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 until now in the game engine you have implemented mostly functional modules in order to provide the fundamentals required for many types of 2D games. That is, you have developed engine components and utility classes that are designed to support the actual gameplay directly. 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 approximating 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 support from a game engine can be more than a simple aesthetic effect. When applied 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 in the physical 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 source that absorbs visible energy 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. As such, the third dimension, or depth, must be specified for the light sources to cast light energy upon the game objects, or the Renderable objects, 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 simplified variation of the Phong illumination model, caters specifically to the 2D aspect of your game engine, will be derived and implemented. 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 recommended reference books from 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 in the following order.

* 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. In order to properly render and display the results of illumination, the associated computation must be performed for each affected pixel. Recall that the GLSL fragment shader is responsible for computing the color of each pixel. In this way, each fundamental element of the Phong illumination model can be implemented as additional functionality to existing or new GLSL fragment shaders. In all projects of this chapter, you will begin by working with the GLSL fragment shader.

# 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 algorithmically complex and can be computationally expensive. Instead, in computer graphics and most 2D games, ambient lighting is approximated by adding a constant color, or the ambient light, to every object within the current scene or world. It is important to note that while ambient lighting can provide the desired results, it is only a rough approximation and does not mimic real-world indirect lighting.

## The Global Ambient Project

This project demonstrates how to implement ambient lighting within your scenes by defining a global ambient color and a global ambient intensity for drawing each Renderable object. 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.global\_ambient folder.



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 or shading functionality for the game engine is the GLSL shader. The GLSL code creation or modification allows you to implement the actual functional details, which in turn, serves as the requirements for expanding the engine.

For example, in this project you will begin by adding ambient lighting functionality to all existing GLSL shaders. The support for this newly added functionality then becomes the requirements that guide the modifications to the rest of the engine. You will observe this implementation pattern for all of the examples in this chapter.

Thus, to begin, integrate the global ambient into your simple\_fs.glsl.

1. Modify the fragment shader simple\_fs.glsl by defining two new uniform variables uGlobalAmbientColor and uGlobalAmbientIntensity, and multiplying these variables with the uPixelColor when computing the final color for each pixel.

precision mediump float;

// Color of pixel

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;

}

1. Similarly modify the texture fragment shader texture\_fs.glsl by adding the uniform variables uGlobalAmbientColor and uGlobalAmbientIntensity. Multiply these two variables with the sampled texture color to create the background lighting effect.

uniform sampler2D uSampler;

// Color of pixel

uniform vec4 uPixelColor;

uniform vec4 uGlobalAmbientColor; // this is shared globally

uniform float uGlobalAmbientIntensity; // this is shared globally

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;

… identical to previous code …

}

### Defining as Global Shared Resources

Ambient lighting affects the entire scene and thus the associated variables must be global and shared. In this case, the two variables, a color (ambient color), and a floating-point (intensity of the color), should be globally accessible to the rest of the engine and to the clients. The defaultResources module is perfectly suited for this purpose. Edit the src/engine/resources/default\_resources.js file and define the color and intensity variables, their corresponding getters and setters, and remember to export the functionality.

import \* as font from "./font.js";

import \* as map from "../core/resource\_map.js";

// Global Ambient color

let mGlobalAmbientColor = [0.3, 0.3, 0.3, 1];

let mGlobalAmbientIntensity = 1;

function getGlobalAmbientIntensity() { return mGlobalAmbientIntensity; }

function setGlobalAmbientIntensity(v) { mGlobalAmbientIntensity = v; }

function getGlobalAmbientColor() { return mGlobalAmbientColor; }

function setGlobalAmbientColor(v) { mGlobalAmbientColor = vec4.fromValues(v[0], v[1], v[2], v[3]); }

… identical to previous code …

export {

init, cleanUp,

// default system font name: this is guaranteed to be loaded

getDefaultFontName,

// Global ambient: intensity and color

getGlobalAmbientColor, setGlobalAmbientColor,

getGlobalAmbientIntensity, setGlobalAmbientIntensity

}

### Modifying SimpleShader

With global ambient color and intensity now implemented in the GLSL shaders, you need to modify the rest of the game engine to support the newly defined functionality. Recall that simple\_fs.glsl is referenced by the SimpleShader class, and that texture\_fs.glsl is referenced by the TextureShader class. Since TextureShader is a subclass of SimpleShader, the newly defined GLSL functionality in texture\_fs.glsl will be supported with appropriate SimpleShader, super class of TextureShader, modifications.

1. Modify the simple\_shader.js file in the src/engine/shaders folder to import from the defaultResources module for accessing the global ambient light effects variables.

import \* as defaultResources from "../resources/default\_resources.js";

1. Define two new instance variables in the constructor for storing the references or locations of the ambient color and intensity variables in the GLSL shader.

this.mGlobalAmbientColorRef = null;

this.mGlobalAmbientIntensityRef = null;

1. In step E of the SimpleShader constructor, call the WebGL getUniformLocation() function to query and store the locations of the uniform variables for ambient color and intensity in the GLSL shader.

// Step E: Gets references to the uniform variables

this.mPixelColorRef = gl.getUniformLocation(this.mCompiledShader, "uPixelColor");

this.mModelMatrixRef = gl.getUniformLocation(this.mCompiledShader, "uModelXformMatrix");

this.mCameraMatrixRef = gl.getUniformLocation(this.mCompiledShader, "uCameraXformMatrix");

this.mGlobalAmbientColorRef = gl.getUniformLocation(this.mCompiledShader, "uGlobalAmbientColor");

this.mGlobalAmbientIntensityRef = gl.getUniformLocation(this.mCompiledShader, "uGlobalAmbientIntensity");

1. In the activate() function, retrieve the global ambient color and intensity values from the defaultResources module and pass to the corresponding uniform variables in the GLSL shader. Notice the data type specific WebGL function names for setting uniform variables. As you can probably guess, uniform4fv corresponds to vec4, which is the color storage, and to uniform1f, corresponds to a float and is the intensity.

activate(pixelColor, trsMatrix, cameraMatrix) {

let gl = glSys.get();

… identical to previous code …

// load uniforms

gl.uniformMatrix4fv(this.mCameraMatrixRef, false, cameraMatrix);

gl.uniform4fv(this.mGlobalAmbientColorRef, defaultResources.getGlobalAmbientColor());

gl.uniform1f(this.mGlobalAmbientIntensityRef, defaultResources.getGlobalAmbientIntensity());

}

### Testing the Ambient Illumination

You can now define the MyGame class to verify the correctness of the newly defined ambient lighting effect. In anticipation of upcoming complexities in testing, the MyGame class source code will be separated into multiple files similar to your experience working with the Camera class in Chapter 7. All files implementing MyGame will begin with my\_game where each individual filename will end with indication of the associated functionality defined in the file. For example, in later examples, my\_game\_light.js indicates the file implements light source related logic. For this project, similar to the Camera class naming scheme, the basic functionality of MyGame class will be implemented in my\_game\_main.js, and the access will be via the file my\_game.js.

1. Create the MyGame class access file in src/my\_game. For now, the MyGame functionality should be imported from the basic class implementation file, my\_game\_main.js. With full access to the MyGame class, it is convenient to define the webpage onload() function in this file.

import engine from "../engine/index.js";

import MyGame from "./my\_game\_main.js";

window.onload = function () {

engine.init("GLCanvas");

let myGame = new MyGame();

myGame.start();

}

1. Create my\_game\_main.js, import from the engine access file, index.js, and from Hero and Minion, and remember to export the MyGame functionality. Now, as in all previous cases, define the MyGame class as a subclass of engine.Scene. Define the constructor by to null-initialize instance variables.

import engine from "../engine/index.js";

// user stuff

import Hero from "./objects/hero.js";

import Minion from "./objects/minion.js";

class MyGame extends engine.Scene {

constructor() {

super();

this.kMinionSprite = "assets/minion\_sprite.png";

this.kBg = "assets/bg.png";

// The camera to view the scene

this.mCamera = null;

this.mBg = null;

this.mMsg = null;

// the hero and the support objects

this.mHero = null;

this.mLMinion = null;

this.mRMinion = null;

}

... implementation to follow …

}

export default MyGame;

1. Practice proper implementation by remembering to load and unload the background and the minions.

load() {

engine.texture.load(this.kMinionSprite);

engine.texture.load(this.kBg);

}

unload() {

engine.texture.unload(this.kMinionSprite);

engine.texture.unload(this.kBg);

}

1. Initialize the camera and scene objects with corresponding values to ensure proper scene view at startup. Note the simple elements in the scene, the camera, the large background, a Hero, the left and right Minion objects, and the status message.

init() {

// Step A: set up the cameras

this.mCamera = new engine.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

let bgR = new engine.SpriteRenderable(this.kBg);

bgR.setElementPixelPositions(0, 1900, 0, 1000);

bgR.getXform().setSize(190, 100);

bgR.getXform().setPosition(50, 35);

this.mBg = new engine.GameObject(bgR);

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 engine.FontRenderable("Status Message");

this.mMsg.setColor([1, 1, 1, 1]);

this.mMsg.getXform().setPosition(1, 2);

this.mMsg.setTextHeight(3);

}

1. Define the draw() function. As always, draw the status message last such that it will not be covered by any other object.

draw() {

// Clear the canvas

engine.clearCanvas([0.9, 0.9, 0.9, 1.0]); // clear to light gray

// Set up the camera and draw

this.mCamera.setViewAndCameraMatrix();

this.mBg.draw(this.mCamera);

this.mHero.draw(this.mCamera);

this.mLMinion.draw(this.mCamera);

this.mRMinion.draw(this.mCamera);

this.mMsg.draw(this.mCamera); // draw last

}

1. Lastly, implement the update() function to update all objects and receive controls over global ambient color and intensity.

update() {

let deltaAmbient = 0.01;

let 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);

let v = engine.defaultResources.getGlobalAmbientColor();

if (engine.input.isButtonPressed(engine.input.eMouseButton.eLeft))

v[0] += deltaAmbient;

if (engine.input.isButtonPressed(engine.input.eMouseButton.eMiddle))

v[0] -= deltaAmbient;

if (engine.input.isKeyPressed(engine.input.keys.Left))

engine.defaultResources.setGlobalAmbientIntensity(

engine.defaultResources.getGlobalAmbientIntensity() - deltaAmbient);

if (engine.input.isKeyPressed(engine.input.keys.Right))

engine.defaultResources.setGlobalAmbientIntensity(

engine.defaultResources.getGlobalAmbientIntensity() + deltaAmbient);

msg += " Red=" + v[0].toPrecision(3) + " Intensity=" +

engine.defaultResources.getGlobalAmbientIntensity().toPrecision(3);

this.mMsg.setText(msg);

}

### Observations

You can now run the project and observe the results. Notice that the initial scene is dark. This is because the RGB values for the global ambient color were all initialized to 0.3. 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 can be accomplished if the RGB values were set to 1.0 and the intensity was set 0.3 because the two sets of values are simply multiplied.

Before moving onto the next project, try fiddling with the ambient red channel and the ambient intensity to observe their effects on the scene. By pressing the right-arrow key, you can increase the intensity of the entire scene and make all objects more visible. Continue with this increment and observe that when the intensity reaches values beyond 15.0, all colors in the scene converges towards white, or begin too over saturate. Without proper context, over saturation can be a distraction. However, it is also true that strategically creating over saturation on selective objects can be used to indicate significant events, e.g., triggering a trap. The next section describes how to create and direct a light source to illuminate only on selected objects.

# Light Source

Examine your surroundings and you can observe many types of light sources, e.g., your table lamp, light rays from the Sun, or, an isolated light bulb. The isolated light bulb can be described as a point that emits light uniformly in all directions, or a point light. The point light is where you will begin to analyze light sources.

Fundamentally, a point light illuminates an area or radius around a specified point. In three-dimensional space, this region of illumination is simply a sphere, referred to as a volume of illumination. The volume of illumination of a point light is defined by the position of the light, or the center of the sphere, and the distance that the light illuminates, or the radius of the sphere. To observe the effects of a light source, objects must be present and within the volume of illumination.

As mentioned in the introduction of this chapter, the 2D engine will need to venture into the third dimension to properly simulate the propagation of light energy. Now, consider your 2D engine; thus far you have implemented a system in which everything is in 2D. An alternative way is to interpret that the engine defines and renders everything on a single plane where z = 0 and objects are layered by drawing order. On this system, you are 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 are defined. Figure 8-2 shows the volume of illumination from a simple point light located at z = 10 intersecting a plane at z = 0. This intersection results in an illuminated circle on the plane. The next project implements Figure 8-2 where you will examine light sources with an object-oriented approach while adhering to the expectations of how a light illuminates a scene. This can be achieved through the definition of a Light object to represent a light source.



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

## GLSL Implementation and Integration into the Game Engine

Recall that the engine interfaces to the GLSL shaders with the corresponding subclasses of the Shader/Renderable object pairs. Shader objects interface to the GLSL shaders and Renderable objects provide programmers with the convenience of manipulating many copies of geometries with the same shader type. For example, texture\_vs.glsl and texture\_fs.glsl 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 texture\_vs/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, light\_fs, via the LightShader/LightRenderable pair for computing the appropriate pixel colors. The GLSL vertex shader, texture\_vs, is reused because light source illumination involves the same information to be processed at each vertex.



Figure 8-3. LightShader/LightRenderable pair and the corresponding GLSL LightShader

Finally, it is important to note 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 make 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 important for a smooth experience.

## The Simple Light Shader Project

This project demonstrates how to implement and illuminate with a simple point light. 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.simple\_light\_shader folder.



Figure 8-4. Running the Simple Light Shader project

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 observe point light illumination
* To implement a GLSL shader that supports point light illumination

### Creating the GLSL Light Fragment Shader

As with the previous section, the implementation will begin with the GLSL shader. It is not necessary to define a new GLSL vertex shader as the per vertex information and computation involved are identical to that of texture\_vs. A new GLSL fragment shader must be defined to compute the illuminated circle.

1. In the src/glsl\_shaders folder, create a new file and name it light\_fs.glsl.
2. Refer to texture\_fs.glsl and copy all uniform and varying variables. This is an important step because the light\_fs fragment shader will interface to the game engine via the LightShader class. The LightShader class, in turn, will be implemented as a subclass of TextureShader, where the existence of these variables are assumed and references created.

precision mediump float;

// The object that fetches data from texture.

// Must be set outside the shader.

uniform sampler2D uSampler;

// Color of pixel

uniform vec4 uPixelColor;

uniform vec4 uGlobalAmbientColor; // this is shared globally

uniform float uGlobalAmbientIntensity;

// The "varying" keyword is for signifying that the texture coordinate will be

// interpolated and thus varies.

varying vec2 vTexCoord;

... implementation to follow …

1. Now, define the variables to support a point light: on/off switch, color, position, and radius. It is important to note that the position and radius are in units of pixels.

// Light information

uniform bool uLightOn;

uniform vec4 uLightColor;

uniform vec3 uLightPosition; // in pixel space!

uniform float uLightRadius; // in pixel space!

1. Implement the light illumination in the main() function as follows.
2. Step A, sample the texture color and apply the ambient color and intensity.
3. Step B, perform the light source illumination. This is accomplished by determining if the computation is required--testing if the light is switched on, and if the pixel is non-transparent. If both are favorable, the distance between the light position and the current pixel is compared with the light radius to determine if current pixel is inside the volume of illumination. Note that gl\_FragCord.xyz is the GLSL-defined variable for current pixel position and that this computation assumes pixel-space units. When all conditions are favorable, the color of the light is accumulated to the final results.
4. The last step is to apply the tint and to set the final color via gl\_FragColor.

void main(void) {

// Step A: sample the texture and apply ambient

vec4 textureMapColor = texture2D(uSampler, vec2(vTexCoord.s, vTexCoord.t));

vec4 lgtResults = uGlobalAmbientIntensity \* uGlobalAmbientColor;

// Step B: decide if the light should illuminate

if (uLightOn && (textureMapColor.a > 0.0)) {

float dist = length(uLightPosition.xyz - gl\_FragCoord.xyz);

if (dist <= uLightRadius)

lgtResults += uLightColor;

}

lgtResults \*= textureMapColor;

// Step C: 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, textureMapColor.a);

gl\_FragColor = result;

}

### Defining a Light Class

With the GLSL light\_fs shader defined; you can now define a class to encapsulate a point light source for the game engine.

1. Create a new lights folder in the src/engine folder. In the lights folder, add a new file and name it lights.js.
2. Edit lights.js to create the Light class, define the constructor to initialize the light color, position, radius, and on-off status. Remember to export the class.

class Light {

constructor() {

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;

}

... implementation to follow …

}

export default Light;

1. Define the getters and setters for the instance variables.

// simple setters and getters

setColor(c) { this.mColor = vec4.clone(c); }

getColor() { return this.mColor; }

set2DPosition(p) { this.mPosition = vec3.fromValues(p[0], p[1], this.mPosition[2]); }

setXPos(x) { this.mPosition[0] = x; }

setYPos(y) { this.mPosition[1] = y; }

setZPos(z) { this.mPosition[2] = z; }

getPosition() { return this.mPosition; }

setRadius(r) { this.mRadius = r; }

getRadius() { return this.mRadius; }

setLightTo(isOn) { this.mIsOn = isOn; }

isLightOn() { return this.mIsOn; }

Lastly, remember to update the engine access file, index.js, to forward the newly defined functionality to the client.

### Defining the LightShader Class

The LightShader class subclasses from the SpriteShader to encapsulate the communication of the values that are specific to the uniform variables defined for a point light source in the light\_fs fragment shader. In this way, the LightShader class can serve as a convenient interface for the GLSL fragment shader.

1. In the src/engine/shaders folder, create a new file and name it light\_shader.js.
2. Define the LightShader class to be a subclass of SpriteShader. In the constructor, define the necessary variables to support sending the informaiton associated with a point light to the light\_fs fragment shader. The point light information in the engine is stored in mLight**,** while the reference to the Camera is important to convert all information from WC to pixel space. The last four lines of the constructor query to obtain the reference locations to the uniform variables in light\_fs. Don’t forget to export the class.

import SpriteShader from "./sprite\_shader.js";

import \* as glSys from "../core/gl.js";

class LightShader extends SpriteShader {

constructor(vertexShaderPath, fragmentShaderPath) {

// Call super class constructor

super(vertexShaderPath, fragmentShaderPath); // call super class constructor

// 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

this.mCamera = null; // the camera to draw for, required for WC to DC transform

//

// create the references to these uniforms in the LightShader

let shader = this.mCompiledShader;

let gl = glSys.get();

this.mColorRef = gl.getUniformLocation(shader, "uLightColor");

this.mPosRef = gl.getUniformLocation(shader, "uLightPosition");

this.mRadiusRef = gl.getUniformLocation(shader, "uLightRadius");

this.mIsOnRef = gl.getUniformLocation(shader, "uLightOn");

}

... implementation to follow …

}

export default LightShader;

1. Define a simple setter function to associate a light and camera with the shader.

setCameraAndLight(c, l) {

this.mCamera = c;

this.mLight = l;

}

1. Override the activate() function to append the new functionality of loading the point light information in mLight when the light is present. Notice that you still call the activate() function of the super class to communicate the rest of the values to the uniform variables of the light\_fs fragment shader.

activate(pixelColor, trsMatrix, cameraMatrix) {

// first call the super class's activate

super.activate(pixelColor, trsMatrix, cameraMatrix);

if (this.mLight !== null) {

this.\_loadToShader();

} else {

glSys.get().uniform1i(this.mIsOnRef, false); // <-- switch off the light!

}

}

1. Implement the \_loadToShader() function to communicate the values of the point light to the uniform variables in the shader. Recall that this communication is performed via the references created in the constructor and the GLSL set uniform functions. It is important to note that the camera provides the new coordinate space transformation functionality of wcPosToPixel() and wcSizeToPixel(). These two functions ensure corresponding values in the light\_fs are in pixel space such that relevant computations such as distances between positions can be performed. The implementation of these functions will be examined shortly.

\_loadToShader(aCamera) {

let gl = glSys.get();

gl.uniform1i(this.mIsOnRef, this.mLight.isLightOn());

if (this.mLight.isLightOn()) {

let p = this.mCamera.wcPosToPixel(this.mLight.getPosition());

let r = this.mCamera.wcSizeToPixel(this.mLight.getRadius());

let c = this.mLight.getColor();

gl.uniform4fv(this.mColorRef, c);

gl.uniform3fv(this.mPosRef, vec3.fromValues(p[0], p[1], p[2]));

gl.uniform1f(this.mRadiusRef, r);

}

}

### Defining the LightRendererable Class

With the LightShader class defined to interface to the GLSL light\_fs shader, you can now focus on defining a new Renderable class for the game programmer. It is important that a light can shine on and illuminate all Renderable types, including those with texture and animated sprites. For this reason, the new class must encapsulate all existing Renderable functionality. For this reasons, the new class should be a subclass of SpriteAnimateRenderable. You can think of this new class as a SpriteAnimateRenderable that can be illuminated by a Light object.

1. Create a new file in the src/engine/renderables folder and name it light\_rendererable.js.
2. Define the LightRenderable class to extend SpriteAnimateRenderable, set the shader to reference the new LightShader and initialize a Light reference in the constructor. This is the light that shines and illuminates the SpriteAnimateRenderable. Don’t forget to export the class.

import SpriteAnimateRenderable from "./sprite\_animate\_renderable.js";

import \* as defaultShaders from "../core/shader\_resources.js";

class LightRenderable extends SpriteAnimateRenderable {

constructor(myTexture) {

super(myTexture);

super.\_setShader(defaultShaders.getLightShader());

// here is the light source

this.mLight = null;

}

... implementation to follow …

}

export default LightRenderable;

1. Define a draw function to pass the illuminating light source to the LightShader before invoking the superclass draw() function to complete the drawing.

draw(camera) {

this.mShader.setCameraAndLight(camera, this.mLight);

super.draw(camera);

}

1. Lastly, simply add the support to get and set the light.

getLight() { return this.mLight; }

addLight(l) { this.mLight = l; }

Before moving on remember to update the engine access file, index.js, to forward the newly defined functionality to the client.

### Defining a Default LightShader Instance

As discussed when you first defined TextureShader (Chapter 5), only a single instance is required for each shader type and all the shaders are always hidden from the game programmer by a corressponding Renderable type. Each instance of the shader type is created during engine initialization in the shaderResources module in the src/engine/core folder.

You can now modify the engine to support the initializing, loading, and unloading of a LightShader object to be shared engine-wide.

1. Edit shader\_resources.js in the src/engine/core/resources folder to import LightShader; define the path to the GLSL source code, a corresponding variable and access function for the shader.

… identical to previous code …

import LightShader from "../shaders/light\_shader.js";

// Light Shader

let kLightFS = "src/glsl\_shaders/light\_fs.glsl"; // Path to the Light FragmentShader

let mLightShader = null;

function getLightShader() { return mLightShader; }

1. Create a new instance of light shader in the createShaders() function.

function createShaders() {

mConstColorShader = new SimpleShader(kSimpleVS, kSimpleFS);

mTextureShader = new TextureShader(kTextureVS, kTextureFS);

mSpriteShader = new SpriteShader(kTextureVS, kTextureFS);

mLineShader = new LineShader(kSimpleVS, kLineFS);

mLightShader = new LightShader(kTextureVS, kLightFS);

}

1. Load the light shader GLSL source code in the init() function.

function init() {

let loadPromise = new Promise(

async function(resolve) {

await Promise.all([

text.load(kSimpleFS),

text.load(kSimpleVS),

text.load(kTextureFS),

text.load(kTextureVS),

text.load(kLineFS),

text.load(kLightFS)

]);

resolve();

}).then(

function resolve() { createShaders(); }

);

map.pushPromise(loadPromise);

}

1. Remember to release GLSL resources and unload the source code during cleanup.

function cleanUp() {

mConstColorShader.cleanUp();

mTextureShader.cleanUp();

mSpriteShader.cleanUp();

mLineShader.cleanUp();

mLightShader.cleanUp();

text.unload(kSimpleVS);

text.unload(kSimpleFS);

text.unload(kTextureVS);

text.unload(kTextureFS);

text.unload(kLineFS);

text.unload(kLightFS);

}

1. Lastly, export the access function to allow sharing of the created instance in the engine.

export {init, cleanUp,

getConstColorShader, getTextureShader, getSpriteShader, getLineShader, 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, e.g. 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\_main.js and define a PerRenderCache class, in the constructor define variables 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 rendering begins.

class PerRenderCache {

// Information to be updated once per render for efficiency concerns

constructor() {

this.mWCToPixelRatio = 1; // WC to pixel transformation

this.mCameraOrgX = 1; // Lower-left corner of camera in WC

this.mCameraOrgY = 1;

}

}

1. Modify the Camera class to instantiate a new PerRenderCache object. It is important to note that this variable represents local caching of information and should be hidden from the rest of the engine.

constructor(wcCenter, wcWidth, viewportArray, bound) {

… identical to prevous code …

// 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!

}

1. Initiate the per-render cache in the setViewAndCameraMatrix() function by adding a step B4 to calculate and set the cache based on the Camera viewport width, world width, and world height.

setViewAndCameraMatrix() {

… identical to previous code …

// Step B3: first operation to perform is to translate camera center to the origin

mat4.translate(this.mCameraMatrix, this.mCameraMatrix, vec3.fromValues(-center[0], -center[1], 0));

// Step B4: compute and cache per-rendering information

this.mRenderCache.mWCToPixelRatio = this.mViewport[eViewport.eWidth] / this.getWCWidth();

this.mRenderCache.mCameraOrgY = center[1] - (this.getWCHeight() / 2);

this.mRenderCache.mCameraOrgX = center[0] - (this.getWCWidth() / 2);

}

Notice that the PerRenderCache class is completely local to the camera\_main.js file. It is important to hide and carefully handle complex local caching functionality.

#### Adding Camera Transform Functions

Now that the per-render cache is defined and properly initialized, you can extend the functionality of the camera support transformations from WC and pixel space. For code readability and maintainability, this functionality will be implemented in a separate file. 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. Edit the Camera access file, camera.js, to import from the file, camera\_xform.js, which will contain the latest functionality additions, the WC to pixel space transform support.

import Camera from "./camera\_xform.js";

export default Camera;

1. In the src/engine/cameras folder, create a new file and name it camera\_xform.js. Import from camera\_input.js such that you can continue to add new functionality to the Camera class, and do not forget to export.

import Camera from "./camera\_input.js";

import { eViewport } from "./camera\_main.js";

... implementation to follow …

export default Camera;

1. Create a function to 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;

}

1. Define a function to convert from WC to pixel space 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

let x = this.mViewport[eViewport.eOrgX] +

((p[0] - this.mRenderCache.mCameraOrgX) \* this.mRenderCache.mWCToPixelRatio) + 0.5;

let y = this.mViewport[eViewport.eOrgY] +

((p[1] - this.mRenderCache.mCameraOrgY) \* this.mRenderCache.mWCToPixelRatio) + 0.5;

let z = this.fakeZInPixelSpace(p[2]);

return vec3.fromValues(x, y, z);

}

1. Lastly, define a function for converting a length from WC to pixel space by scaling with the mWCToPixelRatio variable.

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 newly defined light functionality.

#### Modifying the Hero and Minion

Modify the Hero and Minion classes to accommodate the new LightRenderable object.

1. Edit the hero.js file in the src/my\_game/objects folder, in the constructor, replace the SpriteRenderable with a LightRenderable instantiation.

constructor(spriteTexture) {

super(null);

this.kDelta = 0.3;

this.mRenderComponent = new engine.LightRenderable(spriteTexture);

… identical to previous code …

}

1. Edit the minion.js file in the src/my\_game/objects folder, in the constructor, replace the SpriteRenderable with a LightRenderable instantiation.

constructor(spriteTexture, atX, atY) {

super(null);

this.kDelta = 0.2;

this.mRenderComponent = new engine.LightRenderable(spriteTexture);

… identical to previous code …

}

#### 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 my\_game\_main.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 my\_game\_main.js file within the src/my\_game 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 that the illuminated circle decreases (smaller intersection area) and increases in size. The sphere/plane intersection result can be verified when you continue to increase/decrease the Z position. The illuminated circle will eventually begin to decrease in size and ultimately disappear completely when the sphere is moved more than its radius away from the Z=0 plane.

You can also 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.

Now, press the WASD keys along with the left mouse button to move the Hero and observe that the point light always follow the Hero and properly illuminates the background. Notice 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 red and green 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 be limited 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 light\_fs fragment shader for illumination computations.



Figure 8-5. Support for multiple light sources

The point light illumination results from the previous project can be improved. You have observed that at its boundary the illuminated circle disappears abruptly with a sharp brightness transition. 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 the following, you will implement a near and far cutoff distance, that is, two distances from the light source at which the distance 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 the dimming of light without changing its color. With these additional parameters it becomes possible to define 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 sophistication of your point light model 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.multiple\_lights folder.



Figure 8-6. Running the Multiple Lights project

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 GLSL shaders
* To understand and examine the distance attenuation effects of light
* To experience controlling and manipulating multiple light sources in a scene

### Modifying the GLSL Light Fragment Shader

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

1. In the light\_fs.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.

**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 the performance.

// 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 {

vec3 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

1. Define LightEffect() function to compute the illumination results from a light source. This function uses the distance between the light and the current pixel to determine whether the pixel lies within the near radius, in between near and far radii, or farther than the far radius. If the pixel position lies within the near radius, there is no attenuation, so the strength is set to 1. If the position is in between the near and far radii, then the strength is modulated by a quadratic function. A distance of greater than the far radius will result in no illumination from the corresponding light source, or a strength of 0.

vec4 LightEffect(Light lgt) {

vec4 result = vec4(0);

float strength = 0.0;

float dist = length(lgt.Position.xyz - gl\_FragCoord.xyz);

if (dist <= lgt.Far) {

if (dist <= lgt.Near)

strength = 1.0; // no attenuation

else {

// simple quadratic drop off

float n = dist - lgt.Near;

float d = lgt.Far - lgt.Near;

strength = smoothstep(0.0, 1.0, 1.0-(n\*n)/(d\*d)); // blended attenuation

}

}

result = strength \* lgt.Intensity \* lgt.Color;

return result;

}

1. Modify the main function to iterate through all the defined light sources and call the LightEffect() function to calculate and accumulate the contribution from the corresponding light in the array.

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;

… identical to previous code …

}

### Modifying the Light Class

The game engine Light object must be modified to reflect the newly added properties in the light\_fs fragment shader: near and far attenuations and intensity.

1. Modify the Lights.js constructor to define variables for the new properties.

constructor() {

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; // effective radius in WC

this.mFar = 10; // within near is full on, outside far is off

this.mIntensity = 1;

this.mIsOn = true;

}

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

setNear(n) { this.mNear = n; }

getNear() { return this.mNear; }

setFar(f) { this.mFar = f; }

getFar() { return this.mFar; }

setIntensity(i) { this.mIntensity = i; }

getIntensity() { return this.mIntensity; }

setLightTo(on) { this.mIsOn = on; }

### Defining the LightSet Class

You will define a LightSet class to facilitate the working with a collection of Light objects. In the src/engine/lights folder, create a new file and name it light\_set.js. Define the basic interface for working with a set of Light objects.

class LightSet {

constructor() { this.mSet = []; }

numLights() { return this.mSet.length; }

getLightAt(index) { return this.mSet[index]; }

addToSet(light) { this.mSet.push(light); }

}

export default LightSet;

Lastly, don’t forget to export the class and remember to update the engine access file, index.js, to forward the newly defined functionality to the client.

### Defining the ShaderLightAt Class

Define the ShaderLightAt class to send information from a Light object to an element in the uLights array in the light\_fs GLSL fragment shader.

1. In the src/engine/shaders folder, create a new file and name it shader\_light\_at.js, define the ShaderLightAt class and the constructor to receive a shader and an index to the uLight array. Don’t forget to export the class.

import \* as glSys from "../core/gl.js";

class ShaderLightAt {

constructor(shader, index) {

this.\_setShaderReferences(shader, index);

}

... implementation to follow …

}

export default ShaderLightAt;

1. Implement the \_setShaderReferences function to set the light property references a specific index in the uLights array in the light\_fs fragment shader.

\_setShaderReferences(aLightShader, index) {

let gl = glSys.get();

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");

}

1. Implement the loadToShader() function to push the properties of a light to the light\_fs fragment shader. Notice that this function is similar to the \_loadToShader() function defined in the light\_shader.js file from previous project. The important difference is that in this case light information is loaded to a specific array index.

loadToShader(aCamera, aLight) {

let gl = glSys.get();

gl.uniform1i(this.mIsOnRef, aLight.isLightOn());

if (aLight.isLightOn()) {

let p = aCamera.wcPosToPixel(aLight.getPosition());

let n = aCamera.wcSizeToPixel(aLight.getNear());

let f = aCamera.wcSizeToPixel(aLight.getFar());

let 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.mIntensityRef, aLight.getIntensity());

}

}

1. Define a simple function to update the on/off status of the light in the array of the light\_fs fragment shader.

switchOffLight() {

let gl = glSys.get();

gl.uniform1i(this.mIsOnRef, false);

}

Note that the ShaderIndexAt class is defined for loading a light source in an array to the GLSL fragment shader. This is an internal engine operation. There is no reason for the game programmer to access this class and thus the engine access file, index.js, should not be modified to forward the definition of this class.

### Modifying the LightShader Class

You must now modify the LightShader object to properly handle the communication between the Light object and the array of lights in the light\_fs fragment shader.

1. Begin by editing the light\_shader.js file, importing ShaderLightAt, and removing the \_loadToShader() function. The actual loading of light information to light\_fs fragment shaders is now handled by the newly defined ShaderLightAt objects.

import ShaderLightAt from "./shader\_light\_at.js";

1. Modify the constructor to define mLights, which is an array of ShaderLightAt objects to correspond to the uLights array defined in the light\_fs fragment shader. It is important to note that the mLights and uLights arrays must be the exact same size.

constructor(vertexShaderPath, fragmentShaderPath) {

// Call super class constructor

super(vertexShaderPath, fragmentShaderPath); // call super class constructor

this.mLights = null; // lights from the Renderable

this.mCamera = null; // the camera to draw for, required for WC to DC transform

//\*\*\*\*\*\*\*WARNING\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// this number MUST correspond to the GLSL uLight[] array size (for LightFS.glsl)

//\*\*\*\*\*\*\*WARNING\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

this.kGLSLuLightArraySize = 4; // <-- make sure this is the same as LightFS.glsl

this.mShaderLights = [];

let i, ls;

for (i = 0; i < this.kGLSLuLightArraySize; i++) {

ls = new ShaderLightAt(this.mCompiledShader, i);

this.mShaderLights.push(ls);

}

}

1. Modify the activate() function to iterate and load the contents of each ShaderLightAt object to the light\_fs 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 light\_fs invocation. For this reason, it is important to ensure all unused lights are switched off. This is ensured by the last while loop in the following code.

activate(pixelColor, trsMatrix, cameraMatrix) {

// first call the super class's activate

super.activate(pixelColor, trsMatrix, cameraMatrix);

// now push the light information to the shader

let numLight = 0;

if (this.mLights !== null) {

while (numLight < this.mLights.length) {

this.mShaderLights[numLight].loadToShader(this.mCamera, this.mLights[numLight]);

numLight++;

}

}

// switch off the left over ones.

while (numLight < this.kGLSLuLightArraySize) {

this.mShaderLights[numLight].switchOffLight(); // switch off unused lights

numLight++;

}

}

1. Rename the setCameraAndLight() function to setCameraAndLights(), in addition to setting the corresponding variables, check to ensure that the light array size is not greater than the defined array size in the light\_fs fragment shader. Lastly, remember to update the corresponding function name in sprite\_shader.js.

setCameraAndLights(c, l) {

this.mCamera = c;

this.mLights = l;

if (this.mLights.length > this.kGLSLuLightArraySize)

throw new Error ("Error: " …);

}

### Modifying the LightRenderable Class

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

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

constructor(myTexture) {

super(myTexture);

super.\_setShader(defaultShaders.getLightShader());

// the light sources

this.mLights = [];

}

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

draw(camera) {

this.mShader.setCameraAndLights(camera, this.mLights);

super.draw(camera);

}

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

getLightAt(index) { return this.mLights[index]; }

addLight(l) { this.mLights.push(l); }

### Testing the Light Sources with MyGame

With proper integration for multiple lights support 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. In order to maintain readability, you will divide the light instantiation and controls into separate files. To avoid redundancy and repetitive code listings, the details to the straightforward implementations are not shown.

1. Modify the my\_game\_main.js file in the src/my\_game 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/my\_game folder, create the new file my\_game\_lights.js to import MyGame class from my\_game\_main.js and to add functionality to instantiate and initialize the lights.
3. In the src/my\_game folder, create the new file my\_game\_light\_control.js to import from my\_game\_lights.js and to continue to add controls of the lights to MyGame.
4. Modify my\_game.js to import from my\_game\_light\_control.js ensuring access to all of the newly defined functionality.

### 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 the game programmer has control over which light illuminates which of the objects: 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 changes as she is moved through the near and far radii of light source 0. With light source 0 selected (type 0), press the C key to increase the near radius of the light. Notice that as the near radius approaches the value of the far, the illuminated circle boundary edge also becomes sharper. Eventually, when near radius is greater than far radius, you can once again observe the sudden brightness change at the boundary. You are observing the violation of the implicit assumption of the underlying illumination model that the near is always less than the far radius. This exact situation can be created by decreasing the far radius with the N key.

You can 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. In particular try adjusting light intensities with the K/L keys to observe the effects of over-saturation, and barely noticeable lighting. You can continue to press on the L key till the intensity becomes negative to create a source that removes color from the scene. 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 targeted regions. However, if you run the previous project and move one of the light sources around you may notice some peculiar effects. Figure 8-7 highlights these effects by comparing the illumination results from the previous project on the left to an illumination that you probably expect on the right. Now, refer to the image on the left. First, note the general uniform lighting within the near cutoff region where the expected brighter spot around the position of the point light source cannot be observed. Second, examine the vertical faces of the geometric block and take note of the bright illumination on the bottom face that is clearly behind, or pointing away from, the light source. Both of these peculiarities are absent from the right image in Figure 8-7.

Although visually odd, results from the left image of Figure 8-7 are to be expected in a 2D world. The vertical faces 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.



Figure 8-7. Left: From previous projcet; Right: Expected illumination

As illustrated by the left drawing in Figure 8-8, 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 8-8 shows that in 3D space the surface normal vectors of an object describe the shape or contour of the object.



Figure 8-8. Surface normal vectors of an object

A human’s observation of light illumination is the result of visible energy from 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 8-9 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 light vector, , at the position. It is important to note that the direction of the vector is always towards the light source and that this is a normalized vector with a magnitude of 1.

Figure 8-9 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,, is perpendicular to its light vector , or . Position B can receive all the energy because its normal vector is pointing in the same direction as its light vector, or . 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 the light vector, or . In an illumination model, the term that the include the computation is referred to as the diffuse, or Lambertian, component.



Figure 8-9. Normal and light vectors and diffuse illumination

The human vision system deduces 3D geometric shape contours based significantly on , or the diffuse component. For example, Figure 8-10 shows a sphere and torus (doughnut shape object) with (the left images) and without (the right images) the corresponding diffuse components. Clearly, in both cases, the 3D contour of the objects is captured by the left versions of the image with the diffuse component.



Figure 8-10. Examples of 3D objects with and without diffuse component

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: a vector in the z direction. This lack of distinct normal vectors for objects implies that it is not possible to compute a distinct diffuse component for objects. Fortunately, similar to how texture mapping addresses the limitation of each geometry having only a single color, normal mapping can resolve the issue of each geometry having only a single normal vector.

Figure 8-11 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 8-11 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 8-11) 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 8-11. Normal mapping with two texture images: the normal and the color texture

Figure 8-12 captures the view of the three corresponding positions labeled on the right images of Figure 8-11, the positions n1, n2, and n3 on the normal texture and the corresponding positions c1, c2, and c3 on the color texture, to illustrate the details of normal mapping. The bottom layer of Figure 8-12 shows that the color texture records colors and the colors c1, c2, and c3 are sampled at those three positions. The middle layer of Figure 8-12 shows that the RGB components of the normal texture record the normal vector xyz values of objects at the corresponding color texture positions. The top layer of Figure 8-12 shows that when illuminated by a light source, with the term properly computed and displayed, the human vision system will perceive a sloped contour.



Figure 8-12. 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. In this way, the rendered image will display contours that resemble the shapes encoded in the normal map.

In the previous project, you expanded the engine to support multiple light sources. In this section, you will define the IllumShader class to generalize a LightShader 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 8-13. The source code of this project is located in the chapter8/8.4.normal\_maps\_and\_illumination\_shaders folder.



Figure 8-13. Running the Normal Maps and Illumination Shaders project

The controls of the project are identical to the previous project:

* 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 of in an illumination model

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 light\_fs.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 from light\_fs.glsl and pasting to a new file, illum\_fs.glsl, in the src/glsl\_shaders folder.
2. Edit the illum\_fs.glsl file and add a sampler2D object, uNormalSampler, to sample the normal map.

uniform sampler2D uSampler;

uniform sampler2D uNormalSampler;

… identical to the variables declare in light\_fs.glsl …

1. 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 computation. Enter code to compute the vector, remember to normalize the vector, and use the result of to scale the light strength accordingly.

vec4 LightEffect(Light lgt, vec3 N) {

vec4 result = vec4(0);

float strength = 0.0;

vec3 L = lgt.Position.xyz - gl\_FragCoord.xyz;

float dist = length(L); if (dist <= lgt.Far) {

if (dist <= lgt.Near) }

… identical to the code in light\_fs.glsl …

}

L = L / dist; // To normalize L

// Not calling the normalize() function to avoid re-computing

// the "dist". This is computationally more efficient.

float NdotL = max(0.0, dot(N, L));

strength \*= NdotL;

}

result = strength \* lgt.Intensity \* lgt.Color;

return result;

}

1. 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 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 vector, N, is then passed on to the LightEffect() function for the illumination calculations.

void main(void) {

// simple tint based on uPixelColor setting

vec4 textureMapColor = texture2D(uSampler, vTexCoord);

vec4 normal = texture2D(uNormalSampler, vTexCoord); // using the same coordinate

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);

}

}

}

… identical to the code in light\_fs.glsl …

}

**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 desired surface geometries. It depends entirely upon the tool or artist that created the map.

### Defining the IllumShader Class

With the Illum\_fs fragment shader supporting normal maps, you can create the JavaScript IllumShader class to interface with it.

1. In the src/engine/shaders folder, create illum\_shader.js, define IllumShader to be a subclass of LightShader to take advantage of the functionality related to light sources. In the constructor define a variable, mNormalSamplerRef, to maintain the reference to the normal sampler in the illum\_fs fragment shader. Don’t forget to export the class.

import LightShader from "./light\_shader.js";

import \* as glSys from "../core/gl.js";

class IllumShader extends LightShader {

constructor(vertexShaderPath, fragmentShaderPath) {

super(vertexShaderPath, fragmentShaderPath); // call super class constructor

let gl = glSys.get();

// reference to the normal map sampler

this.mNormalSamplerRef = gl.getUniformLocation(this.mCompiledShader, "uNormalSampler");

}

... implementation to follow …

}

export default IllumShader;

1. Override and extend the activate() function to binding the normal texture sampler reference to WebGL texture unit 1. You may recall from Chapter 5 that TextureShader binds the color texture sampler to texture unit 0. By binding normal mapping to texture unit 1, the WebGL texture system can work concurrently with two active textures: units 0 and 1. As will be discussed in the next subsection, it is important to configure WebGL, via the texture module, to activate the appropriate texture units for the corresponding purpose: color versus normal texture mapping.

activate(pixelColor, trsMatrix, cameraMatrix) {

// first call the super class's activate

super.activate(pixelColor, trsMatrix, cameraMatrix);

let gl = glSys.get();

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 Texture Module

So far, you have been binding the color texture map to WebGL texture unit 0. With the addition of the normal texture, binding to the unit of WebGL texture system must now be parameterized. Fortunately, this is a straightforward change.

Modify the texture module by opening texture.js in the src/engine/resources folder. Edit the activate() function to accept a second parameter, the WebGL texture unit to bind to. Notice that this is an optional parameter with the default value set to texture unit 0. This is such that no changes are required for any of the existing calls to the activate() function.

function activate(textureName, textureUnit = glSys.get().TEXTURE0) {

let gl = glSys.get();

let texInfo = get(textureName);

// Binds our texture reference to the current webGL texture functionality

gl.activeTexture(textureUnit); // activate the WebGL texture unit

gl.bindTexture(gl.TEXTURE\_2D, texInfo.mGLTexID);

… identical to previous code …

}

### Creating the IllumRenderable Class

You can now define the illumination Renderable class to leverage the newly created illumination shader.

1. Begin by creating illum\_renderable.js in the src/engine/renderables folder, defining the IllumRenderable class to subclass from LightRenderable, and initializing 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. This sharing of texture coordinate implicitly 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. Lastly, don’t forget to export the class.

import \* as texture from "../resources/texture.js";

import \* as glSys from "../core/gl.js";

import LightRenderable from "./light\_renderable.js";

import \* as defaultShaders from "../core/shader\_resources.js";

class IllumRenderable extends LightRenderable {

constructor(myTexture, myNormalMap) {

super(myTexture);

super.\_setShader(defaultShaders.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

}

... implementation to follow …

}

export default IllumRenderable;

1. Next override the draw() function to activate the normal map before calling the draw() method of the super class. Notice the second argument of the texture.activate() function call where the WebGL texture unit 1 is explicitly specified. In concert with IllumShader linking uNormalSampler to WebGL texture unit 1 and illum\_fs samples the uNmormalSamper as a normal map, your engine now supports proper normal mapping.

draw(camera) {

texture.activate(this.mNormalMap, glSys.get().TEXTURE1);

// Here the normal map texture coordinate is copied from those of

// the corresponding sprite sheet

super.draw(camera);

}

Lastly, remember to update the engine access file, index.js, to forward the newly defined functionality to the client.

### Defining a Default IllumShader Instance

Similar to all other shaders in the engine, a default instance of the IllumShader must be defined to be shared. The code involved in defining the default IllumShader instance is identical to those of LightShader presented earlier in this chapter, with the straightforward exception of substituting the corresponding variable names and datatype. Please refer to the Defining a Default LightShader Instance subsection and the shader\_resources.js source code file in the src/engine/core folder for details.

### 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. Edit hero.js in src/my\_game/objects to modify the constructor of the Hero class to instantiate the game object with an IllumRenderable.

constructor(spriteTexture, normalMap) {

super(null);

this.kDelta = 0.3;

this.mRenderComponent = new engine.IllumRenderable(spriteTexture, normalMap);

this.mRenderComponent.setColor([1, 1, 1, 0]);

… identical to previous code …

}

1. In the same folder, edit minion.js to modify the constructor of Minion class to conditionally instantiate the game object with either a LightRenderable or an IllumRenderable when the normal texture map is present.

constructor(spriteTexture, normalMap, atX, atY) {

super(null);

this.kDelta = 0.2;

if (normalMap === null) {

this.mRenderComponent = new engine.LightRenderable(spriteTexture);

} else {

this.mRenderComponent = new engine.IllumRenderable(spriteTexture, normalMap);

}

… identical to previous code …

}

#### Modifying MyGame

You can now modify MyGame to test and display your implementation of the illumination shader. Modify the my\_game\_main.js file in the src/my\_game 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 minimal; 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.

You can verify that the peculiar effects observed on the left image of Figure 8-7 is resolved. For a clearer observation, switch off all other lights (type the light number followed the H key) except leaving light 2 switched-on. Now, move the light position (with the arrow keys) to illuminate the geometric block behind the Hero character, you can move her away with the WASD keys. Verify that you are viewing results similar to those in the right image of Figure 8-7. You should be able to clearly observe the shining spot that corresponds to the point light position. Additionally, take note that the bottom face on the block is only illuminated when the light position is in front of the face, or when the diffuse term, , is positive.

In general, as you move the light source, observe faces with vertical orientations, e.g. the side faces of the geometric block or gaps. As the light position moves across such a boundary, the sign of the 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.

Lastly, the slightly pixelated and rough appearances of the Hero and left Minion attest to the fact that the normal maps for these objects for are generated algorithmically from the corresponding color images and that the maps are not created by an artist.

# 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 8-14 shows an example of three spheres, a simple matted sphere, and specular highlights on a slightly and a highly polished sphere. These highlights are results of the Phong specular term.



Figure 8-14. Specularity and Shininess Of Objects

Figure 8-15 illustrates that given a shiny or reflective surface like a polished floor or a polished 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 a shiny surface is referred to as *specular reflection*, *specular highlight*, or *specularity*.



Figure 8-15. Specularity: the reflection of the light source

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 8-16, where the vector is the reflection direction of the light vector . The specular highlight on an object is visible even when the viewing direction is not perfectly aligned with the vector. Real-life experience also informs you that the further away is from , or the larger the angle-α is, the less likely you will observe the light reflection. In fact, you know that when α is zero, you would observe the maximum light reflection, and when α is 90° or when and are perpendicular, you would observe zero light reflection.



Figure 8-16. The Phong specularity model

The Phone illumination model simulates the characteristics of specularity with a term. When and are aligned, or when α=0°, the specularity term evaluates to 1, and the term drops off to 0 according to the cosine function when the separation between and increases to 90° or when α=90°. The power , referred to as shininess, describes how rapidly the specular highlight rolls off as α increases. The larger the value, the faster the cosine function decreases as α increases, the faster the specular highlight drops off, and the glossier the surface would appear. For example, in Figure 8-14, the left, middle, and right spheres have corresponding values of 0, 5, and 30.

While the term models specular highlight effectively, the cost involved in computing the vector for every shaded pixel can be significant. As illustrated in Figure 8-17, , the halfway vector, is defined as the average of the and vectors. It is observed that β, the angle between the and , can also be used to characterize specular reflection. Though slightly different, produces similar results as with less per-pixel computation cost in computing the vector. The halfway vector will be used to approximate specularity in your implementation.



Figure 8-17. The halfway vector

As illustrated in Figure 8-18, 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: the global ambient lighting and the light source. In this way, to support the Phone illumination model the material property of an object can be represented by , , , and . These stand for three colors, representing the ambient, diffuse, and specular reflectivity, and a floating-point number representing the shininess of an object. With the global ambient light intensity, , and color, ; and the light source intensity, , and color, , the three terms of the Phong illumination model are as follows:

* The ambient term:
* The diffuse term:
* The specular term:

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



Figure 8-18. The Phone illumination model

## Integration of Material in the Game Engine and GLSL Shaders

To implement the Phong illumination model, a Material class that encapsulates the surface material property in Figure 8-18 must be defined and referenced by each IllumRenderable object that is to be shaded by the corresponding illum\_fs fragment shader. Figure 8-19 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 illum\_fs GLSL fragment shader.



Figure 8-19. Support for material

## The Material and Specularity Project

This project demonstrates the implementation of a version of Phong illumination model utilizing the normal map and the position of a camera. 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 8-20. The source code of this project is located in the chapter8/8.5.material\_and\_specularity folder.



Figure 8-20. Running the Material and Specularity project

The main controls of the project are identical to the previous project:

* 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

The material property controls are new to this project:

* Number keys 5 and 6: Select the left minion and the hero
* Number keys 7, 8, and 9: Select the , , and 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 specular reflection and the Phong specular term
* To implement specular highlight illumination in GLSL fragment shaders
* To understand and experience the control of Material of illuminated objects
* To examine specular highlights in illuminated images

### Modifying the GLSL Illumination Fragment Shader

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

1. Edit the illum\_fs.glsl file and define a variable, uCameraPosition, for storing the camera position. This position is used to compute the vector, the viewing direction. 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 Ka, Kd, Ks, and n and the terms in the Phong illumination model in Figure 8-18.

// for supporting a simple Phong-like illumination model

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;

1. To support readability, mathematical terms in the illumination model will be defined into separate functions. You will begin by defining the DistanceDropOff() function to perform the exact same near/far cutoff computation as in the previous project.

// Computes the L-vector, returns strength

float DistanceDropOff(Light lgt, float dist) {

float strength = 0.0;

if (dist <= lgt.Far) {

if (dist <= lgt.Near)

strength = 1.0; // no attenuation

else {

// simple quadratic drop off

float n = dist - lgt.Near;

float d = lgt.Far - lgt.Near;

strength = smoothstep(0.0, 1.0, 1.0-(n\*n)/(d\*d)); // blended attenuation

}

}

return strength;

}

1. Define the function to compute the diffuse term. Notice that the texture map color is applied to the diffuse term.

vec4 DiffuseResult(vec3 N, vec3 L, vec4 textureMapColor) {

return uMaterial.Kd \* max(0.0, dot(N, L)) \* textureMapColor;

}

1. Define the function to compute the specular term. The vector, V, is computed by normalizing the results of subtracting uCameraPosition from the current pixel position, 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.

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);

}

1. Now you can implement the Phong illumination model to accumulate the diffuse and specular terms. Notice that lgt.Intensity, and lgt.Color, in Figure 8-18, are factored out and multiplied to the sum of diffuse and specular results. The scaling by the light strength based on near/far cutoff computation, strength, is the only difference between this implementation and the diffuse/specular terms listed in Figure 8-18.

vec4 ShadedResult(Light lgt, vec3 N, vec4 textureMapColor) {

vec3 L = lgt.Position.xyz - gl\_FragCoord.xyz;

float dist = length(L);

L = L / dist;

float strength = DistanceDropOff(lgt, dist);

vec4 diffuse = DiffuseResult(N, L, textureMapColor);

vec4 specular = SpecularResult(N, L);

vec4 result = strength \* lgt.Intensity \* lgt.Color \* (diffuse + specular);

return result;

}

1. 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 illum\_fs.glsl file from the previous project. The only important differences are highlighted in bold.

void main(void) {

… identical to previous code …

vec3 N = normalize(normalMap.xyz);

vec4 shadedResult = uMaterial.Ka + (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) {

shadedResult += ShadedResult(uLights[i], N, textureMapColor);

}

}

}

… identical to previous code …

}

### Defining the Material Class

As described, a simple Material class is required to encapsulate the per-Renderable material property for the Phong illumination model.

1. Create material.js in the src/engine folder, define the Material class, and in the constructor initialize the variables as defined in the surface material property in Figure 8-18. Notice that ambient, diffuse, and specular (Ka, Kd, and Ks) are colors, while shininess is a floating-point number.

class Material {

constructor() {

this.mKa = vec4.fromValues(0.0, 0.0, 0.0, 0);

this.mKs = vec4.fromValues(0.2, 0.2, 0.2, 1);

this.mKd = vec4.fromValues(1.0, 1.0, 1.0, 1);

this.mShininess = 20;

}

... implementation to follow …

}

export default Material;

1. Provide straightforward get and set accessors to the variables.

setAmbient(a) { this.mKa = vec4.clone(a); }

getAmbient() { return this.mKa; }

setDiffuse(d) { this.mKd = vec4.clone(d); }

getDiffuse() { return this.mKd; }

setSpecular(s) { this.mKs = vec4.clone(s); }

getSpecular() { return this.mKs; }

setShininess(s) { this.mShininess = s; }

getShininess() { return this.mShininess; }

Note that the Material class is designed to represent material properties of Renderable objects and must be accessible to the game programmer. As such, remember to update the engine access file, index.js, to forward the newly defined functionality to the client.

### Defining the ShaderMaterial Class

Similar to defining the ShaderIndexAt class to pass light source information at an array index to GLSL fragment shader, a new ShaderMaterial class should be defined to communicate the contents of Material to the GLSL illum\_fs shader. Similar to the implementation of ShaderIndexAt, the ShaderMaterial class will also be defined in the src/engine/shaders folder.

1. Create shader\_material.js in the src/engine/shaders folder, define ShaderMaterial class and in the constructor initialize the variables as references to the ambient, diffuse, specular, and shininess in the illum\_fs GLSL shader.

import \* as glSys from "../core/gl.js";

class ShaderMaterial {

constructor(aIllumShader) {

let gl = glSys.get();

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");

}

... implementation to follow …

}

export default ShaderMaterial;

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

loadToShader(aMaterial) {

let gl = glSys.get();

gl.uniform4fv(this.mKaRef, aMaterial.getAmbient());

gl.uniform4fv(this.mKdRef, aMaterial.getDiffuse());

gl.uniform4fv(this.mKsRef, aMaterial.getSpecular());

gl.uniform1f(this.mShineRef, aMaterial.getShininess());

}

Similar to the ShaderIndexAt class, the ShaderMaterial class is defined for loading a material to the GLSL fragment shader. This is an internal engine operation. There is no reason for the game programmer to access this class and thus the engine access file, index.js, should not be modified to forward the definition of this class.

### Modifying the IllumShader Class

Recall that the IllumShader class is the engine’s interface to the corresponding GLSL illum\_fs fragment shader. Now the IllumShader class must be modified to support the newly defined Phong illumination functionality in illum\_fs. This support can be accomplished by modifying the IllumShader to define a ShaderMaterial object to load the contents of a Material object to the illum\_fs fragment shader.

1. Edit illum\_shader.js in src/glsl\_shaders/illum\_fs.glsl to import ShaderMaterial and modify the constructor to define new variables, mMaterial and mCameraPos, to support Phong illumination computation. Then define the variables mMaterialLoader and mCameraPosRef for keeping references and for loading the corresponding contents to the uniform variables in the shader.

import ShaderMaterial from "./shader\_material.js";

constructor(vertexShaderPath, fragmentShaderPath) {

// Call super class constructor

super(vertexShaderPath, fragmentShaderPath); // call super class constructor

// this is the material property of the Renderable

this.mMaterial = null;

this.mMaterialLoader = new ShaderMaterial(this.mCompiledShader);

let gl = glSys.get();

// Reference to the camera position

this.mCameraPos = null; // points to a vec3

this.mCameraPosRef = gl.getUniformLocation(this.mCompiledShader, "uCameraPosition");

// reference to the normal map sampler

this.mNormalSamplerRef = gl.getUniformLocation(this.mCompiledShader, "uNormalSampler");

}

1. Modify the activate() function to load the material and camera position to the illum\_fs fragment shader.

activate(pixelColor, trsMatrix, cameraMatrix) {

// first call the super class's activate

super.activate(pixelColor, trsMatrix, cameraMatrix);

let gl = glSys.get();

gl.uniform1i(this.mNormalSamplerRef, 1); // binds to texture unit 1

this.mMaterialLoader.loadToShader(this.mMaterial);

gl.uniform3fv(this.mCameraPosRef, this.mCameraPos);

}

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

setMaterialAndCameraPos(m, p) {

this.mMaterial = m;

this.mCameraPos = p;

}

### Modifying the IllumRenderable Class

You can now the IllumRenderable class to include a material property and to properly support IllumShader. This is a straightforward change.

1. Edit illum\_renderable.js in the src/engine/renderables folder and modify the constructor to instantiate a new Material object.

import Material from "../material.js";

constructor(myTexture, myNormalMap) {

… identical to previous code …

// Material for this Renderable

this.mMaterial = new Material();

}

1. Update the draw() function to set the material and camera position to the shader before the actual rendering. Notice that in the call to camera.getWCCenterInPixelSpace() the camera position is properly transformed into pixel space.

draw(camera) {

texture.activate(this.mNormalMap, glSys.get().TEXTURE1);

this.mShader.setMaterialAndCameraPos(

this.mMaterial, camera.getWCCenterInPixelSpace());

super.draw(camera);

}

1. Define a simple accessor for the material object.

getMaterial() { return this.mMaterial; }

### Modifying the Camera Class

As you have seen in the illum\_fs fragment shader implementation, the camera position required for computing the 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 the pixel space camera position should be computed once and cached for each drawing cycle. The PerRenderCache class, defined in the Light Source Project specifically for caching per-drawing cycle information, is ideal for caching the pixel space camera position.

1. Edit the camera\_main.js file and add a vec3 to the PerRenderCache to cache the camera’s position in pixel space.

class PerRenderCache {

// Information to be updated once per render for efficiency concerns

constructor() {

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);

}

}

1. 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 piece of information represents depth and is required for the illumination computation.

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

1. In step B4 of the setViewAndCameraMatrix() 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[eViewport.eWidth] / this.getWCWidth();

this.mRenderCache.mCameraOrgX = center[0] – (this.getWCWidth() / 2);

this.mRenderCache.mCameraOrgY = center[1] – (this.getWCHeight() / 2);

let 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);

1. Define the accessor for the camera position in pixel space.

getWCCenterInPixelSpace() { 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 now exhibit specularity. To ensure prominence of specular reflection, the specular material property, Ks, of the background object is set to bright red in the init() function.

A new function, \_selectCharacter(), is defined to allow the user to work with the material property of either the Hero or the left Minion object. The file my\_game\_material\_control.js implements the actual user interaction for controlling the selected material property.

### Observations

You can run the project and interactively control the material property of the currently selected object (type keys 5 to select the left Minion and 6 for the Hero). 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. You can observe red highlights along vertical boundaries in the background image. If you are unsure, pay attention to the top right region of the background image, select light 3 (type the 3 key) and toggle the on/off switch (typing the H key). Notice that as the light toggles from off to on, the entire top right region becomes brighter with a red highlight along the vertical boundary. This red highlight is the reflection of light 3 towards the camera. Now, with light 3 switched on, move it towards the left and right (the left/right-arrow keys). Observe how the highlight intensifies and then fades as results of the angle between the halfway vector, , and the face normal vector, , changes.

You can also adjust the material to observe specularity on the Hero. Now, select the Hero object (type the 6 key), decrease its diffuse material property (type R, Y, and I keys at the same time) to around 0.2, increase the specularity property (type 9 to select Specular, and then E, T, and U keys at the same time) to values beyond 1. With this setting, the diffused term is reduced and specular highlight emphasized, you can observe a dark Hero figure with bright highlight spots. If you are unsure, try toggling light 0 (type the 0 key) on/off (type the H key). Relatively small objects, such as the Hero, does not occupy many pixels, the associated highlight is likely to span even smaller number of pixels and can be challenging to observe. Specular highlights can convey subtle and important effects; however, its usage can also be challenging to master.

Experiment with selecting and manipulating the material property of the left Minion object and moving the other light sources around to observe the red specular highlight on the background.

# Light Source Types

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

A directional light, in contrast to the point light, does not have a light position or a range, rather it illuminates everything in a specific direction. These characteristics, while they may seem unintuitive, are perfect for general background lighting. This is the case in the real world. During the day, the general environment is illuminated by the Sun where rays from the Sun can conveniently be modelled as a directional light. The light rays from the Sun, from the perspective of the earth, are practically parallel coming from a fixed direction and these rays illuminate everything. 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 8-21, 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.



Figure 8-21. A spotlight and its parameters

**Note** In illustrative diagrams, like Figure 8-21, 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 mathematical meanings. These illustrative diagrams are contrasted with vector diagrams that explain illumination computations, like Figures 8-15 and 8-16. In illumination vector diagrams, all vectors always point away from the position to be illuminated, and assumed to be normalized with a magnitude of 1.

## The Directional and Spotlights Project

This project demonstrates how to integrate directional lights and spotlights into your engine to support a wider range of illumination effects. You can see an example of the project running in Figure 8-22. The source code of this project is located in the chapter8/8.6.directional\_and\_spotlights folder.



Figure 8-22. Running the Directional and Spotlights project

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, note that this has no effect on the directional light (light 1)
* Arrow keys with spacebar pressed: Change the direction of the currently selected light, note that this has no effect on the point light (light 0)
* Z/X key: Increases/decreases the light z position, note that this has no effect on the directional light (light 1)
* C/V and B/N keys: Increase/decrease the inner and outer cone angles of the selected light, note that these only affect the two spotlights in the scene (lights 2 and 3)
* 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 , , and 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

### 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 illum\_fs.glsl and defining constants for the three light types. Notice that to support proper communications between the GLSL 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 \*\*\*\*\*\*

1. Expand the light struct 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. As will be detailed next, instead of the actual angle values, the cosines of the inner and outer angles are stored in the struct to facilitate efficient 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.

struct Light {

vec3 Position; // in pixel space!

vec3 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

};

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

float AngularDropOff(Light lgt, vec3 lgtDir, vec3 L) {

float strength = 0.0;

float cosL = dot(lgtDir, L);

float num = cosL - lgt.CosOuter;

if (num > 0.0) {

if (cosL > lgt.CosInner)

strength = 1.0;

else {

float denom = lgt.CosInner - lgt.CosOuter;

strength = smoothstep(0.0, 1.0, pow(num/denom, lgt.DropOff));

}

}

return strength;

}

The parameter lgt is a spotlight in the Light struct, lgtDir is the direction of the spotlight (or Light.Direction normalized), 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 perform the computations based on cosines of the angular displacements. Figure 8-23 illustrates the parameters involved in angular attenuation computation.



Figure 8-23. Computing the angular attenuation of a spotlight

1. The cosL is the dot product of L with lgtDir; it records the angular displacement of the position currently being illuminated.
2. The num variable stores the difference between cosL and cosOuter. A negative num would mean that the position currently being illuminated is outside the outer cone where the position will not be lit and thus no further computation is required.
3. If the point to be illuminated is within the inner cone, cosL would be less than lgt.CosInner, and full strength of the light, 1.0, will be returned.
4. If the point to be illuminated is in between the inner and outer cone angles, use the smoothstep() function to compute the effective strength from the light.
5. Modify the ShadedResults() function to handle each separate case of light source type before combining the results into a color.

vec4 ShadedResult(Light lgt, vec3 N, vec4 textureMapColor) {

float aStrength = 1.0, dStrength = 1.0;

vec3 lgtDir = -normalize(lgt.Direction.xyz);

vec3 L; // light vector

float dist; // distance 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

aStrength = AngularDropOff(lgt, lgtDir, L);

}

if (lgt.LightType != eDirectionalLight) {

// both spot and point light has distance dropoff

dStrength = DistanceDropOff(lgt, dist);

}

vec4 diffuse = DiffuseResult(N, L, textureMapColor);

vec4 specular = SpecularResult(N, L);

vec4 result = aStrength \* dStrength \* lgt.Intensity \* lgt.Color \* (diffuse + specular);

return result;

}

#### Modifying the GLSL Light Fragment Shader

You can now modify the GLSL light\_fs fragment shader to support the two new light types. The modifications involved are remarkably similar to the changes made for illum\_fs, 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 compute the strengths from the light. Please refer to the light\_fs.glsl source code file for details of the implementation.

### Modifying the Light Class

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

1. Edit light.js and define and export 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 illum\_fs and light\_fs shaders.

// \*\*\*\* WARNING: The following enumerate values must be identical to

// the values of

//

// ePointLight, eDirectionalLight, eSpotLight

//

// defined in LightFS.glsl and IllumFS.glsl

const eLightType = Object.freeze({

ePointLight: 0,

eDirectionalLight: 1,

eSpotLight: 2

});

export { eLightType }

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

constructor() {

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 = eLightType.ePointLight;

this.mIsOn = true;

}

1. 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 ShaderLightAt Class

Recall that the ShaderLightAt class 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 shader\_light\_at.js to import the eLightType enumerated type from light.js.

import { eLightType } from "../lights/light.js";

1. Modify the \_setShaderReferences() function to set the references to the newly added light properties.

\_setShaderReferences(aLightShader, index) {

let gl = glSys.get();

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");

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");

}

1. 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.

loadToShader(aCamera, aLight) {

let gl = glSys.get();

gl.uniform1i(this.mIsOnRef, aLight.isLightOn());

// Process a light only when it is switched on

if (aLight.isLightOn()) {

… identical to previous code …

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());

// Point light does not need the direction

if (aLight.getLightType() === eLightType.ePointLight) {

gl.uniform3fv(this.mDirRef, vec3.fromValues(0, 0, 0));

} else {

// either spot or directional lights: must compute direction

let d = aCamera.wcDirToPixel(aLight.getDirection());

gl.uniform3fv(this.mDirRef, vec3.fromValues(d[0], d[1], d[2]));

if (aLight.getLightType() === eLightType.eSpotLight) {

gl.uniform1f(this.mInnerRef, Math.cos(0.5 \* aLight.getInner())); // stores the cosine of half of inner cone angle

gl.uniform1f(this.mOuterRef, Math.cos(0.5 \* aLight.getOuter())); // stores the cosine of half of outer cone angle

gl.uniform1f(this.mDropOffRef, aLight.getDropOff());

}

}

}

}

Note, for mInnerRef and 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 Class

Directional lights and spotlights require a light direction, and the GLSL illum\_fs and light\_fs shaders expect this direction to be specified in pixel space. Edit the camera\_xform.js file of the Camera object to define the wcDirToPixel() function to transform a direction from WC to pixel space.

Camera.prototype.wcDirToPixel = function (d) { // d is a vec3 direction in WC

// Convert the position to pixel space

let x = d[0] \* this.mRenderCache.mWCToPixelRatio;

let y = d[1] \* this.mRenderCache.mWCToPixelRatio;

let z = d[2];

return vec3.fromValues(x, y, z);

}

### Testing the New Light Types

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

### 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 typing 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 red spots of specular reflections. Now, type the H key to switch off the directional light and observe the entire scene becomes darker. Without any kinds of attenuation, directional lights can be used as effective tools for brightening the entire scene.

Type the 2 or 3 keys to select one of the spotlights. Once again, by holding the spacebar while taking turns pressing the left/right or up/down keys to 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. You can consider switching off the direction light, light 1, for a clearer observation of the spotlight effects.

Try experimenting with the different light settings, including overlapping the light illumination regions and setting the light intensities, the K and L keys, 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 or occluded. As an everyday phenomenon, shadow is something you observe but probably do not give much thought to. However, shadow plays a vital role in the visual perception system of humans. 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 can increase the quality of appearance and fidelity. 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. A position is in shadow with respect to a light source if something occludes it from the light source, or, the position is not visible from the light source. 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, and this operation must be performed for every pixel being illuminated. Algorithmically, this is a challenging problem because the solutions for visibility must be available within the fragment shader during illumination computation, again, for every pixel being illuminated.

Because of the computation and algorithmic challenges, instead of simulating shadow according to the physical world, many videogames approximate or create shadow-like effects for only selected objects based on dedicated hardware resources. In this section, you will learn about approximating shadows by selecting dedicated shadow casters and receivers based on the WebGL stencil buffer.

Figure 8-24 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 computation and thus will not receive the shadow.

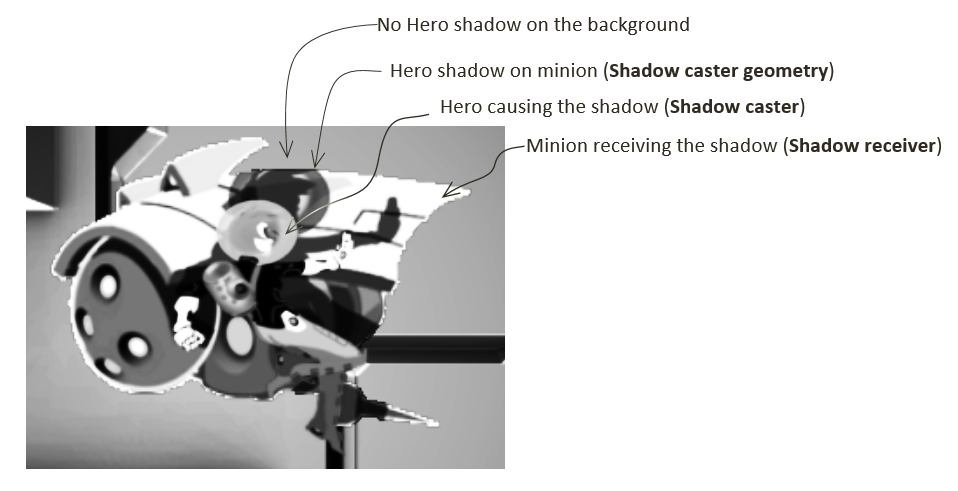


Figure 8-24. Hero casting shadow on the minion but not on the background

To properly simulate and render the shadow in Figure 8-24, as illustrated in Figure 8-25, there are three important elements.

* Shadow caster: This is the object that causes the shadow. In the Figure 8-24 example, the Hero object is the shadow caster.
* Shadow receiver: This is the object that the shadow appears on. In the Figure 8-24 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 8-24 example, the dark imprint of the hero appearing on the minion behind the actual hero object is the shadow caster geometry.



Figure 8-25. The three participating elements of shadow simulation: the caster, the caster geometry, and the receiver

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 8-24, 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.

Take note that shadow is actually a visual effect where colors on objects appear darker because light energy is obstructed. The important point to note is that there are no new objects or geometries involved when a human observes shadows. This is in stark contrast to the described algorithm, where shadows are simulated by the shadow caster geometry, a dark color object. This dark color object does not actually exist in the scene. It is algorithmically created to approximate the visual perception of light being occluded. This creation and rendering of extra geometry to simulate the results of human visual perception, while interesting, has its own challenges.

As depicted in Figure 8-26, the illusion of shadow breaks down when the shadow caster geometry extends beyond the bounds of the shadow receiver. Such situations must be properly resolved in order for shadow to appear genuine. An example of proper handling of this situation can be observed in Figure 8-24, the top portion of the hero helmet shadow extends beyond the bounds of the minion and is not drawn.



Figure 8-26. Shadow caster extends beyond the bounds of shadow receiver

Fortunately, the WebGL stencil buffer is designed specifically to resolve these types of situations. The WebGL stencil buffer can be configured as a 2D array 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 8-27 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 a triangular region appearing in the canvas (bottom layer). This triangular region is formed by pixels that correspond to the on positions of the triangle in the stencil buffer. In this way, the stencil buffer acts exactly like a stencil over the canvas where only the on-regions can be drawn on.



Figure 8-27. The WebGL stencil buffer

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 8-24 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 shadow simulation is actually an algorithmic creation with effects that can be used to approximate human perception. 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

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 on the 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 listed code 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 pixels that are associated with the white triangle in Figure 8-27, enabling the region that can be drawn. 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 enabled to 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 8-28. The source code of this project is located in the chapter8/8.7.shadow\_shaders folder.



Figure 8-28. Running the Shadow Shaders project

The controls of this project are identical to previous 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, note that this has no effect on the directional light (light 1)
* Arrow keys with spacebar pressed: Change the direction of the currently selected light, note that this has no effect on the point light (light 0)
* Z/X key: Increases/decreases the light z position, note that this has no effect on the directional light (light 1)
* C/V and B/N keys: Increase/decrease the inner and outer cone angles of the selected light, note that these only affect the two spotlights in the scene (lights 2 and 3)
* 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 , , and 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 approximated by algorithmically defining and 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

### Creating 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.

#### Defining the GLSL Shadow Caster Fragment Shader

The GLSL shadow\_caster\_fs fragment shader supports the drawing of the shadow caster geometries. Refer to Figure 8-25, the shadow caster geometry is the piece of geometry that fakes being the shadow of the shadow caster. This geometry is typically scaled by the engine according to its distance from the shadow caster, the further from the caster the larger this geometry.

In the fragment shader, this geometry should be rendered as a dark colored object to create the illusion of it being a shadow. Note that one shader caster geometry is required for each shadow casting light source, as such, the fragment shader only supports one light source. Finally, the degree of darkness of this object depends on the effective strength of the shadow casting light sources and thus, the fragment shader must define functionality to compute strengths from each type of light sources.

1. In the src/glsl\_shaders folder, create a file shadow\_caster\_fs.glsl. Since all light types can cast shadow, existing light structures must be supported. Now, copy the Light struct and light type constants from light\_fs (not shown). These data structure and constants must be exactly the same such that the corresponding interfacing shader in the engine can reuse existing utilities that supports LightShader. The only difference is, since a shadow caster geometry must be defined for each light source, the uLight array size is exactly 1 in this case.
2. Define constants for shadow rendering. The kMaxShadowOpacity as how opaque shadows should be and kLightStrengthCutOff is a cutoff threshold where a light with intensity less than this value will not cast shadows.

#define kMaxShadowOpacity 0.7 // max of shadow opacity

#define kLightStrengthCutOff 0.05 // any less will not cause chadow

1. To properly support shadow casting from the three different light types, AngularDropOff() and DistanceDropOff(), functions must also be defined in exactly the same manner as those in light\_fs (and illum\_fs). You can copy these functions from light\_fs. Note that since there is only one light source in the uLight array, you can remove the light parameter from these functions and refer directly to uLight[0] in the computation. This parameter replacement is the only modification required and thus the code is not shown here.
2. Remember that shadow is observed because of light occlusion and is independent from the color of the light source. Now, modify the LightStrength() function to compute the light strength arriving at the position to be illuminated instead of a shaded color.

float LightStrength() {

float aStrength = 1.0, dStrength = 1.0;

vec3 lgtDir = -normalize(uLights[0].Direction.xyz);

vec3 L; // light vector

float dist; // distance 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

aStrength = AngularDropOff(lgtDir, L);

}

if (uLights[0].LightType != eDirectionalLight) {

// both spot and point light has distance dropoff

dStrength = DistanceDropOff(dist);

}

float result = aStrength \* dStrength;

return result;

}

Compared the listed LightStrength() with the same function in light\_fs, there are two main differences. First, the function does not consider the color of the light and returns a float, the aggregated strength of the light source. Second, since the uLight array is of size 1, the function removed the Light parameter and referred to uLight[0] in the computation.

1. Compute the color of 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 actual color of the shadow is not exactly black or opaque. Instead, it is a blend of the programmer-defined uPixelColor and the sampled transparency from the texture map.

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);

}

#### Defining the GLSL Shadow Receiver Fragment Shader

The GLSL shadow\_receiver\_fs 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 where any value returned in gl\_FragColor will switch the corresponding pixel to on. For this reason, transparent receiver fragments must be discarded.

1. Under the src/glsl\_shaders folder, create shadow\_receiver\_fs.glsl, and define a sampler2D object to sample the color texture map of the shadow receiver object. In addition, define the constant kSufficientlyOpaque to be the threshold where fragments with less opacity 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;

uniform vec4 uPixelColor;

// 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 uSampler and vTexCoord must not be changed. These correspond to the variables names defined in texture\_fs.glsl, and the game engine can use the existing SpriteShader to facilitate the loading of information to this shader.

1. Implement the main() function to sample the texture of shadow receiver object, test for opacity threshold in determining if shadow could be received.

void main(void)

{

vec4 texFragColor = texture2D(uSampler, vTexCoord);

if (texFragColor.a < kSufficientlyOpaque)

discard;

else

gl\_FragColor = vec4(1, 1, 1, 1);

}

### Interfacing the GLSL Shadow 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.

* First, only one new engine shader type is required for supporting shadow\_caster\_fs. With the strategic variable naming in the shadow\_receiver\_fs shader, the existing SpriteShader object can be used to communicate with the shadow\_receiver\_fs GLSL fragment shader.
* Second, no new Renderable classes are required. The Renderable classes are designed to support the drawing and manipulation of game objects with the corresponding shaders. In this way, Renderable objects are visible to the players. In the case of shadow shaders, shadow\_caster\_fs draws shadow caster geometries and shadow\_receiver\_fs draws the shadow receiver geometry into the stencil butter. Notice that neither of the shaders is designed to support drawing of objects that are visible to the players. For these reasons, there is no need for the corresponding Renderable objects.

#### Creating the Shadow Caster Shader

A JavaScript Shader class 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 shadow\_caster\_fs fragment shader.

1. Under the src/engine/shaders folder, create shadow\_caster\_shader.js, define the ShadowCasterShader class to inherit from SpriteShader. Since each shadow caster geometry is created by one casting light source, define a single light source for the shader.

import SpriteShader from "./sprite\_shader.js";

import ShaderLightAt from "./shader\_light\_at.js";

class ShadowCasterShader extends SpriteShader {

// constructor

constructor(vertexShaderPath, fragmentShaderPath) {

super(vertexShaderPath, fragmentShaderPath); // call super class constructor

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

this.mCamera = null;

// \*\*\*\* The GLSL Shader must define uLights[1] (array size of 1) as the only light source!!

this.mShaderLight = new ShaderLightAt(this.mCompiledShader, 0);

}

... implementation to follow …

}

export default ShadowCasterShader;

1. Override the activate() function to ensure the single light source is properly loaded to the shader.

// Overriding the activation of the shader for rendering

activate(pixelColor, trsMatrix, cameraMatrix) {

// first call the super class's activate

super.activate(pixelColor, trsMatrix, cameraMatrix);

this.mShaderLight.loadToShader(this.mCamera, this.mLight);

}

1. Define a function to set the current camera and light source for this shader.

setCameraAndLights(c, l) {

this.mCamera = c;

this.mLight = l;

}

#### 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. Modify shader\_resources.js in the src/engine/core folder to import ShadowCasterShader and define the constants and variables for the two new shadow-related shaders.

import ShadowCasterShader from "../shaders/shadow\_caster\_shader.js";

// Shadow shaders

let kShadowReceiverFS = "src/glsl\_shaders/shadow\_receiver\_fs.glsl"; // Path to the FragmentShader

let mShadowReceiverShader = null;

let kShadowCasterFS = "src/glsl\_shaders/shadow\_caster\_fs.glsl"; // Path to the FragmentShader

let mShadowCasterShader = null;

1. Edit the createShaders() function to define engine shaders to interface to the new GLSL fragment shaders. Notice that both of the engine shaders are based on the texture\_vs GLSL vertex shader. In addition, as discussed, the engine SpriteShader is created to interface to the shadow\_receiver\_fs GLSL fragment shader.

function createShaders() {

… identical to previous code …

mIllumShader = new IllumShader(kTextureVS, kIllumFS);

mShadowCasterShader = new ShadowCasterShader(kTextureVS, kShadowCasterFS);

mShadowReceiverShader = new SpriteShader(kTextureVS, kShadowReceiverFS);

}

1. The rest of the modifications to the shader\_resources.js file is routine including: defining accessors, loading and unloading the GLSL source code files, cleaning up the shaders, and exporting the accessors. 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.

### Configuring and Supporting WebGL Buffers

Three modifications are necessary to integrate WebGL stencil buffer into the game engine. First, the WebGL stencil buffer must be enabled and properly configured. Second, functions must be defined to support the drawing with stencil buffer. Third, buffer must be properly cleared before each drawing cycle.

1. Edit the gl.js file in the src/engine/core folder to enable and configure WebGL stencil buffer during engine initialization. In the init() 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.

function init(htmlCanvasID) {

… identical to previous code …

mGL = mCanvas.getContext("webgl2", {alpha: false, depth: true, stencil: true}) ||

mCanvas.getContext("experimental-webgl2", {alpha: false, depth: true, stencil: true});

… identical to previous code …

// make sure depth testing is enabled

mGL.enable(mGL.DEPTH\_TEST);

mGL.depthFunc(mGL.LEQUAL);

}

1. Continue working with gl.js, define functions to begin, end, and disable drawing with the stencil buffer. Remember to export these new stencil buffer support functions.

function beginDrawToStencil(bit, mask) {

mGL.clear(mGL.STENCIL\_BUFFER\_BIT);

mGL.enable(mGL.STENCIL\_TEST);

mGL.colorMask(false, false, false, false);

mGL.depthMask(false);

mGL.stencilFunc(mGL.NEVER, bit, mask);

mGL.stencilOp(mGL.REPLACE, mGL.KEEP, mGL.KEEP);

mGL.stencilMask(mask);

}

function endDrawToStencil(bit, mask) {

mGL.depthMask(mGL.TRUE);

mGL.stencilOp(mGL.KEEP, mGL.KEEP, mGL.KEEP);

mGL.stencilFunc(mGL.EQUAL, bit, mask);

mGL.colorMask(true, true, true, true);

}

function disableDrawToStencil() { mGL.disable(mGL.STENCIL\_TEST); }

1. Edit the engine access file, index.js, in the src/engine folder to clear the stencil and depth buffers when clearing the canvas in the clearCanvas() function.

function clearCanvas(color) {

… identical to previous code …

gl.clear(gl.COLOR\_BUFFER\_BIT | gl.STENCIL\_BUFFER\_BIT | gl.DEPTH\_BUFFER\_BIT);

}

### Defining Shadow Support for Game Developers

As described when defining ShadowCasterShader, Renderable classes should not be defined to pair with the shadow caster and receiver shaders as that would allow game developers the capabilities to manipulate the algorithmically created objects as regular game objects. Instead, the ShadowCaster and ShadowReceiver classes are introduced to allow the game developers to create shadows without granting access to manipulate the underlying geometries.

#### Defining the Shadow Caster Class

Instead of the familiar Renderable class hierarchy, the ShadowCaster class is defined to encapsulate the functionality of the implicitly defined shadow caster geometry. Recall from Figure 8-25, the shadow caster geometry is derived algorithmically for each shadow casting light sources based on the positions of the shadow caster, a Renderable, and the shadow receiver, another Renderable, objects.

To support receiving shadows on an animated sprite element, the shadow receiver must be of SpriteRenderable its subclasses. The shadow casting Renderable object must be able to receive light sources and thus is must be of LightRenderable or its subclasses. A 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 class are as follows.

1. Create the new src/engine/shadows folder for organizing shadow-related support files, and the shadow\_caster.js file in the folder.
2. Create the ShadowCaster class, define the constructor to initialize the instance variables and constants required for caster geometry computations.

import \* as shaderResources from "../core/shader\_resources.js";

import SpriteRenderable from "../renderables/sprite\_renderable.js";

import Transform from "../utils/transform.js";

import { eLightType } from "../lights/light.js";

// shadowCaster: must be GameObject referencing at least a LightRenderable

// shadowReceiver: must be GameObject referencing at least a SpriteRenderable

class ShadowCaster {

constructor(shadowCaster, shadowReceiver) {

this.mShadowCaster = shadowCaster;

this.mShadowReceiver = shadowReceiver;

this.mCasterShader = shaderResources.getShadowCasterShader();

this.mShadowColor = [0, 0, 0, 0.2];

this.mSaveXform = new Transform();

this.kCasterMaxScale = 3; // Max amount a caster will be scaled

this.kVerySmall = 0.001; //

this.kDistanceFudge = 0.01; // Dist between caster geometry and receiver: ensure no overlap

this.kReceiverDistanceFudge = 0.6; // Factor to reduce the projected caster geometry size

}

setShadowColor(c) {

this.mShadowColor = c;

}

... implementation to follow …

}

export default ShadowCaster;

The mShadowCaster is a reference to the shadow caster GameObject with at least a LightRenderable, and the mShadowReceiver is a GameObject with at least a SpriteRenderable render component. As will be detailed in the next step, mCasterShader, mShadowColor, and mSaveXform are variables to support the rendering of shadow caster geometries.

1. 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.

draw(aCamera) {

let casterRenderable = this.mShadowCaster.getRenderable();

// Step A: save caster xform, shader, and color. and, sets caster to shadow color

this.mShadowCaster.getXform().cloneTo(this.mSaveXform);

let s = casterRenderable.swapShader(this.mCasterShader);

let c = casterRenderable.getColor();

casterRenderable.setColor(this.mShadowColor);

let l, lgt;

// Step B: loop through each light in this array, if shadow casting on the light is on

// compute the proper shadow offset

for (l = 0; l < casterRenderable.getNumLights(); l++) {

lgt = casterRenderable.getLightAt(l);

if (lgt.isLightOn() && lgt.isLightCastShadow()) {

// Step C: turn caster into caster geometry, draws as SpriteRenderable

this.mSaveXform.cloneTo(this.mShadowCaster.getXform());

if (this.\_computeShadowGeometry(lgt)) {

this.mCasterShader.setCameraAndLights(aCamera, lgt);

SpriteRenderable.prototype.draw.call(casterRenderable, aCamera);

}

}

}

// Step D: restore the original shadow caster

this.mSaveXform.cloneTo(this.mShadowCaster.getXform());

casterRenderable.swapShader(s);

casterRenderable.setColor(c);

}

The casterRenderable is the Renderable object that is actually casting the shadow. The following are the four main steps of the draw() function.

1. Step A, saves the caster Renderable state: transform, shader, and color; and sets it into a shadow caster geometry by setting its shader to a ShadowCasterShader (mCasterShader), and its color to that of shadow color.
2. Step B, iterates through all light sources and look for lights that are switched on and casting shadow.
3. Step C, for each shadow producing light, calls the \_computeShadowGeometry() function to compute an appropriately size and positioned shadow caster geometry; and renders it as a SpriteRenderable. With the replaced ShadowCasterShader and shadow color, the rendered geometry appears as the shadow of the actual caster Renderable.
4. Step D, restores the state of the caster Renderable.
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 fourth and final region sets the computed parameters to the transform of the caster geometry, cxf.

\_computeShadowGeometry(aLight) {

// Region 1: declaring variables

let cxf = this.mShadowCaster.getXform();

let rxf = this.mShadowReceiver.getXform();

// vector from light to caster

let lgtToCaster = vec3.create();

let lgtToReceiverZ;

let receiverToCasterZ;

let distToCaster, distToReceiver; // measured along the lgtToCaster vector

let scale;

let offset = vec3.fromValues(0, 0, 0);

receiverToCasterZ = rxf.getZPos() - cxf.getZPos();

if (aLight.getLightType() === eLightType.eDirectionalLight) {

// Region 2: Processing a 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]); // measured along lgtToCaster

scale = Math.abs(1 / lgtToCaster[2]);

} else {

// Region 3: Processing a point or spot light

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))) {

// alomst 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]); // measured along lgtToCaster

scale = (distToCaster + (distToReceiver \* this.kReceiverDistanceFudge)) / distToCaster;

}

vec3.scaleAndAdd(offset, cxf.get3DPosition(), lgtToCaster, distToReceiver + this.kDistanceFudge);

// Region 4: Setting casterRenderable xform

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;

}

The aLight parameter is the casting light source. The goals of this function are 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 8-29, for the size of the projected caster geometry there are two cases to consider. First, for a directional light source the projected size is a constant. Second, for a point or spotlight the projected size is a function of distance to the receiver. These are the two cases, Regions 2 and 3 of the if statement with the following details.



Figure 8-29. Computing the shadow caster geometry

1. Region 2: Computes parallel projection according to the directional light. The if statement within this region 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.
2. Region 3: Computes projection from the point or spotlight position. The two if statements within this region are to ensure the shadow caster and receiver are on the same side of the light position and that, for the purpose of maintaining mathematical stability, neither is very close to the light source.
3. Region 4: Uses the computed distToReceiver and scale to set the transform of the shadow caster, or cxf.

The ShadowCaster object is meant for game developers to define and work with shadow. So, remember to update the engine access file, index.js, to forward the newly defined functionality to the client.

#### Defining the Shadow Receiver Class

Recall from Figure 8-25, ShadowReceiver is the object where the shadow of the caster object will appear. As illustrated in Figure 8-26, the ShadowReceiver must draw itself into the stencil buffer to ensure shadow caster geometry will only appear over pixels that are occupied by the ShadowReceiver object.

1. Create a new file, shadow\_receiver.js, in the src/engine/shadows folder, define the ShadowReceiver class. In the constructor, 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.

import \* as shaderResources from "../core/shader\_resources.js";

import ShadowCaster from "./shadow\_caster.js";

import \* as glSys from "../core/gl.js";

class ShadowReceiver {

constructor(theReceiverObject) {

this.kShadowStencilBit = 0x01; // The stencil bit to switch on/off for shadow

this.kShadowStencilMask = 0xFF; // The stencil mask

this.mReceiverShader = shaderResources.getShadowReceiverShader();

this.mReceiver = theReceiverObject;

// To support shadow drawing

this.mShadowCaster = []; // array of ShadowCasters

}

... implementation to follow …

}

export default ShadowReceiver;

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

addShadowCaster(lgtRenderable) {

let c = new ShadowCaster(lgtRenderable, this.mReceiver);

this.mShadowCaster.push(c);

}

// for now, cannot remove shadow casters

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

draw(aCamera) {

let c;

// Step A: draw receiver as a regular renderable

this.mReceiver.draw(aCamera);

// Step B: draw the receiver into the stencil buffer to enable corresponding pixels

glSys.beginDrawToStencil(this.kShadowStencilBit, this.kShadowStencilMask);

// Step B1: swap receiver shader to a ShadowReceiverShader

let s = this.mReceiver.getRenderable().swapShader(this.mReceiverShader);  
 // Step B2: draw the receiver again to the stencil buffer

this.mReceiver.draw(aCamera);

this.mReceiver.getRenderable().swapShader(s);

glSys.endDrawToStencil(this.kShadowStencilBit, this.kShadowStencilMask);

// Step C: draw shadow color to the pixels with stencil pixels switched on

for (c = 0; c < this.mShadowCaster.length; c++) {

this.mShadowCaster[c].draw(aCamera);

}

// switch off stencil checking

glSys.disableDrawToStencil();

}

This function 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 B enables the stencil buffer for drawing 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, in step C the draw() function calls to the mShadowCaster will draw the corresponding shadow caster geometries only into the pixels that are covered by the receiver.

Lastly, once again the ShadowReceiver object is designed for the client game developers to create shadows. So, remember to update the engine access file, index.js, to forward the newly defined functionality to the client.

### Updating Engine Support

With the new objects defined and engine configured, some of the existing engine classes must be modified to support the new shadow operations. The following summarizes the required changes without listing the straightforward changes. Please refer to the source code file for the actual implementation details.

* renderable.js: both the ShadowCaster and ShadowReceiver objects require the ability to swap the shaders to render the objects for shadow simulation purpose. This swapShader() function is best realized in the root of the Renderable hierarchy.
* light.js: the Light source now defines mCastShadow--a boolean variable and the associated getter and setter, indicating if the light should cast shadow.
* camera\_main.js: the Camera WC center must now be located at some z distance away. A kCameraZ constant is defined for this purpose and used in the mCameraMatrix computation in the setViewAndCameraMatrix() function.
* transform.js: the Transform class must be modified to support being cloneTo(), and the manipulation of a z-depth value.

### 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 all ShadowCaster objects and then create and add to the ShadowReceiver objects. The my\_game\_shadow.js file defines the \_setupShadow() function to demonstrate this.

MyGame.prototype.\_setupShadow = function () {

// mLgtMinion has a LightRenderable

this.mLgtMinionShadow = new engine.ShadowReceiver(this.mLgtMinion);

this.mLgtMinionShadow.addShadowCaster(this.mIllumHero);

this.mLgtMinionShadow.addShadowCaster(this.mLgtHero);

// mIllumMinion has a SpriteAnimateRenderable

this.mMinionShadow = new engine.ShadowReceiver(this.mIllumMinion);

this.mMinionShadow.addShadowCaster(this.mIllumHero);

this.mMinionShadow.addShadowCaster(this.mLgtHero);

this.mMinionShadow.addShadowCaster(this.mLgtMinion);

// mBg has a IllumRenderable

this.mBgShadow = new engine.ShadowReceiver(this.mBg);

this.mBgShadow.addShadowCaster(this.mLgtHero);

this.mBgShadow.addShadowCaster(this.mIllumMinion);

this.mBgShadow.addShadowCaster(this.mLgtMinion); }

The \_setupShadow() function is called at the end of the MyGame.init() function, when all other GameObject instances are properly created and initialized. This function demonstrates that different types of Renderable objects can serve as shadow receivers.

* LightRenderable: mLgtMinionShadow is created with mLgtMinon as a receiver, which has a reference to a LightRenderable object.
* IllumRenderable: mBgShadow and mMinionShadow are created with mBg and mIllumMinion being the receivers where both have references to IllumRenderable object.

Note that in order to observe shadow on an object, an explicit corresponding ShadowReceiver must be created and followed by explicitly adding ShadowCaster objects to the receiver. For example, mLgtMinionShadow defines the mLgtMinion object to be a receiver, where only the mIllumHero and mLgtHero will cast shadows on this object. Lastly, notice that mLgtMinon and mIllumMinion are both receivers and casters of shadows.

#### Drawing the Shadow

In 2D drawings, objects are drawn by overwriting 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 my\_game.draw() function in my\_game\_main.js, illustrates the important drawing order of objects.

draw() {

// Clear the canvas

engine.clearCanvas([0.9, 0.9, 0.9, 1.0]); // clear to light gray

// Set up the camera and draw

this.mCamera.setViewAndCameraMatrix();

// always draw shadow receivers first!

this.mBgShadow.draw(this.mCamera); // also draws the receiver object

this.mMinionShadow.draw(this.mCamera);

this.mLgtMinionShadow.draw(this.mCamera);

this.mBlock1.draw(this.mCamera);

this.mIllumHero.draw(this.mCamera);

this.mBlock2.draw(this.mCamera);

this.mLgtHero.draw(this.mCamera);

this.mMsg.draw(this.mCamera); // draw last

this.mMatMsg.draw(this.mCamera);

}

It is important to note the draw ordering. All three shadow receivers are drawn first. Additionally, among the three receivers the mBgShadow object is the actual background and thus is the first being drawn. Recall that in the definition of the ShadowReceiver class, the draw() function also draws the receiver object. For this reason, there is no need to call the draw() function of mLgtMinion, mIllumMinion, and mBg objects.

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 two hero objects. The mLgtHero on the right is illuminated by all four lights and thus casts many shadows. 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 shadow\_caster.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 to develop a variation of the simple yet complete Phong illumination model for the 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 critical 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 three chapters, this complementary pattern will be repeated. Chapter 9 will introduce physical behavior simulation, Chapter 10 will discuss particle effects, and Chapter 11 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 game loop 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. As you begin to define the setting you’ll begin exploring ideas for visual and audio design.

As is the case with most visual art, games rely in no small measure on effectively using 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.



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 programmatically generated or 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 www.playdead.com/limbo for more information).

Lighting is also often used as a core element of the game loop in addition to setting the mood; a game where the player is perhaps navigating in the dark with a virtual flashlight is an obvious 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 either safe areas or areas with deadly gas, flashing lights on a map can help direct players to important locations, and the like.

In the Simple Global Ambient project, you saw the impact that colored environment lighting has on the game setting. In that project the hero character moves in front of a background of metallic panels, tubes, and machinery, perhaps the exterior of a space ship. The environment light is red and can be pulsed --notice the effect on mood when the intensity is set to 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 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.

Before you begin thinking about how lighting and other design elements might enhance the game setting and visual style, let us 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.

For the next phase of the simple game mechanic project, how might you integrate light directly into the game loop 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 game loop at a time will help prevent the design from becoming over-burdened 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.



Figure 8-31. The addition of a movable “flashlight” that shines a special beam

In the first iteration of this game loop 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 will deduce the proper sequence required to unlock the barrier. Now imagine how the addition of a special light beam might take the gameplay in a new direction: building on the basic notion of sequencing you can create an increasingly clever 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.



Figure 8-32. The player is able to directly activate the objects as in the first iteration of the mechanic, but the corresponding section of the lock now remains inactive.



Figure 8-33. The player moves the flashlight under one of the shapes to reveal a hidden clue (#1).

From the gameplay point of view any object in a game environment can be used as a tool; your job as a designer is to ensure 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 game loop in our sample project 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 only 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).



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.

Note that you’ve changed the feedback players receive slightly from the first iteration of the game loop: 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 loop 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: 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.

It’s important to be aware that as gameplay complexity increases so also 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 or a mobile device using only touch. Development teams often pour thousands of hours of research and testing into control schemes, yet they still frequently miss the mark; interaction design is tricky to get right and often requires thousands of hours of testing and refinement even for fairly basic actions, so when possible you should make use of established conventions. 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 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 what makes an experience fun. 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 controlling multiple objects. The most common pattern for our flashlight would likely be for the player to “equip” it; perhaps if the player moves over the flashlight and clicks the left mouse button it becomes a new ability for the player that can be activated by pressing one of the keyboard keys or by clicking the right mouse button. Alternately, perhaps the hero character can move around the game screen freely with the WASD keys while other active objects like the flashlight are first selected with a left mouse click and then moved by holding the left mouse button and dragging them into position. There are similarly a variety of ways to provide the player with contextual feedback that will help teach the puzzle logic and rules (in this case we’re using the ring around the lock as 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 game loop can evolve once again by applying simple physics to objects in the game world.