine.Textures.LoadTexture(this.kBg);
};

MyGame.prototype.unloadScene = function() {
  gEngine.Textures.UnloadTexture(this.kMinionSprite);
  gEngine.Textures.UnloadTexture(this.kBg);
};

3. Now initialize the camera and scene objects by setting them to the values shown here so that the scene is visible by the camera upon startup:

MyGame.prototype.initialize = function() {
  // Step A: set up the cameras
  this.mCamera = new Camera(
    vec2.fromValues(50, 37.5), // position of the camera
    100,                      // width of camera
    [0, 0, 640, 480]           // viewport (orgX, orgY, width, height)
  );
  this.mCamera.setBackgroundColor([0.8, 0.8, 0.8, 1]);
  // Sets the background to gray

  var bgR = new SpriteRenderable(this.kBg);
  bgR.setTexPixelPositions(0, 1900, 0, 1000);
  bgR.getXform().setSize(190, 100);
Chapter 8

Implementing Illumination and Shadow

bgR.getXform().setPosition(50, 35);
this.mBg = new GameObject(bgR);

// Step B: Create the hero object with texture from lower-left corner
this.mHero = new Hero(this.kMinionSprite);
this.mLMinion = new Minion(this.kMinionSprite, 30, 30);
this.mRMinion = new Minion(this.kMinionSprite, 70, 30);

this.mMsg = new FontRenderable("Status Message");
this.mMsg.setColor([1, 1, 1, 1]);
this.mMsg.getXform().setPosition(1, 2);
this.mMsg.setTextSize(3);
};

4. Next, draw each object in the scene, as shown in the following functions:

MyGame.prototype.drawCamera = function(camera) {
    camera.setupViewProjection();
    this.mBg.draw(camera);
    this.mHero.draw(camera);
    this.mLMinion.draw(camera);
    this.mRMinion.draw(camera);
};

// This is the draw function, make sure to setup proper drawing environment,
// and more importantly, make sure to _NOT_ change any state.
MyGame.prototype.draw = function() {
    // Step A: clear the canvas
    gEngine.Core.clearCanvas([0.9, 0.9, 0.9, 1.0]); // clear to light gray

    // Step B: draw with all three cameras
    this.drawCamera(this.mCamera);
    this.mMsg.draw(this.mCamera); // only draw status in the main camera
};

5. Lastly, implement the following update function to update each object as well
as the camera within the scene. Additionally, provide control over the global
ambient color and intensity for testing purposes and display a status for color
and intensity.

// The update function, updates the application state. Make sure to _NOT_ draw
// anything from this function!
MyGame.prototype.update = function() {
    var deltaAmbient = 0.01;
    var msg = "Current Ambient]: ";

    this.mCamera.update(); // to ensure proper interpolated movement effects
    this.mLMinion.update(); // ensure sprite animation
    this.mRMinion.update();

this.mHero.update(); // allow keyboard control to move
this.mCamera.panWith(this.mHero.getXform(), 0.8);

var v = gEngine.DefaultResources.getGlobalAmbientColor();
if (gEngine.Input.isButtonPressed(gEngine.Input.mouseButton.Left))
    v[0] += deltaAmbient;
if (gEngine.Input.isButtonPressed(gEngine.Input.mouseButton.Middle))
    v[0] -= deltaAmbient;
if (gEngine.Input.isKeyPressed(gEngine.Input.keys.Left))
    gEngine.DefaultResources.setGlobalAmbientIntensity(
        gEngine.DefaultResources.getGlobalAmbientIntensity() - deltaAmbient);
if (gEngine.Input.isKeyPressed(gEngine.Input.keys.Right))
    gEngine.DefaultResources.setGlobalAmbientIntensity(
        gEngine.DefaultResources.getGlobalAmbientIntensity() + deltaAmbient);

msg += " Red=" + v[0].toPrecision(3) + " Intensity=" +
    gEngine.DefaultResources.getGlobalAmbientIntensity().toPrecision(3);
this.mMsg.setText(msg);
);

Observations
You can now see the results of the project by running it. Notice that the scene itself is dark. This is because
the RGB values for the global ambient color were all initialized to 0.3, and since the ambient color is
multiplied by the color sampled from the textures, the results are similar to applying a dark tint across the
entire scene. The same effect would happen if the RGB values were set to 1 and the intensity was set 0.3
because applying the ambient values is done through straightforward multiplication. Before moving onto
the next project, try fiddling with the ambient red channel and the ambient intensity in order to see its effect
on the scene. By pressing the right arrow key, you can increase the intensity of the entire scene and make all
objects more visible. The next section describes how to create and direct a light source to illuminate only on
selected objects.

Light Source
With ambient lighting for the scene completed, it is now time to implement light with an object-oriented
approach while adhering to your expectations of what a light is and how it interacts with the environment.
This can be achieved through the definition of a Light object to represent a light source. As mentioned, to
implement a light source, the 2D engine will need to venture into the third dimension to properly simulate
light energy traveling from the source to the surface of your geometries. There are several types of light
sources; you will begin with the implementation of a simple point light. A point light is a light that emits light
uniformly in all directions. In the real world, a point light can be thought of as a simple lightbulb.

The most basic implementation of a point light can be distilled down to illuminating an area or radius
around a specified point. In three-dimensional space, the region of illumination by a point light source
can simply be thought of as a sphere, referred to as volume of illumination. The volume of illumination of a
point light is defined with the light's position being at the center of the sphere and illuminating the volume
that is within the radius of the sphere. To observe the effects of a light source, objects must be present and
within the volume of illumination. Now, consider your 2D engine; thus far you have implemented a system
in which everything is rendered in 2D. Or, rather, everything is rendered at a single plane where z = 0 and
objects are layered (by draw order) in order to display one object in front of another. On this system, you
are now going to add light sources that reside in 3D. To observe the effects of a light source, its illumination volume must overlap an object on the XY plane where your objects exist. Figure 8-2 shows the volume of illumination from a simple point light intersecting an object on the XY plane where \( z = 0 \). This intersection results in an illuminated circle on the object.

**Figure 8-2.** Point light and the corresponding volume of illumination in 3D

**GLSL Implementation and Integration into the Game Engine**

For quality results, the computations associated with illumination models must be performed once for each affected pixel. Recall that in WebGL and GLSL shaders the color of each pixel is computed by the corresponding GLSL fragment shader. In this chapter, as each fundamental element of the Phong illumination model is studied, the accompanied GLSL fragment shader implementation will also be explained.

From your experience building the game engine, you will remember that the engine interfaces to the GLSL shaders with the corresponding subclasses of the `Shader/Renderable` object pairs: `Shader` objects to interface to the GLSL shaders and `Renderable` objects to provide programmers with the convenience of manipulating many copies of geometries of the same shader type. For example, `TextureVS` and `TextureFS` are interfaced to the game engine via the `TextureShader` object, and the `TextureRenderable` objects allow game programmers to create and manipulate multiple instances of geometries shaded by the `TextureVS/FS` shaders. Figure 8-3 depicts that the next project extends this architecture to implement point light illumination. The `Light` class encapsulates the attributes of a point light including position, radius, and color. This information is forwarded to the GLSL fragment shader, `LightFS`, via the `LightShader/LightRenderable` pair for computing the appropriate pixel colors. The GLSL vertex shader, `TextureVS`, is reused because light source illumination involves the same information to be processed at each vertex.
Finally, before you begin learning about the elements of a Phong Illumination model, it is important to point out again that the GLSL fragment shader is invoked once for every pixel covered by the corresponding geometry. This means the GLSL fragment shaders you are about to learn will be invoked many times per frame, probably in the range of hundreds of thousands or even millions. Considering the fact that the game loop initiates redrawing at a real-time rate, or around 60 frame redraws per second, the GLSL fragment shaders will be invoked many millions of times per second! The efficiency of the implementation is of most importance!

**The Simple Light Shader Project**

This project demonstrates how to implement a simple point light and illuminate objects within the scene. You can see an example of this project running in Figure 8-4. The source code of this project is located in the Chapter8/8.2.SimpleLightShader folder.

---

*Figure 8-3. LightShader/LightRenderable pair and the corresponding GLSL LightShader*
The controls of the project are as follows:

- **WASD keys**: Move the hero character on the screen
- **WASD keys + left mouse button**: Move the hero character and the light source around the screen
- **Left/right arrow key**: Decreases/increases the light intensity
- **Z/X key**: Increases/decreases the light Z position
- **C/V key**: Increases/decreases the light radius

The goals of the project are as follows:

- To understand how to simulate the illumination effects from a point light
- To experience illumination results from a point light
- To implement a GLSL shader that supports point light illumination

You can find the following external resources in the assets folder: the fonts folder that contains the default system fonts and two texture images (minion_sprite.png and bg.png). The objects are sprite elements of minion_sprite.png, and the background is represented by bg.png.
Creating the GLSL Light Fragment Shader

As with the previous section, the implementation will begin with the GLSL shader. The shader uses light properties to calculate the illuminated circle. The GLSL vertex shader will remain identical to the TextureVS since the same information and computation will be performed at each vertex.

1. Under the src/GLSLShaders folder, create a new file and name it LightFS.glsl.

2. Add the standard uniform and varying variables for the texture sampler, texture coordinate, ambient properties, and pixel color as in previous projects. Furthermore, you can now add support for a single light by adding variables for each of the light's properties. It is also important to notice that the light's position and radius are in pixel space, or Device Coordinate (DC) space, to facilitate illumination computations.

```cpp
precision mediump float;

// The object that fetches data from texture.
uniform sampler2D uSampler;

// Color of pixel
uniform vec4 uPixelColor;
uniform vec4 uGlobalAmbientColor; // this is shared globally
uniform float uGlobalAmbientIntensity;

// Light information
uniform bool uLightOn;
uniform vec4 uLightColor;
uniform vec4 uLightPosition; // in pixel space!
uniform float uLightRadius; // in pixel space!

varying vec2 vTexCoord;
```

3. To complete the shader, implement the `main()` function to do the following:

   a. Sample the texture color and apply the ambient color and intensity.

   b. Determine whether the current fragment should be illuminated by the light source. To do this, first check whether the light is on; if it is, compute the distance between the light's position (in pixel space) and the current fragment's position (in pixel space) that is defined in the GLSL-provided variable `gl_FragCoord.xyz`. If the distance is less than that of the light's radius (again in pixel space), then accumulate the light's color.

   c. The last step is to apply the tint and to set the final color via `gl_FragColor`.

```cpp
void main(void) {
    vec4 textureMapColor = texture2D(uSampler, vec2(vTexCoord.s, vTexCoord.t));
    vec4 lgtResults = uGlobalAmbientIntensity * uGlobalAmbientColor;

    // now decide if we should illuminate by the light
    if (uLightOn && (textureMapColor.a > 0.0)) {
        float dist = length(uLightPosition.xyz - gl_FragCoord.xyz);
    }
    gl_FragColor = lgtResults;
}```
if (dist <= uLightRadius)
    lgtResults += uLightColor;
}
lgtResults *= textureMapColor;

// tint the textured area, and leave transparent area as defined by the texture
vec3 r = vec3(lgtResults) * (1.0-uPixelColor.a) + vec3(uPixelColor) * uPixelColor.a;
vec4 result = vec4(r, lgtResults.a);

gl_FragColor = result;

---

Creating a Light Object

With the GLSL LightFS shader defined, you can now create an object to encapsulate a point light source.

1. Create a new folder called Lights under the src/Engine folder. In the Lights folder, add a new file and name it Lights.js. Remember to load this new source file in index.html.

2. In Lights.js, create a simple constructor that initializes a color, position, radius, and on-off variable for the light. Set their initial values as follows:

```javascript
// Constructor
function Light() {
    this.mColor = vec4.fromValues(0.1, 0.1, 0.1, 1);  // light color
    this.mPosition = vec3.fromValues(0, 0, 5); // light position in WC
    this.mRadius = 10;  // effective radius in WC
    this.mIsOn = true;
}
```

3. Add the get and set accessors shown here to allow for the proper modification of the light’s instance variables from outside the object.

```javascript
// simple setters and getters
Light.prototype.setColor = function(c) { this.mColor = vec4.clone(c); };
Light.prototype.getColor = function() { return this.mColor; };
Light.prototype.set2DPosition = function(p) {
    this.mPosition = vec3.fromValues(p[0], p[1], this.mPosition[2]);}
Light.prototype.setXPos = function(x) { this.mPosition[0] = x; };
Light.prototype.setYPos = function(y) { this.mPosition[1] = y; };
Light.prototype.setZPos = function(z) { this.mPosition[2] = z; };
Light.prototype.getPosition = function() { return this.mPosition; };
Light.prototype.setRadius = function(r) { this.mRadius = r; };
Light.prototype.getRadius = function() { return this.mRadius; };
Light.prototype.setLightTo = function(isOn) { this.mIsOn = isOn; };
Light.prototype.isLightOn = function() { return this.mIsOn; };
```
Creating the LightShader Object

The LightShader object subclasses from the SpriteShader to encapsulate its communication, which handles the WebGL-specific details of passing information to the GLSL shader. This provides the engine with a convenient interface for the shader.

1. Under the src/Engine/Shaders folder, create a new file and name it LightShader.js. Remember to load this new source file in index.html.

2. Now add a constructor in order to initialize the references for the light's variables and to obtain their reference locations within the shader. Remember to inherit from the SpriteShader object.

   ```javascript
   "use strict";
   function LightShader(vertexShaderPath, fragmentShaderPath) {
     // Call super class constructor
     SpriteShader.call(this, vertexShaderPath, fragmentShaderPath);

     // glsl uniform position references
     this.mColorRef = null;
     this.mPosRef = null;
     this.mRadiusRef = null;
     this.mIsOnRef = null;
     this.mLight = null; // <-- this is the light source in the Game Engine

     // create the references to these uniforms in the LightShader
     var shader = this.mCompiledShader;
     var gl = gEngine.Core.getGL();
     this.mColorRef = gl.getUniformLocation(shader, "uLightColor");
     this.mPosRef = gl.getUniformLocation(shader, "uLightPosition");
     this.mRadiusRef = gl.getUniformLocation(shader, "uLightRadius");
     this.mIsOnRef = gl.getUniformLocation(shader, "uLightOn");
   }
   gEngine.Core.inheritPrototype(LightShader, SpriteShader);

3. Provide a basic set function to specify which light the shader should use.

   ```javascript
   LightShader.prototype.setLight = function(l) {
     this.mLight = l;
   };
   ```

4. Override the activateShader() function from the SpriteShader object to add the new functionality of turning the light on and off, as shown here. Notice that you still call the superclass's activateShader() function.

   ```javascript
   LightShader.prototype.activateShader = function (pixelColor, aCamera) {
     // first call the super class's activate
     SpriteShader.prototype.activateShader.call(this, pixelColor, aCamera);
   };
   ```
// now push the light information to the shader
if (this.mLight !== null) {
    this._loadToShader(aCamera);
} else {
    gEngine.Core.getGL().uniform1i(this.mIsOnRef, false); // <-- switch off the light!
}

5. Implement a function to load the light’s properties into the corresponding shader. Recall that this is achieved by using the references created in the constructor and WebGL’s uniform set functions. Also notice that the camera provides the new coordinate space functionality of wcPosToPixel and wcSizeToPixel. The implementation of these functions will be examined shortly.

LightShader.prototype._loadToShader = function(aCamera) {
    var gl = gEngine.Core.getGL();
    gl.uniform1i(this.mIsOnRef, this.mLight.isLightOn());
    if (this.mLight.isLightOn()) {
        var p = aCamera.wcPosToPixel(this.mLight.getPosition());
        var r = aCamera.wcSizeToPixel(this.mLight.getRadius());
        var c = this.mLight.getColor();
        gl.uniform4fv(this.mColorRef, c);
        gl.uniform4fv(this.mPosRef, vec4.fromValues(p[0], p[1], p[2], 1));
        gl.uniform1f(this.mRadiusRef, r);
    }
};

Creating the LightRenderable Object

With the engine’s LightShader object defined to interface to the GLSL LightFS shader, you can now focus on creating a new Renderable object that subclasses from SpriteAnimateRenderable to support the interaction with lights. You can think of this object as a SpriteAnimateRenderable that can be illuminated by a Light object.

1. Begin by creating a new file in the src/Engine/Renderables folder and naming it LightRenderable.js. Remember to load this new source file in index.html.

2. Next, add a constructor to call the superclass’s constructor to set the corresponding light shader and to create a variable for the light that will illuminate this object.

    function LightRenderable(myTexture) {
        SpriteAnimateRenderable.call(this, myTexture);
        Renderable.prototype._setShader.call(this,
            gEngine.DefaultResources.getLightShader());

        // here is the Light source
        this.mLight = null;
    }

gEngine.Core.inheritPrototype(LightRenderable, SpriteAnimateRenderable);
3. Add a draw function that passes the illuminating light source to the LightShader object for communicating with the GLSL fragment shader.

```javascript
LightRenderable.prototype.draw = function(aCamera) {
  this.mShader.setLight(this.mLight);
  SpriteAnimateRenderable.prototype.draw.call(this, aCamera);
};
```

4. Lastly, simply add the support to get and set the light, as shown in the following code:

```javascript
LightRenderable.prototype.getLight = function() {
  return this.mLight;
};
LightRenderable.prototype.addLight = function(l) {
  this.mLight = l;
};
```

### Defining a Default LightShader Instance

You can now modify the engine to support the initializing, loading, and unloading of the new LightShader object.

1. Begin by adding a variable for the light shader in the Engine_DefaultResources.js file located in the src/Engine/Core/Resources folder. Also, define an accessor function as shown here:

   ```javascript
   // Light Shader
   var kLightFS = "src/GLSLShaders/LightFS.glsl";
   var mLighShader = null;

   var getLightShader = function() { return mLighShader; };  
   ```

2. Now instantiate a new light shader in the _createShaders() function, as shown here:

   ```javascript
   var _createShaders = function(callBackFunction) {
     gEngine.ResourceMap.setLoadCompleteCallback(null);
     mConstColorShader = new SimpleShader(kSimpleVS, kSimpleFS);
     mTextureShader = new TextureShader(kTextureVS, kTextureFS);
     mSpriteShader = new SpriteShader(kTextureVS, kTextureFS);
     mLineShader = new LineShader(kSimpleVS, kSimpleFS);
     mLighShader = new LightShader(kTextureVS, kLightFS);
     callBackFunction();
   }
   ```

3. In the initialize() function, add the following code to properly load the file:

   ```javascript
   gEngine.TextFileLoader.loadTextFile(kLightFS,
     gEngine.TextFileLoader.eTextFileType.eTextFile);
   ```
4. In the `cleanUp()` function, add the following line of code to unload the file when it is no longer needed:

   `gEngine.TextFileLoader.unloadTextFile(kLightFS);`

5. Don’t forget to add the get accessor function to `mPublic()` so that it can be accessed.

   `getLightShader: getLightShader,`

### Modifying the Camera

The Camera utility functions, such as `wcPosToPixel()`, are invoked multiple times while rendering the `LightShader` object. These functions compute the transformation between WC and pixel space. This transformation requires the computation of intermediate values (for example, origin of the Camera) that do not change during each rendering invocation. To avoid repeated computation of these values, a per-render invocation cache should be defined for the Camera object.

### Defining a Per-Render Cache for the Camera

Define a per-render cache to store intermediate values that are required to support shading operations.

1. Edit `Camera.js` and define the constructor for the `PerRenderCache` object to hold the ratio between the WC space and the pixel space as well as the origin of the Camera. These are intermediate values required for computing the transformation from WC to pixel space, and these values do not change once a rendering begins.

   ```javascript
   function PerRenderCache() {
     this.mWCToPixelRatio = 1;  // WC to pixel transformation
     this.mCameraOrgX = 1;      // Lower-left corner of camera in WC
     this.mCameraOrgY = 1;
   }
   ```

2. Modify the constructor of the Camera to instantiate a new `PerRenderCache` object. It is important to note that this variable should be used only for rendering purposes. It should not be used for functionality within the game or game engine.

   ```javascript
   function Camera(wcCenter, wcWidth, viewportArray, bound) {
     // WC and viewport position and size
     this.mCameraState = new CameraState(wcCenter, wcWidth);
     this.mCameraShake = null;

     // ...

     // per-rendering cached information
     // needed for computing transforms for shaders
     // updated each time in setupViewProjection()
   }
   ```
this.mRenderCache = new PerRenderCache();
    // SHOULD NOT be used except
    // xform operations during the rendering
    // Client game should not access this!
}

3. Initiate the per-render cache in the setupViewProjection() function by adding
   step B4 to calculate and set the cache using the existing Camera viewport width,
   world width, and world height.

   // Step B4: compute and cache per-rendering information
   this.mRenderCache.mWCToPixelRatio =
       this.mViewport[Camera.eViewport.eWidth] / this.getWCWidth();
   this.mRenderCache.mCameraOrgX = center[0] - (this.getWCWidth()/2);
   this.mRenderCache.mCameraOrgY = center[1] - (this.getWCHeight()/2);

Adding Camera Transform Functions

Now that the per-render cache is defined and properly initialized, you can extend the functionality
of the camera by implementing the functions to convert from WC and pixel space. For code readability
and maintainability, this functionality has been delegated to a separate file because of its specific purpose.
Another important note is that since you are converting from WC to pixel space and pixel space has no
z-axis, you need to calculate a fake z-value for the pixel space coordinate.

1. Under the src/Engine/Cameras folder, create a new file and name it Camera_ Xform.js. Remember to load this new source file in index.html.

2. Approximate a fake pixel space z value by scaling the input parameter according
to the mWCToPixelRatio variable.

   Camera.prototype.fakeZInPixelSpace = function(z) {
       return z * this.mRenderCache.mWCToPixelRatio;
   };

3. Provide a function to convert from WC to pixel space for a vec3 position. This
   is accomplished by subtracting the camera origin followed by scaling with the
   mWCToPixelRatio. The 0.5 offset at the end of the x and y conversion ensure that
   you are working with the center of the pixel rather than a corner.

   Camera.prototype.wcPosToPixel = function(p) {  // p is a vec3, fake Z
       // Convert the position to pixel space
       var x = this.mViewport[Camera.eViewport.eOrgX] +
               ((p[0] - this.mRenderCache.mCameraOrgX) *
                this.mRenderCache.mWCToPixelRatio) + 0.5;
       var y = this.mViewport[Camera.eViewport.eOrgY] +
               ((p[1] - this.mRenderCache.mCameraOrgY) *
                this.mRenderCache.mWCToPixelRatio) + 0.5;
       var z = this.FakeZInPixelSpace(p[2]);
       return vec3.fromValues(x, y, z);};
4. Simply provide a function for converting a length from WC to pixel space by scaling with the mWCToPixelRatio variable.

```javascript
Camera.prototype.wcSizeToPixel = function(s) {
  return (s * this.mRenderCache.mWCToPixelRatio) + 0.5;
};
```

Testing the Light

The MyGame level must be modified to utilize and test the new light functionality.

Modifying the Hero and Minion

Make a few quick modifications to the Hero and Minion objects to accommodate the new LightRenderable object.

1. In the Hero.js file within the src/MyGame/Objects folder, replace the SpriteRenderable instantiation with a LightRenderable instantiation.

   ```javascript
   this.mDye = new LightRenderable(spriteTexture);
   ```

2. In the Minion.js file within the src/MyGame/Objects folder, replace the SpriteRenderable instantiation with a LightRenderable instantiation.

   ```javascript
   this.mMinion= new LightRenderable(spriteTexture);
   ```

Modifying the MyGame Object

With the implementation of the light completed and the game objects properly updated, you can now modify the MyGame level to display and test the light source. Because of the simplistic and repetitive nature of the code in the MyGame.js file of adding variables for the new objects, initializing the objects, drawing the objects, and updating the objects, each line of code changed will not be listed. Rather, you can open the MyGame.js file within the src/MyGame folder and look at changes made in order to test the newly added light source.

Observations

With the project now complete, you can run it and examine the results. There are a few observations to take note of. First is the fact that the illuminated results from the light source look like a circle. As depicted in Figure 8-2, this is the illuminated circle of the point light on the z = 0 plane where your objects are located. Press the Z or X key to increase or decrease the light z position to observe the illuminated circle decreases (smaller intersection area) and increases in size. Alternatively, you can press the C or V key to increase or decrease the point light radius to increase or decrease the volume of illumination, and observe the corresponding changes in the illuminated circle radius. Another observation to take note of is that the light source illuminates the left minion, the hero, and the background but not the other three objects in the scene. This is because the right minion and the two blocks are not LightRenderable objects and thus cannot be illuminated by the defined light source.
Multiple Light Sources and Distance Attenuation

In the previous project, a single point light source was defined with the capability of illuminating a spherical volume. This type of light source is useful in many games, but it is restrictive to limit a game to only a single light source. The engine should support the illumination from multiple light sources to fulfill the design needs of different games. This shortcoming is remedied in the next project with general support for multiple light sources. The implementation principle for multiple lights remains the same as the previous project, with the modification of replacing the single light source with an array of lights. As illustrated in Figure 8-5, a new Light object will be defined, while the LightRenderable object will be modified to support an array of the Light objects. The LightShader object will define an array of ShaderLightAtIndex objects that are capable of communicating light source information to the uLights array in the GLSL LightFS fragment shader for illumination computations.

The point light illumination results from the previous project can be improved. You have observed that the illuminated circle disappears abruptly with a sharp illuminated bright boundary. This sudden disappearance of illumination results does not reflect real life where effects from a given light source decrease gradually over distance instead of switching off abruptly. A more visually pleasing light illumination result should show an illuminated circle where the illumination results at the boundary disappear gradually. This gradual decrease of light illumination effect over distance is referred to as distance attenuation. It is a common practice to approximate distant attenuation with quadratic functions because they produce effects that resemble the real world. In general, distance attenuation can be approximated in many ways, and it is often refined to suit the needs of the game. In addition to distance attenuation, you will also implement a near cutoff distance and a far cutoff distance (that is, two distances from the light source at which the distant attenuation effect will begin and end). These two values give you control over a light source to show a fully illuminated center area with illumination drop-off occurring only at a specified distance. Lastly, a light intensity will be defined to allow dramatically different effects. For example, you can have a soft, barely noticeable light that covers a wide area or an oversaturated glowing light that is concentrated over a small area in the scene.
The Multiple Lights Project

This project demonstrates how to implement multiple point lights within a single scene. It also demonstrates how to increase the complexity of your point lights so that they are more flexible to serve a wider variety of purposes. You can see an example of this project running in Figure 8-6. The source code of this project is located in the Chapter8/8.3.MultipleLights folder.

![Figure 8-6. Running the Multiple Lights project](image)

The controls of the project are as follows:

- **WASD keys**: Move the hero character on the screen
- **Number keys 0, 1, 2, and 3**: Select the corresponding light source
- **Arrow keys**: Move the currently selected light
- **Z/X key**: Increase/decrease the light z position
- **C/V and B/N keys**: Increase/decrease the near and far cutoff distances of the selected light
- **K/L key**: Increase/decrease the intensity of the selected light
- **H key**: Toggles the selected light on/off
The goals of the project are as follows:

- To build the infrastructure for supporting multiple light sources in the engine and in GLSL shaders
- To understand and examine the distance attenuation effects of light
- To experience controlling and manipulating multiple light sources in a scene

You can find the following external resources in the assets folder: the fonts folder that contains the default system fonts and two texture images (minion_sprite.png and bg.png). The objects are sprite elements of minion_sprite.png, and the background is represented by bg.png.

Modifying the GLSL Light Fragment Shader

The LightFS fragment shader needs to be modified to support the distance attenuation, cutoffs, and multiple light sources.

1. In the LightFS.glsl file, remove the light variables that were added for a single light and add a struct for light information that holds the position, color, near distance, far distance, intensity, and on-off variables. With the struct defined, add a uniform array of lights to the fragment shader. Notice that a #define has been added to hold the number of light sources to be used. You can see these changes in the following code.

```
// Light information
#define kGLSLuLightArraySize 4
// GLSL Fragment shader requires loop control
// variable to be a constant number. This number 4
// says, this fragment shader will _ALWAYS_ process
// all 4 light sources.
// ***********WARNING***********************
// This number must correspond to the constant with
// the same name defined in LightShader.js file.
// ***********WARNING************************
// To change this number MAKE SURE: to update the
// kGLSLuLightArraySize
// defined in LightShader.js file.
struct Light {
    vec4 Position;   // in pixel space!
    vec4 Color;
    float Near;      // distance in pixel space
    float Far;       // distance in pixel space
    float Intensity;
    bool  IsOn;
};
uniform Light uLights[kGLSLuLightArraySize];
// Maximum array of lights this shader supports
```

**Note** You can define as many lights as the hardware can support. For example, you can try increasing the number of lights to 50 and then test and measure its performance.
2. Next add a function called `LightEffect()` that takes a light parameter and returns a color as a result. This function calculates the distance between the light and the current fragment and determines whether it lies within the near radius, in between near and far radii, or farther than the far radius. If the fragment’s current position lies within the near radius, there is no attenuation, so a value of 1 is applied. If the fragment’s current position lies in between the near and far radii, then a quadratic attenuation is applied. A distance of greater than the far radius will result in no illumination from the corresponding light source, or a 0 value will be applied. You can see how this is achieved in the following code:

```cpp
vec4 LightEffect(Light lgt) {
    vec4 result = vec4(0);
    float atten = 0.0;
    float dist = length(lgt.Position.xyz - gl_FragCoord.xyz);
    if (dist <= lgt.Far) {
        if (dist <= lgt.Near)
            atten = 1.0;  // no attenuation
        else {
            // simple quadratic drop off
            float n = dist - lgt.Near;
            float d = lgt.Far - lgt.Near;
            atten = smoothstep(0.0, 1.0, 1.0-(n*n)/(d*d)); // blended attenuation
        }
    }
    result = atten * lgt.Intensity * lgt.Color;
    return result;
}
```

3. The main function iterates through all the defined light sources and calls the `LightEffect()` function to calculate and accumulate the contribution from the corresponding light in the array.

```cpp
void main(void) {
    // simple tint based on uPixelColor setting
    vec4 textureMapColor = texture2D(uSampler, vec2(vTexCoord.s, vTexCoord.t));
    vec4 lgtResults = uGlobalAmbientIntensity * uGlobalAmbientColor;

    // now decide if we should illuminate by the light
    if (textureMapColor.a > 0.0) {
        for (int i=0; i<kGLSLuLightArraySize; i++) {
            if (uLights[i].IsOn) {
                lgtResults += LightEffect(uLights[i]);
            }
        }
    }
    lgtResults *= textureMapColor;
}
```
// tint the textured area, and leave transparent area as defined by the texture
vec3 r = vec3(lgtResults) * (1.0-uPixelColor.a) +
vec3(uPixelColor) * uPixelColor.a;
vec4 result = vec4(r, lgtResults.a);

gl_FragColor = result;

Modifying the Light Object
The game engine Light object must be modified to reflect the newly added properties: near and far attenuation and intensity.

1. Modify the Lights.js constructor to include variables for near and far attenuation and intensity. You can see this achieved in the following code:

```javascript
// Constructor
function Light() {
    this.mColor = vec4.fromValues(0.1, 0.1, 0.1, 1); // light color
    this.mPosition = vec3.fromValues(0, 0, 5); // light position in WC
    this.mNear = 5; // within Near is fully lighted
    this.mFar = 10; // farther than Far is not lighted
    this.mIntensity = 1;
    this.mIsOn = true;
}
```

2. Define the get and set accessors for the new variables. Note that the radius variable has been generalized and replaced by the near and far cutoff distances.

```javascript
Light.prototype.setColor = function(c) { this.mColor = vec4.clone(c); };
Light.prototype.getColor = function() { return this.mColor; };

Light.prototype.set2DPosition = function(p) {
    this.mPosition = vec3.fromValues(p[0], p[1], this.mPosition[2]); };
Light.prototype.setXPos = function(x) { this.mPosition[0] = x; };
Light.prototype.setYPos = function(y) { this.mPosition[1] = y; };
Light.prototype.setZPos = function(z) { this.mPosition[2] = z; };
Light.prototype.getPosition = function() { return this.mPosition; };

Light.prototype.setNear = function(r) { this.mNear = r; };
Light.prototype.getNear = function() { return this.mNear; };

Light.prototype.setFar = function(r) { this.mFar = r; };
Light.prototype.getFar = function() { return this.mFar; };

Light.prototype.setIntensity = function(i) { this.mIntensity = i; };
Light.prototype.getIntensity = function() { return this.mIntensity; };

Light.prototype.isLightOn = function() { return this.mIsOn; };
Light.prototype.setLightTo = function(on) { this.mIsOn = on; };
```
Creating the LightSet Object

Under the src/Engine/Lights folder, create a new file and name it LightSet.js. Provide a basic interface for a light set that makes the process of working with the light array more convenient. Remember to load this new source file in index.html.

```javascript
function LightSet() {
    this.mSet = [];
}

LightSet.prototype.NumLights = function() { return this.mSet.length; };

LightSet.prototype.getLightAt = function(index) {
    return this.mSet[index];
};

LightSet.prototype.AddToSet = function(light) {
    this.mSet.push(light);
};

Creating the ShaderLightAtIndex Object

Define the ShaderLightAtIndex object to send information from a Light object to an element in the uLights array in the LightFS GLSL fragment shader.

1. Under the src/Engine/Shaders folder, create a new file and name it ShaderLightAtIndex.js. Remember to load this new source file in index.html.
2. Define a constructor to set the light property references to a specific index in the uLights array in the fragment shader.

```javascript
function ShaderLightAtIndex(shader, index) {
    this._setShaderReferences(shader, index);
}

ShaderLightAtIndex.prototype._setShaderReferences = function(aLightShader, index) {
    var gl = gEngine.Core.getGL();
    this.mColorRef = gl.getUniformLocation(aLightShader, "uLights[" + index + "]\Color");
    this.mPosRef = gl.getUniformLocation(aLightShader, "uLights[" + index + "]\Position");
    this.mNearRef = gl.getUniformLocation(aLightShader, "uLights[" + index + "]\Near");
    this.mFarRef = gl.getUniformLocation(aLightShader, "uLights[" + index + "]\Far");
    this.mIntensityRef = gl.getUniformLocation(aLightShader, "uLights[" + index + "]\Intensity");
    this.mIsOnRef = gl.getUniformLocation(aLightShader, "uLights[" + index + "]\IsOn");
};
```
3. Implement the `loadToShader()` function to push the light’s properties to the GLSL shader. Notice that this function is similar to the previous `loadToShader()` function defined in the `LightShader.js` file.

```javascript
ShaderLightAtIndex.prototype.loadToShader = function (aCamera, aLight) {
    var gl = gEngine.Core.getGL();
    gl.uniform1i(this.mIsOnRef, aLight.isLightOn());
    if (aLight.isLightOn()) {
        var p = aCamera.wcPosToPixel(aLight.getPosition());
        var ic = aCamera.wcSizeToPixel(aLight.getNear());
        var oc = aCamera.wcSizeToPixel(aLight.getFar());
        var c = aLight.getColor();
        gl.uniform4fv(this.mColorRef, c);
        gl.uniform3fv(this.mPosRef, vec3.fromValues(p[0], p[1], p[2]));
        gl.uniform1f(this.mNearRef, ic);
        gl.uniform1f(this.mFarRef, oc);
        gl.uniform1f(this.mIntensityRef, aLight.getIntensity());
    }
};
```

4. Provide a function to update the shader’s on/off property for the light.

```javascript
ShaderLightAtIndex.prototype.switchOffLight = function() {
    var gl = gEngine.Core.getGL();
    gl.uniform1i(this.mIsOnRef, false);
};
```

### Modifying the LightShader Object

You must modify the `LightShader` object to properly handle the communication between the `Light` object and the array of lights in the `LightFS` fragment shader.

1. Begin by editing the `LightShader.js` file and removing the `loadToShader()` function because the actual loading of light information to GLSL shaders will be handled by the newly defined `ShaderLightAtIndex` objects.

2. Modify the constructor to define `mLights`, which is an array of `ShaderLightAtIndex` objects to correspond to the `uLights` array defined in the `LightFS.glsl` and `IllumFS.glsl` fragment shaders. It is important to note that the `mLights` and `uLights` arrays must be the exact same size. You can see this in the following code:

```javascript
function LightShader(vertexShaderPath, fragmentShaderPath) {
    // Call super class constructor
    SpriteShader.call(this, vertexShaderPath, fragmentShaderPath);
    this.mLights = null; // lights from the renderable

    //**********WARNING**********
    // this number MUST correspond to the GLSL uLight[] array size
    // (for LightFS.glsl and IllumFS.glsl)
    //**********WARNING**********
```
this.kGLSLuLightArraySize = 4; // <-- make sure this is 
// the same as LightFS.glsl
this.mShaderLights = [];
for (var i = 0; i < this.kGLSLuLightArraySize; i++) {
    var ls = new ShaderLightAtIndex(this.mCompiledShader, i);
    this.mShaderLights.push(ls);
}
gEngine.Core.inheritPrototype(LightShader, SpriteShader);

3. Modify the activateShader() function to iterate and load the contents of each 
   ShaderLightAtIndex object to the LightFS shader by calling the corresponding 
   loadToShader() function. Recall that the GLSL fragment shader requires the 
for loop control variable to be a constant. This implies that all elements of 
the uLights array will be processed on each LightFS invocation, and thus it is 
important to ensure all unused lights are switched off. This is ensured by the last 
while loop in the following code:

// Overriding the Activation of the shader for rendering 
LightShader.prototype.activateShader = function(pixelColor, aCamera) {
    // first call the super class's activate 
    SpriteShader.prototype.activateShader.call(this, pixelColor, aCamera);

    // now push the light information to the shader 
    var numLight = 0;
    if (this.mLights !== null) {
        while (numLight < this.mLights.length) {
            this.mShaderLights[numLight].loadToShader(aCamera, this.mLights[numLight]);
            numLight++;
        }
    }
    // switch off the left over ones. 
    while (numLight < this.kGLSLuLightArraySize) {
        this.mShaderLights[numLight].switchOffLight(); // switch off unused lights 
        numLight++;
    }
};

4. Make a simple modification to the setLight() function so that it becomes 
   setLights() and handles the array rather than the single light.

LightShader.prototype.setLights = function(l) {
   this.mLights = l;
};
Modifying the LightRenderable Object

You can now modify the LightRenderable object to support multiple light sources.

1. In the LightRenderable constructor, replace the single light reference variable with an array.

   function LightRenderable(myTexture) {
     SpriteAnimateRenderable.call(this, myTexture);
     Renderable.prototype._setShader.call(this, gEngine.DefaultResources.getLightShader());

     // here is the light source
     this.mLights = []; // allocates a new array
   }

2. Make sure to update the draw function to reflect the change to multiple light sources.

   LightRenderable.prototype.draw = function(aCamera) {
     this.mShader.setLights(this.mLights);
     SpriteAnimateRenderable.prototype.draw.call(this, aCamera);
   }

3. Define the corresponding accessor functions for the light array.

   LightRenderable.prototype.getLightAt = function(index) {
     return this.mLights[index];
   }
   LightRenderable.prototype.addLight = function(l) {
     this.mLights.push(l);
   }

Testing the Light Sources with MyGame

With multiple lights support properly integrated in the engine, you can now modify MyGame to test your implementation and examine the results. In addition to adding multiple lights to the scene, you will be adding the ability to control the properties of each light. Because of the scope of this MyGame, you will divide the light instantiation and controls into separate files to maintain the readability of the source code. To avoid redundancy and repetitive code listings, the details to the straightforward implementations are not shown.

1. Modify the MyGame.js file in the src/MyGame folder to reflect the changes to the constructor, initialize function, draw function, and update function. All these changes revolve around handling multiple lights through a light set.

2. In the src/MyGame folder, create the new file MyGame_Lights.js to instantiate and initialize the lights. Remember to load this new source file in index.html.

3. In the src/MyGame folder, create the new file MyGame_lightControls.js to implement the controls of the lights. Remember to load this new source file in index.html.
Observations
Run the project to examine the implementation. Try selecting the lights with the 0, 1, 2, and 3 keys and toggling the selected light on/off. Notice that all lights illuminate the background, while the hero is illuminated only by lights 0 and 3, the left minion is illuminated only by lights 1 and 3, and the right minion is illuminated only by lights 2 and 3. Move the Hero object with the WASD keys to observe how the illumination on the object changes as she is moved through the near and far radii of light source 0. Select and move the light sources with the arrow keys to observe the additive property of lights. Experiment with changing the light source’s z position and its near/far values to observe how similar illumination effects can be accomplished with different z/near/far settings. The two constant color squares are in the scene to confirm that nonilluminated objects can still be rendered.

Diffuse Reflection and Normal Mapping
You can now place or move many light sources and control the illumination or shading at localized regions. However, if you run the previous project and move the lights around while paying attention to the vertical boundaries of the geometric blocks in the background, you will notice the peculiar effect that the illumination along these boundaries changes uniformly when a light position moves across it. You are observing boundary surfaces being illuminated by light sources that seem to be spatially behind the surface! Although visually odd, this is to be expected in a 2D world. The vertical boundaries are only artist renditions, and your illumination calculation does not consider the geometric contours suggested by the image content. This restriction of illumination in a flat 2D world is remedied in this section with the introduction of diffuse reflection and normal mapping to approximate normal vectors of surfaces.

As illustrated by the left drawing in Figure 8-7, a surface normal vector, a surface normal, or a normal vector is the vector that is perpendicular to a given surface element. The right drawing of Figure 8-7 shows that in 3D space the surface normal vectors of an object describe the shape or contour of the object.

A human’s observation of light illumination is the result of visible energy from the light sources reflecting off object surfaces and reaching the eyes. A diffuse, or Lambertian, surface reflects light energy uniformly in all directions. Examples of diffuse surfaces include typical printer papers or matte painted surfaces. Figure 8-8 shows a light source illuminating three diffuse surface element positions, A, B, and C. First, notice that the direction from the position being illuminated toward the light source is defined as the position’s light vector, \( \hat{L} \). It is important to note that the direction of the \( \hat{L} \) vector is always toward the light source and that this is a normalized vector with a magnitude of 1. Figure 8-8 illustrates the diffuse illumination, or magnitude of diffuse reflection, with examples. Position A cannot receive any energy from the given light source because its normal vector, \( \hat{N} \), is perpendicular to its light vector \( \hat{L} \), or \( \hat{N} \cdot \hat{L} = 0 \). Position B can receive all the energy because its normal vector is pointing in the same direction as its light vector, or \( \hat{N} \cdot \hat{L} = 1 \). In general, as exemplified by position C, the proportion of light energy received and reflected by a diffuse surface is proportional to the cosine of the angle between its normal and light vector, or \( \hat{N} \cdot \hat{L} \).
Displaying the \( \hat{N} \cdot \hat{L} \) computation results, or the diffuse component, cues 3D shape contours for the human vision system. For example, Figure 8-9 shows a sphere and torus (doughnut shape object) with (the left image) and without (the right image) the corresponding diffuse components. Clearly, in both cases, the 3D contour of the objects are captured by the left versions of the image with the \( \hat{N} \cdot \hat{L} \) computation.

In a 2D world, as in the case of your game engine, all objects are represented as 2D images, or textures. Since all objects are 2D textured images defined on the xy plane, the normal vectors for all the objects are the same: the vector in the z direction. This lack of distinct normal vectors for objects implies that it is not possible to compute the diffuse component of objects. Fortunately, similar to how texture mapping addresses the limitation of each geometry having only a single color, normal mapping can resolve the problem of each geometry having only a single normal vector. Figure 8-10 shows the idea behind normal mapping where in addition to the color texture image a corresponding normal texture image is required. The left image of Figure 8-10 is a typical color texture image, and the two right images are zoomed images of the highlighted square on the left image. Notice once again that two images are involved in normal mapping: the color texture image where the RGB channels of the texture record the color of objects (bottom of the right image of Figure 8-10) and a corresponding normal texture image where the RGB channels record the x, y, and z values of the normal vector for the corresponding object in the color texture (top of the right image).
Figure 8-10. Normal mapping with two texture images: the normal and the color texture

Figure 8-11 captures the view of the three corresponding positions labeled on the right images of Figure 8-10 (the positions $n_1$, $n_2$, and $n_3$ on the normal texture and the corresponding positions $c_1$, $c_2$, and $c_3$ on the color texture) to illustrate the details of normal mapping. The bottom layer of Figure 8-11 shows that the color texture records colors and the colors $c_1$, $c_2$, and $c_3$ are sampled at those three positions. The middle layer of Figure 8-11 shows that the RGB components of the normal texture records the normal vector $xyz$ values of objects at the corresponding color texture positions. The top layer of Figure 8-11 shows that when illuminated by a light source with the $\hat{N} \cdot \hat{L}$ term properly computed and displayed, the human vision system will perceive a sloped contour.

Figure 8-11. Normal mapping with two texture images: the normal and the color texture
In summary, a normal texture map or a normal map is a texture map that stores normal vector information rather than the usual color information. Each texel of a normal map encodes the xyz values of a normal vector in the RGB channels. In lieu of displaying the normal map texels as you would with a color texture, the texels are used purely for calculating how the surface would interact with light. In this way, instead of a constant normal vector pointing in the 𝑧 direction, when a square is normal mapped, the normal vector of each pixel being rendered will be defined by texels from the normal map and used for computing the diffuse component. When computing light illumination, the \( \hat{\mathbf{N}} \cdot \hat{\mathbf{L}} \) computation is referred to as diffuse computation, lighting, or illumination. The mathematic term \( \hat{\mathbf{N}} \cdot \hat{\mathbf{L}} \) is referred to as the diffuse term.

In the previous project, you expanded the engine to support multiple light sources with individual light to model the real world closely. In this section, you will define the \texttt{IllumShader} object to generalize the \texttt{LightShader} object to support the computation of the diffuse component based on normal mapping.

The Normal Maps and Illumination Shaders Project

This project demonstrates how to integrate normal mapping into your game engine and use the results to compute the diffuse component of objects. You can see an example of this project running in Figure 8-12. The source code of this project is located in the Chapter8/8.4.NormalMapsAndIlluminationShaders folder.

![Figure 8-12. Running the Normal Maps and Illumination Shaders project](image)
The controls of the project are as follows:

- **WASD keys**: Move the hero character on the screen
- **Number keys 0, 1, 2, and 3**: Select the corresponding light source
- **Arrow keys**: Move the currently selected light
- **Z/X key**: Increases/decreases the light z position
- **C/V and B/N keys**: Increases/decreases the near and far cutoff distances of the selected light
- **K/L key**: Increases/decreases the intensity of the selected light
- **H key**: Toggles the selected light on/off

The goals of the project are as follows:

- To understand and work with normal maps
- To implement normal maps as textures in the game engine
- To implement GLSL shaders that support diffuse component illumination
- To examine the diffuse component illumination of $\hat{N} \cdot \hat{L}$

You can find the following external resource files in the assets folder: the fonts folder that contains the default system fonts, two texture images, and two corresponding normal maps for the texture images (minion_sprite.png and bg.png) and the corresponding normal maps: minion_sprite_normal.png and bg_normal.png. As in previous projects, the objects are sprite elements of minion_sprite.png, and the background is represented by bg.png.

---

**Note** The minion_sprite_normal.png normal map is generated automatically based on the minion_sprite.png image from http://cpetry.github.io/NormalMap-Online/.

---

**Creating the GLSL Illumination Fragment Shader**

As with the previous projects, your normal map integration will begin with the implementation of the GLSL shader. Note that this new shader will be remarkably similar to your LightFS.glsl but with the inclusion of normal mapping and diffuse computation support. To ensure the support for simple lighting without normal mapping, you will create a new GLSL fragment shader.

1. Begin by copying and pasting LightFS.glsl and naming the new file IllumFS.glsl within the src/GLSLShaders folder.
2. Edit the IllumFS.glsl file and add a sampler2D object, uNormalSampler, to sample the normal map.

```glsl
precision mediump float;

// The object that fetches data from texture.
// Must be set outside the shader.
uniform sampler2D uSampler;
uniform sampler2D uNormalSampler;
```
// Color of pixel
uniform vec4 uPixelColor;
uniform vec4 uGlobalAmbientColor; // this is shared globally
uniform float uGlobalAmbientIntensity;

// Light information
#define kGLSLuLightArraySize 4
struct Light {
  vec4 Position;  // in pixel space!
  vec4 Color;
  float Near;     // distance in pixel space
  float Far;      // distance in pixel space
  float Intensity;
  bool IsOn;
};
uniform Light uLights[kGLSLuLightArraySize];
  // Maximum array of lights this shader supports

varying vec2 vTexCoord;

3. Modify the LightEffect() function to receive a normal vector parameter, N. This normal vector N is assumed to be normalized with a magnitude of 1 and will be used in the diffuse component \( \hat{N} \cdot \hat{L} \) computation. Include code to compute the \( \hat{L} \) vector, remember to normalize the vector, and use the result of \( \hat{N} \cdot \hat{L} \) to scale the light attenuation accordingly as follows:

```glsl
vec4 LightEffect(Light lgt, vec3 N) {
  vec4 result = vec4(0);
  float atten = 0.0;
  vec3 L = lgt.Position.xyz - gl_FragCoord.xyz;
  float dist = length(L);
  if (dist <= lgt.Far) {
    if (dist <= lgt.Near)
      atten = 1.0;  //  no attenuation
    else {
      // simple quadratic drop off
      float n = dist - lgt.Near;
      float d = lgt.Far - lgt.Near;
      atten = smoothstep(0.0, 1.0, 1.0-(n*n)/(d*d)); // blended attenuation
    }
  } else {
    // To normalize L
    vec3 L = L / dist;
    // Not calling the normalize() function to avoid re-computing
    // the "dist". This is computationally more efficient.
    float NdotL = max(0.0, dot(N, L));
    atten *= NdotL;
  }
  result = atten * lgt.Intensity * lgt.Color;
  return result;
}
```
4. Edit the main function to sample from both the color texture with uSampler and the normal texture with uNormalSampler. Remember that the normal map provides you with a vector that represents the normal vector direction of the surface element at that given point. Because the xyz normal vector values are stored in the 0 to 1 RGB color format, the sampled normal map results must be scaled and offset to the -1 to 1 range. In addition, recall that texture uv coordinates can be defined with the v direction increasing upward or downward. In this case, depending on the v direction of the normal map, you may also have to flip the y direction of the sampled normal map values. The normalized normal vector, $N$, is then passed on to the LightEffect() function for the illumination calculations.

**Note** Normal maps can be created in a variety of different layouts where x or y might need to be flipped in order to properly represent the surface geometries desired. This entirely depends upon the tool or artist that created the map.

```c
void main(void) {
    // simple tint based on uPixelColor setting
    vec4 textureMapColor = texture2D(uSampler, vTexCoord);
    vec4 normal = texture2D(uNormalSampler, vTexCoord);
    // using the same coordinate as the sprite texture!
    vec4 normalMap = (2.0 * normal) - 1.0;

    // normalMap.y = -normalMap.y; // flip Y
    // depending on the normal map you work with, this may or may not be flipped
    //
    vec3 N = normalize(normalMap.xyz);

    vec4 lgtResult = uGlobalAmbientColor * uGlobalAmbientIntensity;

    // now decide if we should illuminate by the light
    if (textureMapColor.a > 0.0) {
        for (int i=0; i<kGLSLuLightArraySize; i++) {
            if (uLights[i].IsOn) {
                lgtResult += LightEffect(uLights[i], N);
            }
        }
    }
    lgtResult *= textureMapColor;

    // tint the textured area, and
    // leave transparent area as defined by the texture
    vec3 r = vec3(lgtResult) * (1.0-uPixelColor.a) +
            vec3(uPixelColor) * uPixelColor.a;
    vec4 result = vec4(r, lgtResult.a);

    gl_FragColor = result;
}
```
Creating the **IllumShader** Object

With the GLSL shader now supporting normal maps, you can create the JavaScript **IlluminationShader** object to interface to it.

1. Under the `src/Engine/Shaders` folder, create a new file and name it `IllumShader.js`. Remember to load this new source file in `index.html`.

2. Create the constructor by first subclassing from the **LightShader** to take advantage of the functionality related to light sources and define a variable, `mNormalSamplerRef`, to maintain a reference to the normal sampler in the GLSL shader.

   ```javascript
   function IllumShader(vertexShaderPath, fragmentShaderPath) {
     // Call super class constructor
     LightShader.call(this, vertexShaderPath, fragmentShaderPath);

     // reference to the normal map sampler
     var gl = gEngine.Core.getGL();
     this.mNormalSamplerRef = gl.getUniformLocation(
       this.mCompiledShader, "uNormalSampler");
   }
   gEngine.Core.inheritPrototype(IllumShader, LightShader);
   ```

3. Override and extend the `activateShader()` function by binding the normal texture sampler reference to WebGL texture unit 1. So far, you have been working with the color texture sampler that is bounded to the default texture unit of 0. In this way, the WebGL texture system can work with two active textures: units 0 and 1. As will be discussed, the **TextureShader** object must now explicitly bind the color texture to unit 0, and in `gEngine.Textures`, it is important to configure the WebGL to activate the appropriate texture units for the corresponding purpose: color versus normal texture mapping.

   ```javascript
   IllumShader.prototype.activateShader = function(pixelColor, aCamera) {
     // first call the super class's activate
     LightShader.prototype.activateShader.call(this, pixelColor, aCamera);
     var gl = gEngine.Core.getGL();
     gl.uniform1i(this.mNormalSamplerRef, 1); // binds to texture unit 1
     // do not need to set up texture coordinate buffer
     // as we are going to use the ones from the sprite texture
     // in the fragment shader
   };
   ```

**Note**  WebGL supports simultaneous activation of multiple texture units during rendering. Depending on the graphics card capability, up to 32 texture units can be active simultaneously during a single rendering pass. In this book, you will activate only two of the texture units during rendering: one for color texture and the other for normal texture.
Modifying the TextureShader Object

So far, you have been working with the default binding of the color texture map to WebGL texture unit 0. With the addition of the normal texture, you now need to explicitly bind your color texture to WebGL texture unit 0. Fortunately, this is a straightforward change.

1. Modify the constructor to include and initialize a reference to the sampler location, as shown in the following code:

```javascript
function TextureShader(vertexShaderPath, fragmentShaderPath) {
    // Call super class constructor
    SimpleShader.call(this, vertexShaderPath, fragmentShaderPath);

    // reference to aTextureCoordinate within the shader
    this.mShaderTextureCoordAttribute = null;
    this.mSamplerRef = null; // reference to the uSampler, when using only texture,
    // this is not necessary,
    // with NormalMap, we must do this.

    // get the reference of uSampler and aTextureCoordinate within the shader
    var gl = gEngine.Core.getGL();
    this.mSamplerRef = gl.getUniformLocation(this.mCompiledShader, "uSampler");
    this.mShaderTextureCoordAttribute =
        gl.getAttribLocation(this.mCompiledShader, "aTextureCoordinate");
}
```

2. Add a line to the activateShader() function to bind the texture to unit 0, as shown in the following code:

```javascript
// Overriding the Activation of the shader for rendering
TextureShader.prototype.activateShader = function(pixelColor, aCamera) {
    // first call the super class’s activate
    SimpleShader.prototype.activateShader.call(this, pixelColor, aCamera);

    // now our own functionality: enable texture coordinate array
    var gl = gEngine.Core.getGL();
    gl.bindBuffer(gl.ARRAY_BUFFER, gEngine.VertexBuffer.getGLTexCoordRef());
    gl.enableVertexAttribArray(this.mShaderTextureCoordAttribute);
    gl.vertexAttribPointer(this.mShaderTextureCoordAttribute, 2, gl.FLOAT,
        false, 0, 0);
    gl.uniform1i(this.mSamplerRef, 0); // <-- binds to texture unit 0
};
```

Creating the IllumRenderable Object

You can now create your renderable object to leverage the illumination shader.

1. Begin by creating a new file under the src/Engine/Renderables folder and naming it IllumRenderable.js. Remember to load this new source file in index.html.
2. Create a constructor to subclass from the LightRenderable object and define a mNormalMap instance variable to record the normal map ID. The IllumRenderable object works with two texture maps: myTexture for color texture map and myNormalMap for normal mapping. Note that these two texture maps share the same texture coordinates defined in mTexCoordBuffer in the SpriteShader superclass. This assumes that the geometry of the object is depicted in the color texture map and the normal texture map is derived to capture the contours of the object, which is almost always the case.

```javascript
function IllumRenderable(myTexture, myNormalMap) {
    LightRenderable.call(this, myTexture);
    Renderable.prototype._setShader.call(this, gEngine.DefaultResources.getIllumShader());

    // here is the normal map resource id
    this.mNormalMap = myNormalMap;

    // Normal map texture coordinate will reproduce the corresponding sprite sheet
    // This means, the normal map MUST be based on the sprite sheet
}
gEngine.Core.inheritPrototype(IllumRenderable, LightRenderable);
```

3. Next override the draw() function to activate the normal map before calling the base class's draw() method.

```javascript
IllumRenderable.prototype.draw = function (aCamera) {
    gEngine.Textures.activateNormalMap(this.mNormalMap);
    // Here the normal map texture coordinate is copied from those of
    // the corresponding sprite sheet
    LightRenderable.prototype.draw.call(this, aCamera);
};
```

**Modifying the Engine**

You have to update the engine to support the new texture map, shader, and renderable objects.

**Defining the Default in Engine_DefaultResources**

You must modify the Engine_DefaultResources.js file to define the default instance of the IllumShader object.

1. Define a constant file path and variable for the newly created fragment shader.

```javascript
// Illumination Shader
var kIllumFS = "src/GLSLShaders/IllumFS.glsl";
// Path to the Illumination FragmentShader
var mIllumShader = null;
```
2. Modify the _createShaders() function to instantiate an IllumShader object.

```javascript
var _createShaders = function(callBackFunction) {
    //...
    mIllumShader = new IllumShader(kIllumVS, kIllumFS);
    callBackFunction();
};
```

3. Add a simple accessor for the illumination shader.

```javascript
var getIllumShader = function() { return mIllumShader; };
```

4. Modify the initialize() function to load the text file that defines the illumination shader.

```javascript
var initialize = function(callBackFunction) {
    //...
    // Illumination Shader
    gEngine.TextFileLoader.LoadTextFile(kIllumFS,
        gEngine.TextFileLoader.eTextFileType.eTextFile);
    // load default font
    gEngine.Fonts.LoadFont(kDefaultFont);
    gEngine.ResourceMap.setLoadCompleteCallback(function() {
        _createShaders(callBackFunction);});
}
```

5. Modify the cleanUp() function to unload the text file that defines the illumination shader.

```javascript
// Unload all resources
var cleanUp = function() {
    //...
    mIllumShader.cleanUp();
    // Illumination Shader
    gEngine.TextFileLoader.unloadTextFile(kIllumFS);
    // default font
    gEngine.Fonts.unloadFont(kDefaultFont);
};
```

6. Export the get accessor in the public function list.

```javascript
var mPublic = {
    initialize: initialize,
    getConstColorShader: getConstColorShader,
    // ...
    getIllumShader: getIllumShader,
    getDefaultFont: getDefaultFont,
    // ...
};
```
Configuring WebGL Texture Units in Engine_Textures

The engine texture support in the Engine_Textures.js file must be updated to support configuring the WebGL texture units accordingly: color texture binding to unit 0 and normal texture binding to unit 1.

1. Add a line to the activateTexture() function to specify the texture unit 0. Recall that this function activates the color texture mapping functionality. The gl.TEXTURE0 constant informs WebGL to bind to the texture unit 0.

   var activateTexture = function(textureName) {
     var gl = gEngine.Core.getGL();
     var texInfo = gEngine.ResourceMap.retrieveAsset(textureName);

     // Binds our texture reference to the current webGL texture functionality
     gl.activeTexture(gl.TEXTURE0);
     gl.bindTexture(gl.TEXTURE_2D, texInfo.mGLTexID);

     //...  
   };

2. Add a function to activate the normal map texture and bind this texture to WebGL texture unit 1 with the TEXTURE1 constant.

   // texture 1 is always normal map for this game engine
   var activateNormalMap = function(textureName) {
     var gl = gEngine.Core.getGL();
     var texInfo = gEngine.ResourceMap.retrieveAsset(textureName);

     // Binds our texture reference to the current webGL texture functionality
     gl.activeTexture(gl.TEXTURE1);
     gl.bindTexture(gl.TEXTURE_2D, texInfo.mGLTexID);

     // To prevent texture wrappings
     gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.CLAMP_TO_EDGE);
     gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T, gl.CLAMP_TO_EDGE);

     // Handles how magnification and minimization filters will work.
     gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.LINEAR);
     gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR_MIPMAP_LINEAR);
   };

3. Remember to export the activation of the normal map in the public function list.

   var mPublic = {
     // ...
     activateTexture: activateTexture,
     activateNormalMap: activateNormalMap,
     // ...
   };


Testing the Normal Map

Testing the newly integrated normal map functionality must include the verification that the non-normal mapped simple color texture is working correctly. To accomplish this, the background, hero, and left minion will be created as the newly defined IllumRenderable object, while the right minion will remain a LightRenderable object.

Modifying the Hero and the Minion

The Hero and Minion objects should be modified to support the newly defined IllumRenderable object.

1. Modify the Hero’s constructor to utilize the IllumRenderable.

```javascript
function Hero(spriteTexture, normalMap) {
this.kDelta = 0.3;

this.mDye = new IllumRenderable(spriteTexture, normalMap);
this.mDye.setColor([1, 1, 1, 0]);
this.mDye.getXform().setPosition(15, 50);
this.mDye.getXform().setSize(18, 24);
this.mDye.setElementPixelPositions(0, 120, 0, 180);
GameObject.call(this, this.mDye);
}
```

2. Modify the Minion’s constructor to utilize the IllumRenderable, and notice that depending on whether a normal texture map is present, a Minion can be either an IllumRenderable or a LightRenderable.

```javascript
function Minion(spriteTexture, normalMap, atX, atY) {
this.kDelta = 0.2;

if (normalMap === null) {
    this.mMinion = new LightRenderable(spriteTexture);
} else {
    this.mMinion = new IllumRenderable(spriteTexture, normalMap);
}

this.mMinion.setColor([1, 1, 1, 0]);
this.mMinion.getXform().setPosition(atX, atY);
this.mMinion.getXform().setSize(18, 14.4);
// first element pixel position:
//    top-right 512 is top of image, 0 is right of image
this.mMinion.setSpriteSequence(512, 0,
    204, 164, // widthxheight in pixels
    5,      // number of elements in this sequence
    0);     // horizontal padding in between
this.mMinion.setAnimationType(SpriteAnimateRenderable.eAnimationType.eAnimateSwing);
this.mMinion.setAnimationSpeed(30);
// show each element for mAnimSpeed updates
GameObject.call(this, this.mMinion);
}
```
Modifying MyGame

You can now modify MyGame to test and display your implementation of the illumination shader. Modify the MyGame.js file in the src/MyGame folder to load and unload the new normal maps and to create the Hero and Minion objects with the normal map files. As previously, the involved changes are straightforward and relatively minimum; as such, the details are not shown here.

Observations

With the project now complete, you can run it and check your results to observe the effects of diffuse illumination. Notice that the Hero, left Minion, and the background objects are illuminated with a diffuse computation and appear to provide more depth from the lights. There is much more variation of colors and shades across these objects (most notably, in the background where the composing geometric blocks now appear to be individually defined 3D shapes). The fact that the normal maps for the Hero and left Minion objects are generated automatically can be observed with their slightly pixelated and rough appearances. Select one of the light sources, such as light 2, and move the light position with the arrow keys. Take note of the boundary edges of the geometric blocks in the background image; these edges are surfaces facing either horizontally or vertically, whereas the corresponding normal vector directions point either toward the x or y direction. As the light position moves across such a boundary, the sign of the $\hat{N} \cdot \hat{L}$ term would flip, and the corresponding surface illumination would undergo drastic changes (from dark to lit, or vice versa). In this way, with the normal map and diffuse computation, you have turned a static background image into a background that is defined by complex 3D geometric shapes. Try moving the other light sources and observe the illumination changes on all the objects as the light sources move across them.

Specular Reflection and Materials

The diffuse lighting you have implemented is suitable for simulating the illumination of matted surfaces such as typical printer papers, many painted interior walls, or even a traditional blackboard. The Phong illumination model extends this simple diffuse lighting by introducing a specular term to simulate the reflection of the light source across a shiny surface. Figure 8-13 illustrates that given a shiny or reflective surface like a polished floor or plastic, the reflection of the light source will be visible when the eye, or the camera, is in the reflection direction of the light source. This reflection of the light source across shiny surface is referred to as specular reflection, specular highlight, or specularity.

![Figure 8-13. Specularity: the reflection of the light source](image)

\[ \hat{N} = \text{Normal Vector} \]
\[ \hat{V} = \text{View Vector} \]
\[ \hat{L} = \text{Light Vector} \]
From real-life experience, you know that specular highlights are visible even when the eye’s viewing direction is not perfectly aligned with the reflection direction of the light source. As illustrated in Figure 8-14, where the \( \hat{R} \) vector is the reflection direction of the light vector \( \hat{L} \), the specular highlight on an object is visible even when the viewing direction \( \hat{V} \) is not perfectly aligned with the \( \hat{R} \) vector. Real-life experience also informs you that the further away \( \hat{V} \) is from \( \hat{R} \), the less likely you will observe the light reflection. In fact, you know that when \( \alpha \), the angle between \( \hat{V} \) and \( \hat{R} \), is zero, you would observe the maximum light reflection, and when \( \alpha \) is 90° or when \( \hat{V} \) and \( \hat{R} \) are perpendicular, you would observe zero light reflection.

The Phong illumination model simulates the characteristic of specularity with a \( \hat{V} \times (\hat{V} \cdot \hat{R})^n \) term. When \( \hat{V} \) and \( \hat{R} \) are aligned, or when \( \alpha=0° \), the specularity term evaluates to 1, and the term drops off to 0 according to the cosine function when the separation between \( \hat{V} \) and \( \hat{R} \) increases to 90° or when \( \alpha=90° \). The power \( n \), referred to as shininess, describes how rapidly the specular highlight will roll off. The larger the \( n \) value, the faster the cosine function decreases as \( \alpha \) increases, the faster the specular highlight drops off, and the glossier the surface would appear. For example, in Figure 8-15, the right sphere has a \( n \) value of 0, the middle sphere has a \( n \) value of 5, and the right sphere’s \( n \) value is 30.

While the \( (\hat{V} \cdot \hat{R})^n \) term models specular highlight effectively, the cost involved in computing the \( \hat{R} \) vector for every shaded pixel can be significant. As illustrated in Figure 8-16, \( \hat{H} \) is the halfway vector, which is the vector halfway between the \( \hat{L} \) and \( \hat{V} \) vectors. It is observed that \( \beta \), the angle between the \( \hat{N} \) and \( \hat{H} \),
can also be used to characterize specular reflection. Though slightly different, \((\hat{N} \cdot \hat{H})^n\) produces similar results as \((\hat{V} \cdot \hat{R})^n\) with less per-pixel computation cost in computing the \(\hat{H}\) vector. The halfway vector will be used to approximate specularity in your implementation.

**Figure 8-16.** The halfway vector

As illustrated in Figure 8-17, the variation of the Phong illumination model that you will implement consists of simulating the interaction of three participating elements in the scene through three distinct terms. The three participating elements are the global ambient lighting, the light source, and the material property of the object to be illuminated. The previous examples have explained the first two participating elements: the global ambient lighting and the light source. The materials property of an object are represented by \(K_a, K_d, K_s\) and \(n\). These stand for three colors, representing the ambient, diffuse, and specular reflectivity, and a floating-point number representing the shininess of an object. The three terms of the Phong illumination model are as follows:

- **The ambient term:** \(K_a + I_a C_a\)
- **The diffuse term:** \(I_c C_t K_d \hat{N} \cdot \hat{L}\)
- **The specular term:** \(I_c C_t K_s (\hat{N} \cdot \hat{H})^n\)

**Figure 8-17.** The Phong illumination model
Note that the first two terms, the ambient and diffuse terms, have been covered in the previous examples. The IllumFS GLSL fragment shader from the previous example implements these two terms with a light distance attenuation and without the $K_a$ and $K_d$ material properties. This project guides you to build the support for per-object material property and complete the Phong illumination model implementation in the IllumFS GLSL shader with the engine support in the IllumShader/IllumRenderable object pair.

Integration of Material in the Game Engine and GLSL Shaders

To implement the Phong illumination model, a Material object that corresponds to the surface material property in Figure 8-17 must be defined and referenced by each IllumRenderable object that is to be shaded by the corresponding IllumFS GLSL shader. Figure 8-18 illustrates that in your implementation a new ShaderMaterial object will be defined and referenced in the IllumShader to load the content of the Material object to the IllumFS GLSL fragment shader.

![Figure 8-18. Support for material](image)

The Material and Specularity Project

This project demonstrates the implementation of the Phong illumination model utilizing the normal map and the camera’s position. It also implements a system that stores and forwards per-Renderable object material properties to the GLSL shader for the Phong lighting computation. You can see an example of the project running in Figure 8-19. The source code of this project is located in the Chapter8/8.5.MaterialAndSpecularity folder.
The controls of the project are as follows:

- **WASD keys**: Move the hero character on the screen
- **Lighting controls**:
  - *Number keys 0, 1, 2, and 3*: Select the corresponding light source
  - *Arrow keys*: Move the currently selected light
  - *Z/X key*: Increases/decreases the light z position
  - *C/V and B/N keys*: Increases/decreases the near and far cutoff distances of the selected light
  - *K/L key*: Increases/decreases the intensity of the selected light
  - *H key*: Toggles the selected light on/off
- **Material property controls**:
  - *Number keys 5 and 6*: Select the left minion and the hero
  - *Number keys 7, 8, and 9*: Select the : $K_a$, $K_d$, and $K_s$ material properties of the selected character (left minion or the hero)
Chapter 8 ■ Implementing Illumination and Shadow

- **E/R, T/Y, and U/I keys**: Increase/decrease the red, green, and blue channels of the selected material property
- **O/P keys**: Increase/decrease the shininess of the selected material property

The goals of the project are as follows:

- To understand specular reflection and the Phong specular term
- To implement specular highlight illumination in GLSL shaders
- To understand and experience the control of material of illuminated objects
- To examine specular highlights in illuminated images

In the assets folder you can find the same set of external resource files as in the previous project: the fonts folder that contains the default system fonts, two texture images, two corresponding normal maps for the texture images (minion_sprite.png and bg.png), and the corresponding normal maps: (minion_sprite_normal.png, and bg_normal.png). As in previous projects, the objects are sprite elements of minion_sprite.png, and the background is represented by bg.png.

Modifying the GLSL Illumination Fragment Shader

As in the previous projects, you will begin with implementing the actual illumination model in the GLSL fragment shader.

1. Edit the IllumFS.glsl file and define a variable, uCameraPosition, for storing the camera position. This position is used to compute the $V$ vector. In addition, create a material struct and a corresponding variable, uMaterial, for storing the per-object material properties. Note the correspondence between the variable names $K_a$, $K_d$, $K_s$, and $n$ and the terms in the Phong illumination model in Figure 8-17.

   ```
   uniform vec3 uCameraPosition; // for computing the V vector
   // material properties
   struct Material {
     vec4 Ka;    // simple boosting of color
     vec4 Kd;    // Diffuse
     vec4 Ks;    // Specular
     float Shininess; // this is the "n"
   };
   uniform Material uMaterial;
   ```

2. As in the previous project, create the LightAttenuation() function to calculate the distance attenuation of the light.

   ```
   // Computes the L-vector, and returns attenuation
   float LightAttenuation(Light lgt, dist) {
     float atten = 0.0;
     if (dist <= lgt.Far) {
       if (dist <= lgt.Near)
         atten = 1.0; // no attenuation
   ```
else {
    // simple quadratic drop off
    float n = dist - lgt.Near;
    float d = lgt.Far - lgt.Near;
    atten = smoothstep(0.0, 1.0, 1.0-(n*n)/(d*d)); // blended attenuation
}
return atten;
}

3. Define the SpecularResult() and DiffuseResults() functions to calculate the specular and diffuse terms. The \( \hat{V} \) vector, \( V \), is computed by subtracting \( u\text{CameraPosition} \) from the current fragment coordinate, \( gl\_FragCoord \). It is important to observe that this operation is performed in the pixel space, and the IllumShader/IllumRenderable object pair must transform the WC camera position to pixel space before sending over the information. In addition, notice that the texture map color is accumulated in the diffuse and not the specular term.

```cpp
vec4 SpecularResult(vec3 N, vec3 L) {
    vec3 V = normalize(uCameraPosition - gl_FragCoord.xyz);
    vec3 H = (L + V) * 0.5;
    return uMaterial.Ks * pow(max(0.0, dot(N, H)), uMaterial.Shininess);
}
vec4 DiffuseResult(vec3 N, vec3 L, vec4 textureMapColor) {
    return uMaterial.Kd * max(0.0, dot(N, L)) * textureMapColor;
}
```

4. Implement a ShadedResult() function to compute and accumulate the diffuse and specular terms. Notice that \( lgt.\text{Intensity}, I_l \), in Figure 8-17, and \( lgt.\text{Color}, C_l \), in Figure 8-17, are factored out and multiplied to the sum of diffuse and specular results. The scaling by the light distance attenuation, \( atten \), is the only variation between this implementation and the diffuse/specular terms listed in Figure 8-17.

```cpp
vec4 ShadedResult(Light lgt, vec3 N, vec4 textureMapColor) {
    vec3 L = lgt.Position.xyz - gl_FragCoord.xyz;
    float dist = length(L);
    L = L / dist;
    float atten = LightAttenuation(lgt, dist);
    vec4 diffuse = DiffuseResult(N, L, textureMapColor);
    vec4 specular = SpecularResult(N, L);
    vec4 result = atten * lgt.Intensity * lgt.Color * (diffuse + specular);
    return result;
}
```
5. Complete the implementation in the `main()` function by accounting for the ambient term and looping over all defined light sources to accumulate for `ShadedResults()`. The bulk of the main function is similar to the one in the `IllumFS.gls` file from the previous project; the only important differences are highlighted in bold in the following:

```c
void main(void) {
    // simple tint based on uPixelColor setting
    vec4 textureMapColor = texture2D(uSampler, vTexCoord);
    vec4 normal = texture2D(uNormalSampler, uNormalMapCoord);
    vec4 normalMap = (2.0 * normal) - 1.0;
    // normalMap.y = -normalMap.y; // flip Y
    // depending on the normal map you work with, this may or may not be flipped
    vec3 N = normalize(normalMap.xyz);

    vec4 shadedResult = uMaterial.Ka +
                        (textureMapColor  * uGlobalAmbientColor * uGlobalAmbientIntensity);

    // now decide if we should illuminate by the light
    if (textureMapColor.a > 0.0) {
        for (int i=0; i<4; i++) {
            if (uLights[i].IsOn) {
                shadedResult += ShadedResult(uLights[i], N, textureMapColor);
            }
        }
    }

    vec3 tintResult = vec3(shadedResult) * (1.0-uPixelColor.a) +
                        vec3(uPixelColor) * uPixelColor.a;
    vec4 result = vec4(tintResult, shadedResult.a);
    gl_FragColor = result;
}
```

Creating the Material Object

As described, a simple `Material` object is required to store the per-object material property for the Phong illumination model.

1. Create a new file under the `src/Engine` folder and name it `Material.js`. Remember to load this new source file in `index.html`.

2. Define a constructor to initialize the variables as defined in the surface material property in Figure 8-17. Notice that ambient, diffuse, and specular (Ka, Kd, and Ks) are colors, while shininess is a floating point number.

```javascript
function Material() {
    this.mKa = vec4.fromValues(0.0, 0.0, 0.0, 0);
    this.mKs = vec4.fromValues(0.2, 0.2, 0.2, 1);
    // other variables...
}
```
this.mKd = vec4.fromValues(1.0, 1.0, 1.0, 1);
this.mShininess = 20;
}

3. Provide straightforward get and set accessors to these properties.

Material.prototype.setAmbient = function(a) { this.mKa = vec4.clone(a); };
Material.prototype.getAmbient = function() { return this.mKa; };

Material.prototype.setDiffuse = function(d) { this.mKd = vec4.clone(d); };
Material.prototype.getDiffuse = function() { return this.mKd; };

Material.prototype.setSpecular = function(s) { this.mKs = vec4.clone(s); };
Material.prototype.getSpecular = function() { return this.mKs; };

Material.prototype.setShininess = function(s) { this.mShininess = s; };
Material.prototype.getShininess = function() { return this.mShininess; };

Defining the ShaderMaterial Object

Define a new ShaderMaterial object to communicate the contents of Material to the GLSL IllumFS shader.

1. Create a new file under the src/Engine/Shaders folder and name it ShaderMaterial.js. Remember to load this new source file in index.html.

2. Define a constructor to initialize the variables as references to the ambient, diffuse, specular, and shininess in the IllumFS GLSL shader.

function ShaderMaterial(aIllumShader) {
    // reference to the normal map sampler
    var gl = gEngine.Core.getGL();
    this.mKaRef = gl.getUniformLocation(aIllumShader, "uMaterial.Ka");
    this.mKdRef = gl.getUniformLocation(aIllumShader, "uMaterial.Kd");
    this.mKsRef = gl.getUniformLocation(aIllumShader, "uMaterial.Ks");
    this.mShineRef = gl.getUniformLocation(aIllumShader, "uMaterial.Shininess");
}

3. Define the loadToShader() function to push the content of a Material to the GLSL shader.

ShaderMaterial.prototype.loadToShader = function (aMaterial) {
    var gl = gEngine.Core.getGL();
    gl.uniform4fv(this.mKaRef, aMaterial.getAmbient());
    gl.uniform4fv(this.mKdRef, aMaterial.getDiffuse());
    gl.uniform4fv(this.mKsRef, aMaterial.getSpecular());
    gl.uniform1f(this.mShineRef, aMaterial.getShininess());
};
Modifying the IllumShader Object

Recall that the IllumShader object is the engine’s interface to the corresponding GLSL IllumFS shader. Modify the IllumShader object to define an instance of the ShaderMaterial object to load the contents of the Material object.

1. Edit the IllumShader.js file to define variables for Material and ShaderMaterial. Recall that ShaderMaterial is the Material loader. In addition, define the variables for the camera position and the reference to the camera uniform location in the GLSL shader.

```javascript
function IllumShader(vertexShaderPath, fragmentShaderPath) {
    var gl = gEngine.Core.getGL();

    // Call super class constructor
    LightShader.call(this, vertexShaderPath, fragmentShaderPath);

    // this is the material property of the renderable
    this.mMaterial = null;
    this.mMaterialLoader = new ShaderMaterial(this.mCompiledShader);

    // Reference to the camera position
    this.mCameraPos = null;  // points to a vec3
    this.mCameraPosRef = gl.getUniformLocation(
        this.mCompiledShader, "uCameraPosition");

    //...
}
```

2. Modify the activateShader() function to include the loading of the material and camera position to the shader.

```javascript
IllumShader.prototype.activateShader = function(pixelColor, aCamera) {
    //...
    this.mMaterialLoader.loadToShader(this.mMaterial);
    gl.uniform3fv(this.mCameraPosRef, this.mCameraPos);
};
```

3. Define the setMaterialAndCameraPos() function to set the corresponding variables for Phong illumination computation.

```javascript
IllumShader.prototype.setMaterialAndCameraPos = function(m, p) {
    this.mMaterial = m;
    this.mCameraPos = p;
};
```
Modifying the **IllumRenderable Object**

You need to modify the `IllumRenderable` object to support a material property. This is a straightforward change.

1. Edit the `IllumRenderable.js` file and modify the constructor to instantiate a new `Material` object.
   
   ```javascript
   function IllumRenderable(myTexture, myNormalMap) {
     //...
     // Material for this renderable
     this.mMaterial = new Material();
   }
   ```

2. Modify the `draw()` function to set the material in camera position before the actual rendering. Notice the call to `aCamera.getPosInPixelSpace()`: the camera position obtained is properly transformed into pixel space.

   ```javascript
   IllumRenderable.prototype.draw = function(aCamera) {
     gEngine.Textures.ActivateNormalMap(this.mNormalMap);
     // Here thenormal map texture coordinate is copied from those of
     // the corresponding sprite sheet
     this.mShader.setMaterialAndCameraPos(this.mMaterial,
      aCamera.getPosInPixelSpace());
     LightRenderable.prototype.draw.call(this, aCamera);
   }
   ```

3. Define a simple accessor for the material.

   ```javascript
   IllumRenderable.prototype.getMaterial = function() { return this.mMaterial; }
   ```

**Modifying the Camera Object**

As you have seen in the `IllumFS` GLSL shader implementation, the camera position required for computing the $\hat{V}$ vector must be in pixel space. The Camera object must be modified to provide such information. Since the Camera object stores its position in WC space, this position must be transformed to pixel space for each `IllumRenderable` object rendered. There may be a large number of `IllumRenderable` objects in a scene, and the camera position cannot be changed once rendering begins. These observations suggest that a pixel space camera position should be computed and cached.

1. Edit the `Camera.js` file and add a `vec3` to your `PerRenderCache` function to cache the camera’s position in pixel space.

   ```javascript
   function PerRenderCache() {
     this.mWCToPixelRatio = 1; // WC to pixel transformation
     this.mCameraOrgX = 1;     // Lower-left corner of camera in WC
     this.mCameraOrgY = 1;
     this.mCameraPosInPixelSpace = vec3.fromValues(0, 0, 0);
   }
   ```
2. In the Camera constructor, define a z variable to simulate the distance between the Camera object and the rest of the Renderable objects. This third depth information is required for illumination computation.

   this.kCameraZ = 10; // This is for illumination computation

3. In step B4 of the setupViewProjection() function, call the wcPosToPixel() function to transform the camera's position to 3D pixel space and cache the computed results.

   // Step B4: compute and cache per-rendering information
   this.mRenderCache.mWCToPixelRatio = this.mViewport[Camera.eViewport.eWidth] / this.getWCWidth();
   this.mRenderCache.mCameraOrgX = center[0] - (this.getWCWidth() / 2);
   this.mRenderCache.mCameraOrgY = center[1] - (this.getWCHeight() / 2);
   var p = this.wcPosToPixel(this.getWCCenter());
   this.mRenderCache.mCameraPosInPixelSpace[0] = p[0];
   this.mRenderCache.mCameraPosInPixelSpace[1] = p[1];
   this.mRenderCache.mCameraPosInPixelSpace[2] = this.fakeZInPixelSpace(this.kCameraZ);

4. Define a simple get accessor function for the camera position in pixel space.

   Camera.prototype.getPosInPixelSpace = function () {
     return this.mRenderCache.mCameraPosInPixelSpace;
   };

Testing Specular Reflection

You can now test your implementation of the Phong illumination model and observe the effects of altering object material property and specularity. Since the background, Hero, and left Minion are already instances of the IllumRenderable object, these three objects will exhibit specularity by default. To ensure prominence of specular reflection, the specular material property, Ks, of the background object is set to bright red in the initialize() function.

MyGame.prototype.initialize = function () {
  // ...
  // the Background
  var bgR = new IllumRenderable(this.kBg, this.kBgNormal);
  bgR.setElementPixelPositions(0, 1024, 0, 1024);
  bgR.getXform().setSize(100, 100);
  bgR.getXform().setPosition(50, 35);
  // set background material properties
  bgR.getMaterial().setShininess(100);
  bgR.getMaterial().setSpecular([1, 0, 0, 1]);
  // ...
};
A new function, _selectCharacter(), is defined to allow the user to work with the material property of either the Hero or the Minion object. The file MyGame_MaterialControl.js implements the actual user interaction for controlling the selected material property.

Observations

You can run the project and interactively control the object’s material property. For example, by default the material property of the Hero object is selected. You can try changing the diffuse RGB components by pressing the E/R, T/Y, or U/I keys. Notice that you can press multiple keys simultaneously to change multiple color channels at the same time.

The normal map of the background image is carefully generated and thus is best for examining specularity effects. Press the 1 key to select the first light source, and press the right arrow key to move the light toward the right. You should see the illuminated circle of the light moving toward the right from its initial position that is slightly toward the left of the center of the window. After the light crosses the center of the window and as you continue to move it toward the right, notice there will be a red specular highlight showing up at around the right boundary of the background image. This is the specular reflection of light source number 1 reflecting off the background image reaching the current camera position.

Do experiment with selecting and manipulating the material property of the Minion object and moving the other light sources around to observe the red specular highlight on the background. Notice that because the normal maps of the Hero and Minion objects are generated automatically and do not fully represent the geometric characteristics of these objects, it can be tricky to observe specular reflection off them.

Light Source Types

So far your game engine supports the illumination by many instances of a single type of light, a point light. A point light, behaving much like a lightbulb in the real world, illuminates from a single position with near and far radii where objects can be fully, partial, or not lit at all by the light source. There are two other light types that are popular in most game engines: the directional light and the spotlight.

A directional light models the sun rays where the light appears to arrive in parallel from the same direction, instead of a single position, and does not seem to suffer from distance attenuation. While in reality the sun casts light in all directions, from the perspective of the earth, the light rays from the sun are practically parallel because of the great distance. A directional light is a simple light type that requires only a direction variable and has no distance drop-off. The directional lights are typically used as global lights that illuminate the entire scene.

A spotlight models a desk lamp with a cone-shape lampshade. As illustrated in Figure 8-20, a spotlight is a point light encompassed by a cone pointing in a specific direction, the light direction, with angular attenuation parameters for the inner and outer cone angles. Similar to the near and far radii of the distance attenuation, objects inside the inner cone angle are fully lit, outside the outer cone angle are not lit, and in between the two angles are partially lit. Just as in the case of a point light, a spotlight is often used for creating illumination effects in specific regions of a game scene. The spotlight, with directional and angular attenuation parameters, offers finer controls for simulating effects that are local to specific areas in a game.
In illustrative diagrams, like Figure 8-20, for clarity purposes light directions are usually represented by lines extending from the light position toward the environment. These lines are usually for illustrative purposes and do not carry mathematic meanings. These illustrative diagrams are contrasted with vector diagrams that explain illumination computations, like Figures 8-13 and 8-14. In illumination vector diagrams, all vectors always point away from the position to be illuminated, and all vectors are assumed to be normalized vectors with a magnitude of 1.

The Directional and Spot Lights Project

This project demonstrates how to integrate directional lights and a spotlight into your engine to support a wider range of illumination effects. You can see an example of the project running in Figure 8-21. The source code of this project is located in the Chapter8/8.6.DirectionalAndSpotLights folder.
The controls of the project are as follows:

- **WASD keys**: Move the hero character on the screen
- **Lighting controls**:
  - **Number keys 0, 1, 2, and 3**: Select the corresponding light source
  - **Arrow keys**: Move the currently selected light
  - **Arrow keys with spacebar pressed**: Change the direction of the currently selected light
  - **Z/X key**: Increases/decreases the light z position
  - **C/V and B/N keys**: Increase/decrease the inner and outer cone angles of the selected light
  - **K/L key**: Increases/decreases the intensity of the selected light
  - **H key**: Toggles the selected light on/off


- **Material property controls:**
  - *Number keys 5 and 6:* Select the left minion and the hero
  - *Number keys 7, 8, and 9:* Select the \( K_a, K_d, \) and \( K_s \) material properties of the selected character (left minion or the hero)
  - *E/R, T/Y, and U/I keys:* Increase/decrease the red, green, and blue channels of the selected material property
  - *O/P keys:* Increase/decrease the shininess of the selected material property

The goals of the project are as follows:

- To understand the two additional light types: directional lights and spotlights
- To examine the illumination results from all three different light types
- To experience controlling the parameters of all three light types
- To support the three different light types in the engine and GLSL shaders

In the assets folder you can find the same set of external resource files as in the previous project: the fonts folder that contains the default system fonts, two texture images, two corresponding normal maps for the texture images (minion_sprite.png and bg.png), and the corresponding normal maps (minion_sprite_normal.png and bg_normal.png). As in previous projects, the objects are sprite elements of minion_sprite.png, and the background is represented by bg.png.

---

**Supporting New Light Types in GLSL Fragment Shaders**

As with the previous projects, the integration of the new functionality will begin with the GLSL shader. You must modify the GLSL IllumShader and LightShader fragment shaders to support the two new light types.

**Modifying the GLSL Illumination Fragment Shader**

Recall that the IllumShader simulates the Phong illumination model based on a point light. This will be expanded to support the two new light types.

1. Begin by editing IllumFS.glsl and defining constants for the three light types. Notice that to support proper communications between the WebGL shader and the engine, these constants must have identical values as the corresponding enumerated data defined in the Light.js file.

   ```
   #define ePointLight       0
   #define eDirectionalLight   1
   #define eSpotLight        2
   // ********** WARNING ******
   // The above enumerated values must be identical to
   // Light.eLightType values defined in Light.js
   // ********** WARNING ******
   ```

---

58
2. Expand the light struct within the shader to accommodate the new light types. While the directional light requires only a Direction variable, a spotlight requires a Direction, inner and outer angles, and a DropOff variable. Notice that, as will be detailed next, instead of the actual angle values, the cosine of the inner and outer angles are stored in the struct to facilitate implementation. The DropOff variable controls how rapidly light drops off between the inner and outer angles of the spotlight. The LightType variable identifies the type of light that is being represented in the struct.

```cpp
struct Light {
    vec4 Position; // in pixel space!
    vec4 Direction; // Light direction
    vec4 Color;
    float Near;
    float Far;
    float CosInner; // cosine of inner cone angle for spotlight
    float CosOuter; // cosine of outer cone angle for spotlight
    float Intensity;
    float DropOff; // for spotlight
    bool IsOn;
    int LightType; // One of ePointLight, eDirectionalLight, eSpotLight
};
```

3. Define an AngularDropOff() function to compute the angular attenuation for the spotlight.

```cpp
float AngularDropOff(Light lgt, vec3 lgtDir, vec3 L) {
    float atten = 0.0;
    float cosL = dot(lgtDir, L);
    float num = cosL - lgt.CosOuter;
    if (num > 0.0) {
        if (cosL > lgt.CosInner)
            atten = 1.0;
        else {
            float denom = lgt.CosInner - lgt.CosOuter;
            atten = smoothstep(0.0, 1.0, pow(num/denom, lgt.DropOff));
        }
    }
    return atten;
}
```

The parameter lgt is the reference to the spotlight in light struct, lgtDir is the direction of the spotlight (or Light.Direction), and L is the light vector of the current position to be illuminated. Note that since the dot product of normalized vectors is the cosine of the angle between the vectors, it is convenient to represent all angular displacements by their corresponding cosine values and to carry out the computations based on cosines of the actual angular displacements. Figure 8-22 illustrates the parameters involved in angular attenuation computation.
a. The \( \cos L \) is the dot product of \( L \) with \( \text{lgtDir} \); it records the angular displacement of the position currently being illuminated.

b. The \( \text{num} \) variable stores the difference between \( \cos L \) and \( \cosOuter \). A negative \( \text{num} \) would mean the position currently being illuminated is outside the outer cone, and the angular attenuation would result in no contribution. Thus, no further computation would be required.

c. If the point to be illuminated is within the inner cone, \( \cos L \) would be less than \( \text{lgt.CosInner} \), and the angular attenuation would result in full contribution.

d. If the point to be illuminated is in between the inner and outer cone angles, use the \( \text{smoothstep()} \) function to compute a drop-off.

4. Rename the \( \text{LightAttenuation()} \) function from the previous project to \( \text{DistanceDropOff()} \).

```c
float DistanceDropOff(Light lgt, float dist) {
    float atten = 0.0;
    if (dist <= lgt.Far) {
        if (dist <= lgt.Near)
            atten = 1.0; // no attenuation
        else {
            // simple quadratic drop off
            float n = dist - lgt.Near;
            float d = lgt.Far - lgt.Near;
            atten = smoothstep(0.0, 1.0, 1.0-(n*n)/(d*d)); // blended attenuation
        }
    }
    return atten;
}
```

---

**Figure 8-22. Computing the angular attenuation of a spotlight**
5. Modify the ShadedResults() function to handle each separate case of light source type before combining the results into a color.

```glsl
vec4 ShadedResult(Light lgt, vec3 N, vec4 textureMapColor) {
    float aAtten = 1.0, dAtten = 1.0;
    vec3 lgtDir = -normalize(lgt.Direction.xyz);
    vec3 L; // light vector
    float dist; // distant to light
    if (lgt.LightType == eDirectionalLight) {
        L = lgtDir;
    } else {
        L = lgt.Position.xyz - gl_FragCoord.xyz;
        dist = length(L);
        L = L / dist;
    }
    if (lgt.LightType == eSpotLight) {
        // spotlight: do angle dropoff
        aAtten = AngularDropOff(lgt, lgtDir, L);
    }
    if (lgt.LightType != eDirectionalLight) {
        // both spot and point light has distant dropoff
        dAtten = DistanceDropOff(lgt, dist);
    }
    vec4 diffuse = DiffuseResult(N, L, textureMapColor);
    vec4 specular = SpecularResult(N, L);
    vec4 result = aAtten * dAtten * lgt.Intensity * lgt.Color * (diffuse + specular);
    return result;
}
```

**Modifying the GLSL Light Fragment Shader**

You can now modify the GLSL LightFS fragment shader to support the two new light types. The modifications involved are remarkably similar to the case of IllumFS discussed previously, where constant values that correspond to light types are defined, the Light struct is extended to support directional and spotlights, and the angular and distant attenuation functions are defined to properly attenuate the light. Please refer to the LightFS.glsl source code file for details of the implementation.

**Modifying the Light Object**

You must extend the Light object to support the parameters of the two new light types.

1. Edit the Light.js file and define an enumerated data type for the different light types. It is important that the enumerated values correspond to the constant values defined in the GLSL IllumFS and LightFS shaders.

```javascript
// **** WARNING: The following enumerate values must be identical to
// the values of
// ePointLight, eDirectionalLight, eSpotLight
// defined in LightFS.glsl and IllumFS.glsl
```
Light.eLightType = Object.freeze({
  ePoint: 0,
  eDirectionalLight: 1,
  eSpotLight: 2
});

2. Modify the constructor to define and initialize the new variables that correspond to the parameters of directional light and spotlight.

```
function Light() {
  this.mColor = vec4.fromValues(1, 1, 1, 1);  // light color
  this.mPosition = vec3.fromValues(0, 0, 5);   // light position in WC
  this.mDirection = vec3.fromValues(0, 0, -1); // in WC
  this.mNear = 5;  // effective radius in WC
  this.mFar = 10;
  this.mInner = 0.1;  // in radian
  this.mOuter = 0.3;
  this.mIntensity = 1;
  this.mDropOff = 1;
  this.mLightType = Light.eLightType.ePointLight;
  this.mIsOn = true;
}
```

3. Define the get and set accessors for the new variables. The exhaustive listing of these functions is not shown here. Please refer to the Light.js source code file for details.

### Modifying the ShaderLightAtIndex Object

Recall that the ShaderLightAtIndex object is responsible for loading the values in a light source to the GLSL fragment shader. This object must be refined to support the new light source parameters that correspond to directional lights and spotlights.

1. Edit the ShaderLightAtIndex.js file and modify the setShaderReferences() function to obtain and save the references to the newly added light properties, as shown in the following code:

```
ShaderLightAtIndex.prototype._setShaderReferences = function (aLightShader, index) {
  var gl = gEngine.Core.getGL();
  this.mColorRef = gl.getUniformLocation(aLightShader, "uLights[" + index + "].Color");
  this.mPosRef = gl.getUniformLocation(aLightShader, "uLights[" + index + "].Position");
  this.mDirRef = gl.getUniformLocation(aLightShader, "uLights[" + index + "].Direction");
  this.mNearRef = gl.getUniformLocation(aLightShader, "uLights[" + index + "].Near");
  this.mFarRef = gl.getUniformLocation(aLightShader, "uLights[" + index + "].Far");
  this.mInnerRef = gl.getUniformLocation(aLightShader, "uLights[" + index + "].CosInner");
```

Chapter 8 ■ Implementing Illumination and Shadow

```javascript
this.mOuterRef = gl.getUniformLocation(aLightShader, 
    "uLights[" + index + "]CosOuter");
this.mIntensityRef = gl.getUniformLocation(aLightShader, 
    "uLights[" + index + "]Intensity");
this.mDropOffRef = gl.getUniformLocation(aLightShader, 
    "uLights[" + index + "]DropOff");
this.mIsOnRef = gl.getUniformLocation(aLightShader, "uLights[" + index + "]IsOn");
this.mLightTypeRef = gl.getUniformLocation(aLightShader, 
    "uLights[" + index + "]LightType");
};

2. Modify the loadToShader() function to load the newly added light variables for the directional light and spotlight. Notice that depending upon the light type, the values of some variables may not be transferred to the GLSL shader. For example, the parameters associated with angular attenuation, the inner and outer angles, and the drop-off will be transferred only for spotlights.

ShaderLightAtIndexed.prototype.loadToShader = function (aCamera, aLight) {
    var gl = gEngine.Core.getGL();
    gl.uniform1i(this.mIsOnRef, aLight.isLightOn());
    if (aLight.isLightOn()) {
        var p = aCamera.wcPosToPixel(aLight.getPosition());
        var n = aCamera.wcSizeToPixel(aLight.getNear());
        var f = aCamera.wcSizeToPixel(aLight.getFar());
        var c = aLight.getColor();
        gl.uniform4fv(this.mColorRef, c);
        gl.uniform3fv(this.mPosRef, vec3.fromValues(p[0], p[1], p[2]));
        gl.uniform1f(this.mNearRef, n);
        gl.uniform1f(this.mFarRef, f);
        gl.uniform1f(this.mInnerRef, 0.0);
        gl.uniform1f(this.mOuterRef, 0.0);
        gl.uniform1f(this.mIntensityRef, aLight.getIntensity());
        gl.uniform1f(this.mDropOffRef, 0);
        gl.uniform1i(this.mLightTypeRef, aLight.getLightType());
        if (aLight.getLightType() === Light.eLightType.ePointLight) {
            gl.uniform3fv(this.mDirRef, vec3.fromValues(0, 0, 0));
        } else {
            var d = aCamera.wcDirToPixel(aLight.getDirection());
            gl.uniform3fv(this.mDirRef, vec3.fromValues(d[0], d[1], d[2]));
            if (aLight.getLightType() === Light.eLightType.eSpotLight) {
                gl.uniform1f(this.mInnerRef, Math.cos(0.5 * aLight.getInner()));
                // stores cosine of half of inner angle
                gl.uniform1f(this.mOuterRef, Math.cos(0.5 * aLight.getOuter()));
                // stores cosine of half of outer cone
                gl.uniform1f(this.mDropOffRef, aLight.getDropOff());
            }
        }
    } else {
        // either spot or directional lights: must compute direction
        var d = aCamera.wcDirToPixel(aLight.getDirection());
        gl.uniform3fv(this.mDirRef, vec3.fromValues(d[0], d[1], d[2]));
        if (aLight.getLightType() === Light.eLightType.eSpotLight) {
            gl.uniform1f(this.mInnerRef, Math.cos(0.5 * aLight.getInner()));
            // stores cosine of half of inner angle
            gl.uniform1f(this.mOuterRef, Math.cos(0.5 * aLight.getOuter()));
            // stores cosine of half of outer cone
            gl.uniform1f(this.mDropOffRef, aLight.getDropOff());
        }
    }
};
```

63
Note, for \texttt{mInnerRef} and \texttt{mOuterRef}, the cosine of half the angle is actually computed and passed. Half angles are used because they capture the angular displacements from the light direction. This optimization relieves the GLSL fragment shaders from computing the cosine of these angles for every invocation.

**Modifying the Camera Transform Object**

Directional lights and spotlights require a light direction, and the GLSL \texttt{IllumFS} and \texttt{LightFS} shaders expect this direction to be specified in pixel space. Edit the \texttt{Camera\_Xform.js} file of the Camera object to define the \texttt{wcDirToPixel()} function to transform a direction from WC to pixel space.

```javascript
Camera.prototype.wcDirToPixel = function (d) {  // d is a vec3 direction in WC 
    // Convert the position to pixel space 
    var x = d[0] * this.mRenderCache.mWCToPixelRatio;
    var y = d[1] * this.mRenderCache.mWCToPixelRatio;
    var z = d[2];
    return vec3.fromValues(x, y, z);
};
```

**Testing the New Light Types**

The main goals of the \texttt{MyGame} level are to test and provide functionality for manipulating the new light types. The modifications involved are straightforward; \texttt{MyGame\_Lights.js} is modified to create all three light types, and \texttt{MyGame\_LightControl.js} is modified to support the manipulation of the direction of the selected light when the arrow and space keys are pressed simultaneously. The listing to these simple changes are not shown here; please refer to the source code files for details of the implementation.

**Observations**

You can run the project and interactively control the lights to examine the corresponding effects. There are four light sources defined, each illuminating all objects in the scene. Light source 0 is a point light, 1 is a directional light, and 2 and 3 are spotlights.

You can examine the effect from a directional light by pressing the 1 key to select the light. Now hold the spacebar while taking turns pressing the left/right or up/down keys to swing the direction of the directional light. You will notice drastic illumination changes on the boundary edges of the 3D geometric shapes in the background image, together with occasional prominent specular reflections in red. Now, press the H key to switch off the directional light and observe as the entire scene becomes darker. Without any kinds of attenuation, directional lights can be used as effective tools for brightening the entire scene.

Press the 2 or 3 key to select one of the spotlights. Once again, by holding the spacebar while taking turns pressing the left/right or up/down keys, swing the direction of the spotlight. With the spotlight, you will observe the illuminated region swinging and changing shapes between a circle (when the spotlight is pointing perpendicularly toward the background image) and different elongated ellipses. The arrow keys will move the illuminated region around. Try experimenting with the C/V and B/N keys to increase/decrease the inner and outer cone angles. Notice that if you set the inner cone angle to be larger than the outer one, the boundary of the illuminated region becomes sharp where lighting effects from the spotlight will drop off abruptly.

Try experimenting with the different light settings, including overlapping the light illumination regions and setting the light intensities to negative numbers. While impossible in the physical world, negative intensity lights are completely valid options in a game world.
Shadow Simulation

Shadow is the result of light being obstructed. As an everyday phenomenon, shadow is something you observe but probably do not give much thought to. However, shadow plays a vital role in a human's vision system. For example, the shadows of objects convey important cues of relative sizes, depths, distances, orderings, and so on. In video games, proper simulation of shadows not only can increase the quality of appearance of the games but can significantly increase the fidelity of the games. For example, you can use shadows to properly convey the distance between two game objects or the height that the hero is jumping.

Shadows can be simulated by determining the visibility between the position to be illuminated and each of the light source positions in the environment. Computationally, this is an expensive operation because general visibility determination is an \(O(n)\) operation, where \(n\) is the number of objects in the scene. Algorithmically, this is a challenging problem because the visibility computation needs to occur within the fragment shader during illumination computation. In this section, you will learn about simulating shadows using a dedicated shadow caster and receiver to facilitate the rendering of the shadow based on the WebGL stencil buffer.

Figure 8-23 shows an example where a game wants to cast the shadow of the Hero object on the minion and yet not on the background. In this case, the background object will not participate in the shadow simulation computation and thus will not receive the shadow.

![Figure 8-23. Hero casting shadow on the minion but not on the background](image)

To properly simulate and render the shadow in Figure 8-23, as illustrated in Figure 8-24, there are three important elements.

- **Shadow caster**: This is the object that causes the shadow. In the Figure 8-23 example, the Hero object is the shadow caster.
- **Shadow receiver**: This is the object that the shadow appears on. In the Figure 8-23 example, the Minion object is the shadow receiver.
- **Shadow caster geometry**: This is the actual shadow, in other words, the darkness on the shadow receiver because of the occlusion of light. In the Figure 8-23 example, the dark imprint of the hero appearing on the minion behind the actual hero object is the shadow caster geometry.
Given the three participating elements, the shadow simulation algorithm is rather straightforward: compute the shadow caster geometry, render the shadow receiver as usual, render the shadow caster geometry as a dark shadow caster object over the receiver, and, finally, render the shadow caster as usual. For example, to render the shadow in Figure 8-23, the dark hero shadow caster geometry is first computed based on the positions of the light source, the Hero object (shadow caster), and the Minion object (shadow receiver). After that, the Minion object (shadow receiver) is first rendered as usual, followed by rendering the shadow caster geometry as the Hero object with a dark constant color, and lastly the Hero object (shadow caster) is rendered as usual. As illustrated in Figure 8-25, the challenging problem of this simple simulation occurs when the shadow caster geometry extends beyond the bounds of the shadow receiver. This situation can be observed in Figure 8-23: the top portion of the hero helmet shadow extends beyond the bounds of the minion and is not drawn.

Fortunately, the WebGL stencil buffer is designed specifically to resolve these types of situations. The WebGL stencil buffer can be configured as a buffer of on/off switches with the same pixel resolution as the canvas that is displayed on the web browser. With this configuration, when stencil buffer checking is enabled, the only pixels in the canvas that can be drawn on will be those with corresponding stencil buffer pixels that are switched on. Figure 8-26 uses an example to illustrate this functionality. In this example, the middle layer is the stencil buffer with all pixels initialized to off except for the pixels in the white triangular region being initialized to on. When the stencil buffer checking is enabled, the drawing of the top layer image will result in only the triangular region that corresponds to the stencil triangle appearing in the canvas (bottom layer). In this way, the stencil buffer acts exactly like a stencil over the canvas where only the on-regions can be drawn on.
With the support of the WebGL stencil buffer, shadow simulation can now be specified accordingly by identifying all shadow receivers and by grouping corresponding shadow casters with each receiver. In the Figure 8-23 example, the Hero object is grouped as the shadow caster of the minion shadow receiver. In this example, for the background object to receive a shadow from the hero, it must be explicitly identified as a shadow receiver, and the Hero object must be grouped with it as a shadow caster. Notice that without explicitly grouping the minion object as a shadow caster of the background shadow receiver, the minion will not cast a shadow on the background. As will be detailed in the following implementation discussion, the transparencies of the shadow casters and receivers and the intensity of the casting light source can all affect the generation of shadows. It is important to recognize that this is a fake and virtual simulation. This procedure does not describe how shadows are formed in the real world, and it is entirely possible to create unrealistic dramatic effects such as casting transparent or blue-colored shadows.

The Shadow Simulation Algorithm

Given the WebGL stencil buffer, the shadow simulation and rendering algorithm can now be outlined as follows:

Given a shadowReceiver
A: Draw the shadowReceiver to the canvas as usual

// Stencil operations to enable the region for drawing shadowCaster
B1: Initialize all stencil buffer pixels to off
B2: Switch on the stencil buffer pixels that correspond to the shadowReceiver object
B3: Enable stencil buffer checking

// Compute shadowCaster geometries and draw them on the shadowReceiver
C: For each shadowCaster of this shadowReceiver
   D: For each shadow casting light source
      D1: Compute the shadowCaster geometry
      D2: Draw the shadowCaster geometry

The previous listing renders the shadow receiver and all the shadow caster geometries without rendering the actual shadow caster objects. The B1, B2, and B3 steps switch on the stencil buffer pixels that correspond to the shadow receiver; this is similar to switching on the white triangle in Figure 8-26, enabling the region that can be drawn on. The loops of steps C and D point out that a separate geometry must be computed for each shadow casting light source. By the time step D1 draws the shadow caster geometry, with
the stencil buffer containing the shadow receiver imprint and checking enabled, only pixels occupied by the shadow receiver will be drawn on in the canvas.

The Shadow Shaders Project

This project demonstrates how to implement and integrate the shadow simulation algorithm into your game engine. You can see an example of the project running in Figure 8-27. The source code of this project is located in the Chapter8/8.7.ShadowShaders folder.

![Figure 8-27. Running the Shadow Shaders project](image)

The controls of the project are as follows:

- **WASD keys**: Move both of the hero characters on the screen
- **Lighting controls**:
  - **Number keys 0, 1, 2, and 3**: Select the corresponding light source
  - **Arrow keys**: Move the currently selected light
  - **Arrow keys with spacebar pressed**: Change the direction of the currently selected light
  - **Z/X key**: Increases/decreases the light z position
• **C/V and B/N keys**: Increase/decrease the inner and outer cone angles of the selected light
• **K/L key**: Increases/decreases the intensity of the selected light
• **H key**: Toggles the selected light on/off

**Material property controls:**
• **Number keys 5 and 6**: Select the left minion and the hero
• **Number keys 7, 8, and 9**: Select the $K_a$, $K_d$, and $K_s$ material properties of the selected character (left minion or the hero)
• **E/R, T/Y, and U/I keys**: Increase/decrease the red, green, and blue channels of the selected material property
• **O/P keys**: Increase/decrease the shininess of the selected material property

The goals of the project are as follows:
• Understand shadows can be simulated by rendering explicit geometries
• Appreciate the basic operations of the WebGL stencil buffer
• Understand the simulation of shadows with shadow caster and receiver
• Implement the shadow simulation algorithm based on the WebGL stencil buffer

In the assets folder, you can find the same set of external resource files as in the previous project: the fonts folder that contains the default system fonts, two texture images, two corresponding normal maps for the texture images (minion_sprite.png and bg.png), and the corresponding normal maps (minion_sprite_normal.png and bg_normal.png). As in previous projects, the objects are sprite elements of minion_sprite.png, and the background is represented by bg.png.

**Create GLSL Fragment Shaders**

Two separate GLSL fragment shaders are required to support the rendering of shadow: one for drawing the shadow caster geometry onto the canvas and one for drawing the shadow receiver into the stencil buffer.

**Creating the GLSL Shadow Caster Fragment Shader**

The GLSL ShadowCasterFS fragment shader is the shader for drawing the shadow caster geometries.

1. Under the src/GLSLShaders folder, make a copy of the IllumFS.glsl file and name it ShadowCasterFS.glsl.
2. Keep the light type constants and Light struct definitions (not shown), and define new constants: kMaxShadowOpacity as how opaque shadows should be and kLightStrengthCutOff as a cutoff where a light with intensity less than this value will not cast shadows.

```glsl
#define kMaxShadowOpacity 0.7 // max of shadow opacity
#define kLightStrengthCutOff 0.05 // any less will not cause shadow
```

// ...
3. Leave the AngularDropOff() and DistanceDropOff() functions the same (not shown) and create a LightStrength() function to compute the strength of the given light source. This function is similar to the ShadedResult() function of the IllumFS shader, except that this function computes the light strength arriving at the position to be illuminated instead of a shaded color.

```cpp
float LightStrength() {
    float aAtten = 1.0, dAtten = 1.0;
    vec3 lgtDir = -normalize(uLights[0].Direction.xyz);
    vec3 L; // light vector
    float dist; // distant to light
    if (uLights[0].LightType == eDirectionalLight) {
        L = lgtDir;
    } else {
        L = uLights[0].Position.xyz - gl_FragCoord.xyz;
        dist = length(L);
        L = L / dist;
    }
    if (uLights[0].LightType == eSpotLight) { // spotlight: do angle dropoff
        aAtten = AngularDropOff(lgtDir, L);
    }
    if (uLights[0].LightType != eDirectionalLight) { // both spot and point light has distant dropoff
        dAtten = DistanceDropOff(dist);
    }
    float result = aAtten * dAtten;
    return result;
}
```

4. Compute the shadow in the main() function based on the strength of the light source. Notice that no shadows will be cast if the light intensity is less than kLightStrengthCutOff and that the shadow caster geometry's color is not exactly black or opaque. Instead, it is a blend of the programmer-defined uPixelColor and the sampled transparency from the texture map.

```cpp
void main(void) {
    vec4 texFragColor = texture2D(uSampler, vTexCoord);
    float lgtStrength = LightStrength();
    if (lgtStrength < kLightStrengthCutOff)
        discard;
    vec3 shadowColor = lgtStrength * uPixelColor.rgb;
    shadowColor *= uPixelColor.a * texFragColor.a;
    gl_FragColor = vec4(shadowColor, kMaxShadowOpacity * lgtStrength * texFragColor.a);
    }
```
Creating the GLSL Shadow Receiver Fragment Shader

The GLSL ShadowReceiverFS fragment shader is the shader for drawing the shadow receiver into the stencil buffer. Take note that the stencil buffer is configured as an on/off buffer, and the shader returning any value in \( \text{gl}_\text{FragColor} \) will switch the corresponding pixel to on. For this reason, transparent receiver fragments must be discarded.

1. Under the src/GLSLShaders folder, create a new file and name it ShadowReceiverFS.glsl.

2. Define a \texttt{sampler2D} object such that the shadow receiver object's color texture map can be properly sampled. In addition, define the constant \texttt{kSufficientlyOpaque}. Fragments with opacity of less than this value will be treated as transparent and discarded. Stencil buffer pixels that correspond to discarded fragments will remain off and thus will not be able to receive shadow geometries.

   ```
   // The object that fetches data from texture.
   // Must be set outside the shader.
   uniform sampler2D uSampler;
   
   // The "varying" keyword is for signifying that the texture coordinate will be
   // interpolated and thus varies.
   varying vec2 vTexCoord;
   
   #define kSufficientlyOpaque       0.1
   ```

   Note that to facilitate engine Shader object code reuse, the variable names of \texttt{uSampler} and \texttt{vTexCoord} must not be changed. These correspond to the variables names defined in TextureFS.glsl, and the game engine can use the existing SpriteShader to facilitate the loading of information to this shader.

3. Implement the \texttt{main()} function to sample the texture for the shadow receiver object and test for sufficient opacity for receiving shadow caster geometries.

   ```
   void main(void) {
     vec4 texFragColor = texture2D(uSampler, vTexCoord);
     if (texFragColor.a < kSufficientlyOpaque)
       discard;
     else
       gl_FragColor = vec4(1, 1, 1, 1);
   }
   ```

Interfacing the GLSL Shaders to the Engine

With two new GLSL shaders defined, you may expect that it is necessary to define two corresponding Shader/Renderable object pairs to facilitate the communications. This is not the case for two reasons.

- With the strategic variable naming in the ShadowReceiverFS shader, the existing SpriteShader object can be used to communicate with the ShadowReceiverFS GLSL fragment shader.
Recall that the Renderable objects are designed to support instantiating and manipulating multiple game objects with the corresponding shaders. In this case, the ShadowCasterFS shader is meant for drawing shadow caster geometries, while the ShadowReceiverFS shader is meant for drawing the shadow receiver object into the stencil butter. Notice that neither of the shaders is designed to support objects that are suitable for direct instantiation or manipulations. For these reasons, there is no need for the corresponding Renderable objects.

Creating the Shadow Caster Shader

A JavaScript Shader object must be defined to facilitate the loading of information from the game engine to the GLSL shader. In this case, a ShadowCasterShader needs to be defined to communicate with the GLSL ShadowCasterFS fragment shader.

1. Under the src/Engine/Shaders folder, create a new file and name it ShadowCasterShader.js. Remember to load this new source file in index.html.

2. Define a constructor to inherit ShadowCasterShader from the SpriteShader object. Since each shadow caster geometry is created by one casting light source, define a single light source for the shader.

```javascript
function ShadowCasterShader(vertexShaderPath, fragmentShaderPath) {
  // Call super class constructor
  SpriteShader.call(this, vertexShaderPath, fragmentShaderPath);

  this.mLight = null;  // The light that casts the shadow

  // **** The GLSL Shader must define uLights[1] as the only light source!!
  this.mShaderLight = new ShaderLightAtIndex(this.mCompiledShader, 0);
}
gEngine.Core.inheritPrototype(ShadowCasterShader, SpriteShader);
```

3. Override the activateShader() function to ensure the single light source is loaded to the shader.

```javascript
// Overriding the Activation of the shader for rendering
ShadowCasterShader.prototype.activateShader = function (pixelColor, aCamera) {
  // first call the super class’s activate
  SpriteShader.prototype.activateShader.call(this, pixelColor, aCamera);
  this.mShaderLight.loadToShader(aCamera, this.mLight);
};
```

4. Define a function to set the current light source for this shader.

```javascript
ShadowCasterShader.prototype.setLight = function (l) {
  this.mLight = l;
};
```
Modifying the Engine Core

The core of the game engine and objects defined under the src/Engine/Core folder must be updated in two ways. First, the WebGL stencil buffer must be enabled and maintained. Second, default instances of the engine shaders must be defined to interface to the new GLSL shaders.

Configuring and Maintaining the WebGL Stencil and Depth Buffers

The WebGL stencil buffer must be allocated during WebGL system initialization and cleared when the canvas is cleared. With the well-designed and organized engine system, both of these operations should be defined in the Engine_Core.js file.

1. Edit the Engine_Core.js file. In the _initializeWebGL() function, add the request for the allocation and configuration of stencil and depth buffers during WebGL initialization. Notice that the depth buffer, or z buffer, is also allocated and configured. This is necessary for proper shadow caster support, where a shadow caster must be in front of a receiver, or with a larger z depth in order to cast shadow on the receiver.

```javascript
var _initializeWebGL = function (htmlCanvasID) {
    var canvas = document.getElementById(htmlCanvasID);

    // Get the standard or experimental webgl and binds to the Canvas area
    // store the results to the instance variable mGL
    mGL = canvas.getContext("webgl", {alpha: false, depth: true, stencil: true}) ||
        canvas.getContext("experimental-webgl",
            {alpha: false, depth: true, stencil: true});

    // Allows transparency with textures.
    mGL.blendFunc(mGL.SRC_ALPHA, mGL.ONE_MINUS_SRC_ALPHA);
    mGL.enable(mGL.BLEND);

    // Set images to flip y axis to match the texture coordinate space.
    mGL.pixelStorei(mGL.UNPACK_FLIP_Y_WEBGL, true);

    // make sure depth testing is enabled
    mGL.enable(mGL.DEPTH_TEST);
    mGL.depthFunc(mGL.LEQUAL);

    if (mGL === null) {
        document.write("<br><b>WebGL is not supported!</b>" superclass="null");
        return;
    }
};
```
2. Modify the `clearCanvas()` function. In addition to clearing the canvas, the stencil and depth buffers must also be cleared.

```javascript
var clearCanvas = function (color) {
    mGL clearColor(color[0], color[1], color[2], color[3]); // set the color to be cleared
    mGL clear(mGL.COLOR_BUFFER_BIT | mGL.STENCIL_BUFFER_BIT | mGL.DEPTH_BUFFER_BIT);
    // clear to the color, stencil bit, and depth buffer bits
}
```

---

**Instantiating Default Shadow Caster and Receiver Shaders**

Default instances of engine shaders must be created to connect to the newly defined GLSL shader caster and receiver fragment shaders.

1. Create constants and variables for the shaders in the `Engine_DefaultResources.js` file located in the `src/Engine/Core/Resources` folder.

   ```javascript
   // Shadow shaders
   var kShadowReceiverFS = "src/GLSLShaders/ShadowReceiverFS.glsl";
   // Path to the FragmentShader
   var mShadowReceiverShader = null;
   var kShadowCasterFS = "src/GLSLShaders/ShadowCasterFS.glsl";
   // Path to the FragmentShader
   var mShadowCasterShader = null;
   ```

2. Define engine shaders to interface to the new GLSL fragment shaders. Notice that both of the engine shaders are based on the `TextureVS` GLSL vertex shader. In addition, as discussed, the engine `SpriteShader` is created to interface to the `ShadowReceiverFS` GLSL fragment shader.

   ```javascript
   var _createShaders = function (callBackFunction) {
       gEngine.ResourceMap.setLoadCompleteCallback(null);
       // ...
       mShadowReceiverShader = new SpriteShader(kTextureVS, kShadowReceiverFS);
       mShadowCasterShader = new ShadowCasterShader(kTextureVS, kShadowCasterFS);
       callBackFunction();
   }
   ```

3. The rest of the modifications to the `Engine_DefaultResources.js` file are routine, involving defining accessors, loading and unloading the GLSL source code files, cleaning up the shaders, and exporting the accessors via the public function list. The detailed listings of these are not included here because you saw similar changes on many occasions. Please refer to the source code file for the actual implementations.

---

**Creating the Shadow Caster Object**

As mentioned, creating a `Renderable` object to pair with the `ShadowCasterShader` object would allow game clients to create and manipulate shadow casters as game objects. However, shadow casters and the associated geometries are implicitly computed based on the associated shadow receiver and light sources. For this reason, shadow casters cannot be directly manipulated by the game clients.
Instead of the familiar Renderable object hierarchy, the ShadowCaster object is defined to encapsulate the implicitly defined shadow caster geometry functionality. A ShadowCaster object represents a Renderable game object that will cast shadow on a shadow receiver, another Renderable game object. To support receiving shadows on an animated sprite element, the shadow receiver must be at least a SpriteRenderable object. The shadow casting Renderable object must be able to receive light sources and thus is at least a LightRenderable object. The ShadowCaster object maintains references to the actual shadow casting and receiving Renderable objects and defines the algorithm to compute and render shadow caster geometries for each of the light sources referenced by the caster LightRenderable object. The details of the ShadowCaster object implementation are as follows:

1. Create the new src/Engine/Shadows folder for organizing shadow-related support files.
2. Create a new file in the src/Engine/Shadows/ folder and name it ShadowCaster.js. Remember to load this new source file in index.html.
3. Define the constructor to initialize the instance variables and constants required for caster geometry computations.

```javascript
function ShadowCaster (shadowCaster, shadowReceiver) {
    this.mShadowCaster = shadowCaster;
    this.mShadowReceiver = shadowReceiver;
    this.mCasterShader = gEngine.DefaultResources.getShadowCasterShader();
    this.mShadowColor = [0, 0, 0, 0.2];
    this.mSaveXform = new Transform();

    this.kCasterMaxScale = 3;   // maximum size a caster will be scaled
    this.kVerySmall = 0.001;   //
    this.kDistanceFudge = 0.01;
    // Ensure shadow caster is not at the same depth as receiver
    this.kReceiverDistanceFudge = 0.6;
    // Reduce the projection size increase of the caster
}
```

As discussed, the mShadowCaster is a reference to the shadow caster GameObject, which must reference at least a LightRenderable object, and the mShadowReceiver is a GameObject referencing at least a SpriteRenderable object. As will be detailed in the next step, mCasterShader, mShadowColor, and mSaveXform are variables to support the rendering of shadow caster geometries.

4. Implement the draw() function to compute and draw a shadow caster geometry for each of the light sources that illuminates the Renderable object of mShadowCaster.

```javascript
ShadowCaster.prototype.draw = function(aCamera) {
    var casterRenderable = this.mShadowCaster.getRenderable();
    this.mShadowCaster.getXform().cloneTo(this.mSaveXform);
    var s = casterRenderable.swapShader(this.mCasterShader);
    var c = casterRenderable.getColor();
    casterRenderable.setColor(this.mShadowColor);
    var l, lgt;
    for (l = 0; l < casterRenderable.numLights(); l++) {
        lgt = casterRenderable.getLightAt(l);
        // ...
    }
}
```
if (lgt.isLightOn() && lgt.isLightCastShadow()) {
    this.mSaveXform.cloneTo(this.mShadowCaster.getXform());
    if (this._computeShadowGeometry(lgt)) {
        this.mCasterShader.setLight(lgt);
        SpriteRenderable.prototype.draw.call(casterRenderable, aCamera);
    }
}
this.mSaveXform.cloneTo(this.mShadowCaster.getXform());
casterRenderable.swapShader(s);
casterRenderable.setColor(c);
}

casterRenderable is the Renderable object that is actually casting the shadow. The draw() function first saves the current transform, shader, and color of the casterRenderable object; iterates through all light sources, turning the casterRenderable into the shadow caster geometry; and renders it in three steps.

a. Sets the casterRenderable shader to ShadowCasterShader.

b. Calls the _computeShadowGeometry() function for each illuminating light source to project the casterRenderable onto the shadow receiver.

c. Renders the casterRenderable as a SpriteRenderable. Recall that the ShadowCasterShader will sample the texture map, compute the strength of the current light source to scale the mShadowColor, and turn the pixel into the resulting color.

The casterRenderable state is restored before the draw() function returns.

5. Define the _computeShadowGeometry() function to compute the shadow caster geometry based on the mShadowCaster, the mShadowReceiver, and a casting light source. Although slightly intimidating in length, the following function can be logically separated into four regions. The first region declares and initializes the variables. The second and third regions are the two cases of the if statement that handle the computation of transform parameters for directional and point/spotlights. The last and fourth region sets the computed parameters to the cxf transform.

ShadowCaster.prototype._computeShadowGeometry = function(aLight) {
    // Region 1: variable initialization
    var cxf = this.mShadowCaster.getXform();
    var rxf = this.mShadowReceiver.getXform();
    // vector from light to caster
    var lgtToCaster = vec3.create();
    var lgtToReceiverZ;
    var receiverToCasterZ;
    var distToCaster, distToReceiver; // measured along the lgtToCaster vector
    var scale;
    var offset = vec3.fromValues(0, 0, 0);
    receiverToCasterZ = rxf.getZPos() - cxf.getZPos();
if (aLight.getLightType() === Light.eLightType.eDirectionalLight) {
  // Region 2: parallel projection based on the directional light
  if (Math.abs(aLight.getDirection())[2] < this.kVerySmall ||
      (receiverToCasterZ * (aLight.getDirection())[2] < 0)) {
    return false; // direction light casting side way or
    // caster and receiver on different sides of light in Z
  }
  vec3.copy(lgtToCaster, aLight.getDirection());
  vec3.normalize(lgtToCaster, lgtToCaster);
  distToReceiver = Math.abs(receiverToCasterZ / lgtToCaster[2]);
  // distant measured along lgtToCaster
  scale = Math.abs(1/lgtToCaster[2]);
} else {
  // Region 3: projection from a point (point light or spot light position)
  vec3.sub(lgtToCaster, cxf.get3DPosition(), aLight.getPosition());
  lgtToReceiverZ = rxf.getZPos() - (aLight.getPosition())[2];

  if ((lgtToReceiverZ * lgtToCaster[2]) < 0) {
    return false; // caster and receiver on different sides of light in Z
  }

  if (Math.abs(lgtToReceiverZ) < this.kVerySmall ||
      Math.abs(lgtToCaster[2]) < this.kVerySmall)) {
    // almost the same Z, can't see shadow
    return false;
  }

  distToCaster = vec3.length(lgtToCaster);
  vec3.scale(lgtToCaster, lgtToCaster, 1/distToCaster);
  // normalize lgtToCaster

  distToReceiver = Math.abs(receiverToCasterZ / lgtToCaster[2]);
  // distant measured along lgtToCaster
  scale = (distToCaster + (distToReceiver * this.kReceiverDistanceFudge))
    / distToCaster;
}
// Region 4: sets the cxf transform
vec3.scaleAndAdd(offset, cxf.get3DPosition(), lgtToCaster,
    distToReceiver + this.kDistanceFudge);

cxf.setRotationInRad(cxf.getRotationInRad());
cxf.setPosition(offset[0], offset[1]);
cxf.setZPos(offset[2]);
cxf.setWidth(cxf.getWidth() * scale);
cxf.setHeight(cxf.getHeight() * scale);

return true;
}
Chapter 8 ■ Implementing Illumination and Shadow

The aLight parameter is the casting light source. The goals of this function is to compute and set the shadow caster geometry transform, \( cxf \), by using the aLight to project the shadow caster onto the shadow receiver. As illustrated in Figure 8-28, there are two cases to consider: parallel projection for a directional light source or projection from a point for the point or spotlight.

![Figure 8-28. Computing the shadow caster geometry](image)

- **Region 2**: Computes parallel projection according to the directional light. The if statement is to ensure no shadow is computed when the light direction is parallel to the xy plan or when the light is in the direction from the shadow receiver toward the shadow caster. Notice that for dramatic effects, the shadow caster geometry will be moderately scaled.

- **Region 3**: Computes projection from the point or spotlight position. The two if statements ensure the shadow caster and receiver are on the same side of the light position and that, for the purpose of maintaining mathematic stability, neither is close to the light source.

- **Region 4**: Uses the computed \( \text{distToReceiver} \) and \( \text{scale} \) to set the \( cxf \) transform or the transform of the shadow caster.

Creating the Shadow Receiver Object

The ShadowReceiver object implements the outlined shadow simulation algorithm. The actual implementation of this object is separated into two files. The first file implements the core operations of the object, while the second defines the WebGL-specific stencil operations.

Defining the Shadow Receiver Operations

Follow these steps:

1. Create a new file in the `src/Engine/Shadows/` folder and name it `ShadowReceiver.js`. Remember to load this new source file in `index.html`.

2. Define the constructor to initialize the constants and variables necessary for receiving shadows. As discussed, the `mReceiver` is a GameObject with at least a SpriteRenderable reference and is the actual receiver of the shadow. Notice that `mShadowCaster` is an array of `ShadowCaster` objects. These objects will cast shadows on the `mReceiver`.
function ShadowReceiver (theReceiverObject) {
    this.kShadowStencilBit = 0x01;              // The stencil bit to switch on/off for shadow
    this.kShadowStencilMask = 0xFF;             // The stencil mask
    this.mReceiverShader = gEngine.DefaultResources.getShadowReceiverShader();
    this.mReceiver = theReceiverObject;
    // To support shadow drawing
    this.mShadowCaster = [];                    // array of ShadowCasters
}

3. Define the function addShadowCaster() to add a game object as a shadow caster for this receiver.

ShadowReceiver.prototype.addShadowCaster = function (lgtRenderable) {
    var c = new ShadowCaster(lgtRenderable, this.mReceiver);
    this.mShadowCaster.push(c);
};

4. Define the draw() function to draw the receiver and all the shadow caster geometries.

ShadowReceiver.prototype.draw = function (aCamera) {
    var c;

    // A: draw receiver as a regular renderable
    this.mReceiver.draw(aCamera);

    this._shadowRecieverStencilOn();  // B1
    var s = this.mReceiver.getRenderable().swapShader(this.mReceiverShader);
    this.mReceiver.draw(aCamera);   // B2
    this.mReceiver.getRenderable().swapShader(s);
    this._shadowRecieverStencilOff();  // B3

    // C + D: now draw shadow color to the pixels in the stencil that are switched on
    for (c = 0; c < this.mShadowCaster.length; c++)
        this.mShadowCaster[c].draw(aCamera);

    // switch off stencil checking
    this._shadowRecieverStencilDisable();
};

This function closely implements the outlined shadow simulation algorithm and does not draw the actual shadow caster. Notice that the mReceiver object is drawn twice, in steps A and B2. Step A, the first draw() function, renders the mReceiver to the canvas as usual. Step B1 enables the stencil buffer where all subsequent drawings will be directed to switching on stencil buffer pixels. For this reason, the draw() function at step B2 uses the ShadowReceiverShader and switches on all pixels in the stencil buffer that corresponds to the mReceiver object. With the proper stencil buffer setup, the calls to the mShadowCaster draw() function will draw shadow caster geometries only into the pixels that are covered by the receiver.
Defining the Shadow Receiver Stencil Operations

The stencil buffer configuration actually consists of WebGL-specific operations. These operations are gathered in this file for convenience.

1. Create a new file in the src/Engine/Shadows/ folder and name it ShadowReceiver_Stencil.js. Remember to load this new source file in index.html.

2. Please refer to the source code file for the WebGL operations to configure the stencil buffer to implement the _shadowRecieverStencilOn(), _shadowRecieverStencilOff(), and _shadowRecieverStencilDisable() functions.

Updating Engine Supporting Objects

With the new objects defined and engine configured, some of the existing engine objects must also be modified to support the new shadow operations.

Modifying the Renderable

Both of the ShadowCaster and ShadowReceiver objects require the ability to swap the shaders and render the objects for shadow simulation purpose. This function is best realized in the root of the Renderable hierarchy. Edit the Renderable.js file and define the swapShader() function.

```javascript
Renderable.prototype.swapShader = function (s) {
    var out = this.mShader;
    this.mShader = s;
    return out;
};
```

Modifying the SpriteShader

The engine interfaces to the GLSL ShadowReceiverFS using a SpriteShader, while the engine ShadowReceiver may be a reference to any of the SpriteRenderable, LightRenderable, and IllumRenderable objects. Edit the SpriteShader.js file to define the following two functions to ensure proper drawing for all ShadowReceiver objects:

```javascript
// will be override by LightShader
SpriteShader.prototype.setLights = function (l) { };

// will be override by IllumShader
SpriteShader.prototype.setMaterialAndCameraPos = function (m, p) { };
```

Modifying the Light

The Light object should support the ability to switch shadow casting on or off. Edit the Light.js file to define the instance variable mCastShadow and accessor functions.

```javascript
function Light() {
    // ... code identical to previous
    this.mCastShadow = false;
}
```
Light.prototype.isLightCastShadow = function () { return this.mCastShadow; }
Light.prototype.setLightCastShadowTo = function (on) { this.mCastShadow = on; }

Modifying the Camera

The Camera WC center must now be located at some z distance away. This is easily implemented by editing the Camera.js file and modifying the camera lookAt() matrix computation in the setupViewProjection() function.

mat4.lookAt(this.mViewMatrix,
    [center[0], center[1], this.kCameraZ],  // WC center
    [center[0], center[1], 0],
    [0, 1, 0]);  // orientation

Modifying the Transform Object

The last object that must be modified is the Transform utility. Recall that the Transform object is defined to implement the transformation operations in 2D. This object must now be updated to support some 3D positioning.

1. Edit the Transform.js file and add a z component.

    function Transform() {
        this.mPosition = vec2.fromValues(0, 0); // this is the translation
        this.mScale = vec2.fromValues(1, 1);    // this is the width (x) and height (y)
        this.mZ = 0.0;                          // must be a positive number,
                                                // larger is closer to eye
        this.mRotationInRad = 0.0;              // in radians!
    }

2. Define assessors for the z position.

    Transform.prototype.get3DPosition = function () {
        return vec3.fromValues(this.getXPos(), this.getYPos(), this.getZPos());
    }
    Transform.prototype.setZPos = function (d) { this.mZ = d; }
    Transform.prototype.getZPos = function () { return this.mZ; }
    Transform.prototype.incZPosBy = function (delta) { this.mZ += delta; }

3. Define the cloneTo() function to duplicate the transform.

    Transform.prototype.cloneTo = function (aXform) {
        aXform.mPosition = vec2.clone(this.mPosition);
        aXform.mScale = vec2.clone(this.mScale);
        aXform.mZ = this.mZ;
        aXform.mRotationInRad = this.mRotationInRad;
    }
4. Utilize the z component when computing an object transform.

```javascript
Transform.prototype.getXform = function () {
// Creates a blank identity matrix
var matrix = mat4.create();

// The matrices that WebGL uses are transposed, thus the typical matrix
// operations must be in reverse.

// Step A: compute translation, for now z is the mHeight
mat4.translate(matrix, matrix, this.get3DPosition());
// Step B: concatenate with rotation.
mat4.rotateZ(matrix, matrix, this.getRotationInRad());
// Step C: concatenate with scaling
mat4.scale(matrix, matrix, vec3.fromValues(this.getWidth(), this.getHeight(), 1.0));

return matrix;
};
```

Testing the Shadow Algorithm

There are two important aspects to testing the shadow simulation. First, you must understand how to
program and create shadow effects based on the implementation. Second, you must verify that Renderable
objects can serve as shadow casters and receivers. The MyGame level test case is similar to the previous project
with the exception of the shadow setup and drawing.

Setting Up the Shadow

The proper way of setting up the shadow system is to create ShadowReceiver objects and then add
ShadowCaster objects to it. The MyGame_Shadow.js file defines the _setupShadow() function to demonstrate
this. The _setupShadow() function is called at the end of the MyGame.initialize() function, when all
GameObject instances are properly created and initialized. The details of the MyGame _setupShadow() function are as follows:

```javascript
MyGame.prototype._setupShadow = function () {
    this.mBgShadow = new ShadowReceiver(this.mBg);
    this.mBgShadow.addShadowCaster(this.mLgtHero);
    this.mBgShadow.addShadowCaster(this.mIllumMinion);
    this.mMinionShadow = new ShadowReceiver(this.mIllumMinion);
    this.mMinionShadow.addShadowCaster(this.mIllumHero);
    this.mMinionShadow.addShadowCaster(this.mLgtHero);
    this.mLgtMinionShaodw = new ShadowReceiver(this.mLgtMinion);
    this.mLgtMinionShaodw.addShadowCaster(this.mIllumHero);
    this.mLgtMinionShaodw.addShadowCaster(this.mLgtHero);
};
```
This function demonstrates that three types of Renderable objects can serve as shadow receivers.

- **IllumRenderable**: mBgShadow has mBg as a receiver, which has a reference to an IllumRenderable object.
- **SpriteAnimateRenderable**: mMinionShadow has mIllumMinion as a receiver, which has a reference to a SpriteAnimateRenderable object.
- **LightRenderable**: mLgtMinionShadow has mLgtMinon as a receiver, which has a reference to a LightRenderable object.

The shadow casters for these receivers show that IllumRenderable, SpriteAnimateRenderable, and LightRenderable can all serve as shadow casters.

### Drawing the Shadow

In 2D drawings, objects are drawn and overwrite the previously drawn objects. For this reason, it is important to draw the shadow receivers and the shadow caster geometries before drawing the shadow casters. The following `drawCamera()` function is defined in the `MyGame.js` file:

```javascript
MyGame.prototype.drawCamera = function (camera) {
    // set up the View Projection matrix
    camera.setupViewProjection();

    // always draw shadow receivers first!
    this.mBgShadow.draw(camera);        // also draws the receiver object
    this.mMinionShadow.draw(camera);
    this.mLgtMinionShaodw.draw(camera);

    this.mBlock1.draw(camera);
    this.mIllumHero.draw(camera);
    this.mBlock2.draw(camera);
    this.mLgtHero.draw(camera);
};
```

The rest of the `MyGame` level is largely similar to previous projects and is not listed here. Please refer to the source code for the details.

### Observations

You can now run the project and observe the shadows. Notice the effect of the stencil buffer where the shadow from the `mIllumHero` object is cast on the minion and yet not on the background. Press the WASD keys to move both of the Hero objects. Observe how the shadows offer depth and distance cues as they move with the Hero objects. The `mLgtHero` on the right is illuminated by all four lights and thus casts many shadows. Light 1 does not illuminate the background, and thus the `mLgtHero` shadow from light 1 is not visible on the background but visible on the minions. Try selecting and manipulating each of the lights, such as moving or changing the direction or switching the light on/off to observe the effects on the shadows. You can even try changing the color of the shadow (in `ShadowCaster.js`) to something dramatic, such as to bright blue [0, 0, 5, 1], and observe shadows that could never exist in the real world.
Summary

This chapter guided you in developing a variation of the simple yet complete Phong illumination model for your game engine. The examples were organized to follow the three terms of the Phong illumination model: ambient, diffuse, and specular. The light source examples were strategically intermixed because without the lights illumination cannot occur.

The first example in this chapter on ambient illumination introduced the idea of interactively controlling and fine-tuning the color of the scene. The following two examples on light sources presented the notion that illumination, an algorithmic approach to color manipulation, can be localized and developed in the engine infrastructure for supporting the eventual Phong illumination model. The example on diffuse reflection and normal mapping was a critical one because it enabled illumination computation based on simple physical models and simulation of an environment in 3D. The Phong illumination model and the need for a per-object material property were presented in the specular reflection example. The halfway vector variant of the Phong illumination model was implemented to avoid computing the light source reflection vector for each pixel. The light source types example demonstrated how subtle but important illumination variations can be accomplished by simulating different light sources in the real world. Finally, the last example explained that accurate shadow computation is nontrivial and introduced an approximation algorithm. The resulting shadow simulation, though inaccurate from a real-world perspective and with limitations, can be aesthetically appealing and is able to convey many of the vital visual cues.

The first four chapters of this book introduced the basic foundations and components of a game engine. Chapters 5, 6, and 7 extended the core engine functionality to support drawing, game object behaviors, and camera controls, respectively. This chapter complements Chapter 5 by bringing the engine’s capability in rendering higher-fidelity scenes to a new level. Over the next two chapters, this complementary pattern will be repeated. Chapter 9 will introduce physical behavior simulation, and Chapter 10 will complete the engine development with more advanced support for the camera including tiling and parallax.

Game Design Considerations

The work you did in the “Game Design Consideration” section of Chapter 7 to create a basic well-formed game mechanic will ultimately need to be paired with the other elements of game design to create something that feels satisfying for players. In addition to the basic mechanic or mechanics, you’ll need to think about your game’s systems, setting, and metagame and how they’ll help determine the kinds of levels you design, and you’ll want to begin exploring ideas for visual and audio design as you begin to define the setting.

As with most visual work, games rely in no small measure on lighting to convey setting. A horror game taking place in a graveyard at midnight will typically use a very different lighting model and color palette than a game focusing on upbeat, happy themes. Many people think that lighting applies primarily to games created in 3D engines that are capable of simulating realistic light and shadow, but the notion of lighting applies to most 2D game environments as well; consider the example presented by Playdead studio’s 2D side-scrolling platform game Limbo, as shown in Figure 8-29.
Lighting is also often used to help drive a game mechanic in addition to setting the mood; if you’ve ever played a game where you were navigating a game space in the dark with a virtual flashlight, that’s one direct example, but lights can also indirectly support game mechanics by providing important information about the game environment. Red pulsing lights often signal dangerous areas, certain kinds of green environment lights might signal areas with deadly gas, flashing lights on a map can help direct players to important areas, and the like.

In the Simple Global Ambient project, you saw the impact that colored environment lighting has on the game setting. In this project, the hero character moves in front of a background that appears to be made of metallic panels, tubes, and machinery; perhaps it’s the exterior of a space ship. The environment light is red, and it can be pulsed. Notice the effect on mood when the intensity is set at a comparatively low 1.5 versus when it’s set to something like a super-saturated 3.5, and imagine how the pulsing between the two values might convey a story or increase tension. In the Simple Light Shader: One Light Source project, a light was attached to the hero character (a point light in this case), and you can imagine that the hero must navigate the environment to collect objects to complete the level that are visible only when illuminated by the light (or perhaps activate objects that switch on only when illuminated).

The Diffuse Shader with Multiple Light Sources project illustrated how various light sources and colors can add considerable visual interest to an environment, sometimes referred to as localized environment lighting. Varying the types, intensities, and color values of lights often makes environments (especially representational environments of physical spaces) appear more alive and engaging because the light you encounter in the real world typically originates from many different sources. The other projects in this chapter all served to similarly enhance the sense of presence in the game level; as you work with diffuse shaders, normal maps, specularity, different light types, and shadows, consider how you might integrate some or all of these techniques into a level’s visual design to make game objects and environments feel more vibrant and interesting.

Figure 8-29. Playdead and Double Eleven’s Limbo, a 2D side-scrolling game making clever use of background lighting and chiaroscuro techniques to convey tension and horror. Lighting can be both programmatic and designed into the color palettes of the images themselves by the visual artist and is frequently a combination of the two (image copyright Playdead media; please see http://www.playdead.com/limbo for more information)
Before you begin thinking about how lighting and other design elements might enhance the game setting and visual style, let’s return for a moment to the simple game mechanic project from the “Game Design Consideration” section of Chapter 7 and consider how you might think about adding lighting to the mechanic to make the puzzle more engaging; Figure 8-30 begins with the basic mechanic from the end of the exercise.

**Figure 8-30.** The simple game mechanic project, without lighting. Recall that the player controls the circle labeled with a P and must activate each of the three sections of the lock in proper sequence to disengage the barrier and reach the reward.

Lighting can be used exclusively to support a game’s visual style with no direct impact on gameplay, or as mentioned earlier, it can be integrated more or less directly with the game mechanic; depending on the direction your mechanic follows, either you’ll want to begin experimenting with lighting right away (if it’s integrated with the game mechanic) or you’ll first focus on defining the game setting (if it’s purely part of the visual style) because the setting will typically determine the colors, shapes, and lighting you choose.

For the next phase of the simple game mechanic project, how might you integrate light directly into the mechanic so that it becomes part of gameplay? As with the previous exercise, minimizing complexity and limiting yourself to one addition or evolution to the current mechanic at a time will help prevent the design from becoming overburdened or too complex. Start this phase of the exercise by considering all the different ways that light might impact the current game screen. You might choose to have a dark environment where the player sees only shadowy shapes unless illuminating an area with a flashlight, you might use colored light to change the visible color of illuminated objects, or you might use something like an X-ray or ultraviolet beam to reveal information about the objects that wouldn’t be seen with the naked eye. For this example, you’ll add one additional dimension to the simple sequence mechanic: a light beam that reveals hidden information about the objects, as shown in Figure 8-31.
In the first iteration of this mechanic, the design required players to activate each segment of the lock in both the correct relative position (top on top, middle in the middle, bottom on bottom) and the correct order (top-middle-bottom). The interaction design provided consistent visual feedback for both correct and incorrect moves that allowed the player to understand the rules of play, and with some experimentation, astute players could deduce the proper sequence required to unlock the barrier. Now imagine how the addition of a special light beam might take the mechanic in a new direction. Building upon the basic notion of sequencing, you can create an incrementally cleverer puzzle requiring players to first discover the flashlight in the environment and experiment with it as a tool before making any progress on the lock. Imagine perhaps that the player can still directly activate the shapes when the hero character touches them even without the flashlight (triggering the highlight ring around the object as was the case in the first iteration, as shown in Figure 8-32) but that direct interaction is insufficient to activate the corresponding area of the lock unless the flashlight first reveals the secret clues required to understand the puzzle. Figure 8-33 shows the flashlight moved to illuminate one of the objects with its beam, revealing a single white dot.
From a gameplay point of view, any object in a game environment can be conscripted into service as a tool; your job as a mechanic designer is to ensure the use of the tool follows consistent, logical rules the player can first understand and then predictively apply to achieve their goal. In this case, it’s reasonable to assume that players will explore the game environment looking for tools or clues; if the flashlight is an active object, players will attempt to learn how it functions in the context of the level.
The mechanic is evolving with the flashlight but uses the same basic sequencing principles and feedback metaphors. When the player reveals the secret symbol on the object with the flashlight, the player can begin the unlocking sequence by activating the object when the symbol is visible. The new design requires players to activate each of the three objects corresponding to each section of the lock in the correct order, in this case from one dot to three dots; when all objects in a section are activated in order, that section of the lock will light up just as it did in the first iteration. Figures 8-34 to 8-36 show the new sequence using the flashlight beam.

**Figure 8-34.** With the flashlight revealing the hidden symbol, the player can now activate the object (#2), and a progress bar (#3) on the lock indicates the player is on the right track to complete a sequence.

**Figure 8-35.** The player activates the second of the three top sections in the correct order (#4), and the progress bar confirms the correct sequence by lighting another section (#5). In this implementation, the player would not be able to activate the object with two dots before activating the object with one dot (the rules require activating like objects in order from one to three dots).
Note that you’ve changed the feedback players receive slightly from the first iteration of the mechanic. You originally used the progress bar to signal overall progress toward unlocking the barrier, but you’re now using it to signal overall progress toward unlocking each section of the lock. The flashlight introduces an extra step into the causal chain leading to the level solution, and you’ve now taken a one-step elemental game mechanic and made something considerably more complex and challenging while maintaining logical consistency and following a set of rules that players can first learn and then predictively apply. In fact, the level is beginning to typify the kind of puzzle found in many adventure games. If the game screen was a complex environment filled with a number of movable objects, finding the flashlight and learning that its beam reveals hidden information about objects in the game world would become part of the game setting itself.

Although somewhat tangential to the focus of this chapter, it’s important to be aware that as gameplay complexity increases, so increases the complexity of the interaction model and the importance of providing players with proper audiovisual feedback to help them make sense of their actions (recall from Chapter 1 that the interaction model is the combination of keys, buttons, controller sticks, touch gestures, and the like that the player uses to accomplish game tasks). In the current example, the player is now capable of controlling not just the hero character but also the flashlight. Creating intuitive interaction models is a critical component of game design and often much more complex than designers realize; as one example, consider the difficulty in porting many PC games designed for a mouse and keyboard to a game console using buttons and thumb sticks. Development teams often pour thousands of hours of research and testing into control schemes, yet they still frequently miss the mark. There are many books dedicated to examining interaction design in detail, but for the purposes of this book, you should keep the two golden rules in mind when you design interactions. First, use known and tested patterns when possible unless you have a compelling reason to ask players to learn something new; second, keep the number of unique actions players must remember to a minimum. Decades of user testing have clearly shown that players don’t enjoy relearning basic key combinations for tasks that are similar across titles (which is why so many games have standardized on WASD for movement, for example), and similar data is available showing how easily players can become overwhelmed when you ask them to remember more than a few simple unique button

**Figure 8-36.** The third of the three top sections is revealed with the flashlight beam and activated by the player (#6), thereby activating the top section of the lock (#7). Once the middle and lower sections of the lock have been similarly activated, the barrier is disabled and players can claim the reward.
combinations. There are exceptions, of course; many classic arcade fighting games, for example, use dozens of complex combinations, but those genres are targeted to a specific kind of player who considers mastering button combinations to be a fundamental component of the game mechanic. As a general rule, most players prefer to keep interaction complexity as streamlined and simple as possible if it’s not an intentional component of play.

There are a number of ways to deal with moving two objects in the current example. It would be reasonable to assign one kind of interaction for the hero character (the circle labeled P) and another for all other game objects (in this case, the flashlight). Perhaps the hero character can move around the game screen freely, while other objects are first selected with a left mouse click and can be moved by holding the left mouse button and dragging them into position. There are similarly a number of ways to provide the player with contextual feedback that will help teach the puzzle logic and rules (in this case, using a progress bar to confirm players are following the correct sequence). As you experiment with various interaction and feedback models, it’s always a good idea to review how other games have handled similar tasks, paying particular attention to things you believe to work especially well.

In the next chapter, you’ll investigate how your mechanic can evolve once again by applying simple physics to objects in the game world.