Implementing Illumination and Shadow

After completing this chapter, you will be able to:

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

# Introduction

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

An illumination model, or a lighting model, is a mathematic formulation that describes the color and brightness of a scene based on simulating light energy reflecting off the surfaces in the scene. In this chapter, you will implement an illumination model that indirectly affects the types of gameplay your game engine can support and the visual fidelity that can be achieved. This is because illumination within a game engine can be more than a simple aesthetic effect. When used creatively, illumination can enhance gameplay or provide a dramatic setting for your game. For example, you could have a scene with a torch light that illuminates an otherwise dark pathway for the hero, with the torch flickering to communicate a sense of unease or danger to the player. Additionally, while the lighting model is based on light behaviors within the real world, in your game implementation the lighting model allows surreal or physically impossible settings, such as an oversaturated light source that displays bright or iridescent colors or even a negative light intensity that seemingly absorbs the light around it.

When implementing illumination models commonly present in game engines, you will need to venture into concepts in 3D space to properly simulate light within your scenes. You will need to define depth values for the light sources to cast light energy upon the game objects, or renderables, which are flat 2D geometries. Once you consider concepts in 3D, the task of implementing a lighting model becomes much more straightforward, and you can apply knowledge from computer graphics to properly illuminate a scene.

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

# Overview of Illumination and GLSL Implementation

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

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

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

# Ambient Light

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

## The Global Ambient Project

This project demonstrates how to implement ambient lighting within your scenes by providing a global ambient color and a global ambient intensity that each renderable object references before being drawn. You can see an example of this project running in Figure 8-1. The source code of this project is located in the chapter8/8.1.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 for the game engine is the GLSL shader. This is because it allows you to implement the shading technique, which in turn provides the outline for how your engine must be modified in order to support this new shader. Thus, to start, implement the global ambient into your simple\_fs.glsl.

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

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. Then utilize them by multiplying them by c (the fragment color sampled from the texture) to get the final color for each fragment. Remember that the color for c was obtained by using the interpolated vTexCoord variable from the vertex shader to sample a fragment from the passed-in texture. You can see this implemented in the following code:

precision mediump float;

uniform sampler2D uSampler;

// Color of pixel

uniform vec4 uPixelColor;

uniform vec4 uGlobalAmbientColor; // this is shared globally

uniform float uGlobalAmbientIntensity; // this is shared globally

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

// interpolated and thus varies.

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;

// different options:

// e.g. tint the transparent area also

// vec4 result = c \* (1.0-uPixelColor.a) + uPixelColor \* uPixelColor.a;

// or: tint the textured area, and leave transparent area as defined by the texture

vec3 r = vec3(c) \* (1.0-uPixelColor.a) + vec3(uPixelColor) \* uPixelColor.a;

vec4 result = vec4(r, c.a);

// or: ignore pixel tinting ...

// vec4 result = c;

// or: simply multiply pixel color with texture color

// vec4 result = c \* uPixelColor;

gl\_FragColor = result;

}

### Modifying SimpleShader

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

1. TEMP TEXT

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

1. Modify the simple\_shader.js file in the src/engine/shaders folder to hold two new variables in the constructor for storing the references or locations of the ambient color and intensity variables within the GLSL shader.

this.mGlobalAmbientColorRef = null;

this.mGlobalAmbientIntensityRef = null;

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

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

activate(pixelColor, trsMatrix, cameraMatrix) {

let gl = glSys.get();

gl.useProgram(this.mCompiledShader);

// bind vertex buffer

gl.bindBuffer(gl.ARRAY\_BUFFER, vertexBuffer.get());

gl.vertexAttribPointer(this.mVertexPositionRef,

3, // each element is a 3-float (x,y.z)

gl.FLOAT, // data type is FLOAT

false, // if the content is normalized vectors

0, // number of bytes to skip in between elements

0); // offsets to the first element

gl.enableVertexAttribArray(this.mVertexPositionRef);

// load uniforms

gl.uniform4fv(this.mPixelColorRef, pixelColor);

gl.uniformMatrix4fv(this.mModelMatrixRef, false, trsMatrix);

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

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

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

}

### Modifying the Engine

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

1. Begin by adding a global ambient color and a global ambient intensity variable in the default\_resources.js file, located in the src/engine/resources folder, as shown here:

// Global Ambient color

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

let mGlobalAmbientIntensity = 1;

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

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]); }

1. TEMP TEXT

export {

init, cleanUp,

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

getDefaultFontName,

// Global ambient: intensity and color

getGlobalAmbientColor, setGlobalAmbientColor,

getGlobalAmbientIntensity, setGlobalAmbientIntensity

}

### Testing the Ambient Illumination

Now that the engine supports ambient lighting for a scene, all that is left is to verify the correctness by utilizing it within my\_game.js and observe the shaded results. To get a better picture of what exactly is happening and how exactly the ambient lighting can be utilized, you can begin with a clean my\_game.js file and re-implement the core functions.

1. TEMP TEXT

"use strict"; // Operate in Strict mode such that variables must be declared before used!

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. Modify my\_game\_main.js by creating the MyGame class which extends the Scene class. Define the constructor by adding variables to the for a camera, a background, a hero, and some minions. Additionally, remember to import and export the need functionalities.

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

load() {

engine.texture.load(this.kMinionSprite);

engine.texture.load(this.kBg);

}

unload() {

engine.texture.unload(this.kMinionSprite);

engine.texture.unload(this.kBg);

}

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

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);

// Step B: Create the hero object with texture from the lower-left corner

this.mHero = new Hero(this.kMinionSprite);

this.mLMinion = new Minion(this.kMinionSprite, 30, 30);

this.mRMinion = new Minion(this.kMinionSprite, 70, 30);

this.mMsg = new engine.FontRenderable("Status Message");

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

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

this.mMsg.setTextHeight(3);

}

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

\_drawCamera(camera) {

// set up the View Projection matrix

camera.setViewAndCameraMatrix();

// Now draws each primitive

this.mBg.draw(camera);

this.mHero.draw(camera);

this.mLMinion.draw(camera);

this.mRMinion.draw(camera);

}

draw() {

// Step A: clear the canvas

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

// Step B: Draw with all three cameras

this.\_drawCamera(this.mCamera);

this.mMsg.draw(this.mCamera); // only draw status in the main camera

}

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

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

# Light Source

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

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



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

## GLSL Implementation and Integration into the Game Engine

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

From your experience building the game engine, you will remember that the engine interfaces to the GLSL shaders with the corresponding subclasses of the Shader/Renderable object pairs: Shader objects to interface to the GLSL shaders and Renderable objects to provide programmers with the convenience of manipulating many copies of geometries of the same shader type. For example, 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.

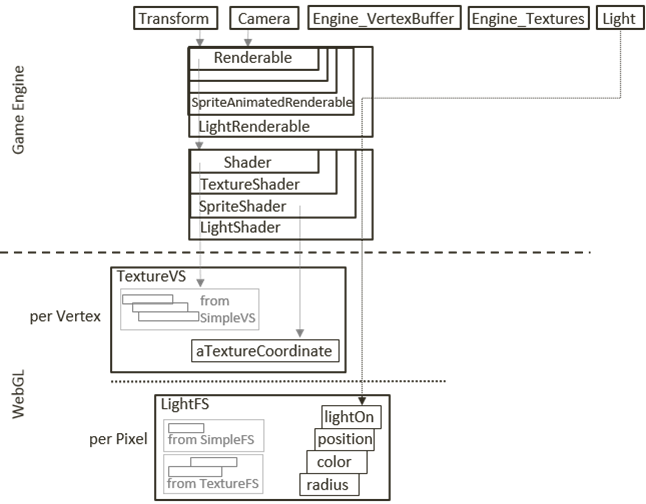


Figure 8-3. LightShader/LightRenderable pair and the corresponding GLSL LightShader

Finally, before you begin learning about the elements of a Phong Illumination model, it is important to point out again that the GLSL fragment shader is invoked once for every pixel covered by the corresponding geometry. This means the GLSL fragment shaders you are about to learn will be invoked many times per frame, probably in the range of hundreds of thousands or even millions. Considering the fact that the game loop initiates redrawing at a real-time rate, or around 60 frame redraws per second, the GLSL fragment shaders will be invoked many millions of times per second! The efficiency of the implementation is of most importance!

## The Simple Light Shader Project

This project demonstrates how to implement a simple point light and illuminate objects within the scene. You can see an example of this project running in Figure 8-4. The source code of this project is located in the chapter8/8.2.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 experience illumination results from a point light
* To implement a GLSL shader that supports point light illumination

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

### Creating the GLSL Light Fragment Shader

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

1. Under the src/glsl\_shaders folder, create a new file and name it light\_fs.glsl.
2. Add the standard uniform and varying variables for the texture sampler, texture coordinate, ambient properties, and pixel color as in previous projects. Furthermore, you can now add support for a single light by adding variables for each of the light’s properties. It is also important to notice that the light’s position and radius are in pixel space, or Device Coordinate (DC) space, to facilitate illumination computations.

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;

// Light information

uniform bool uLightOn;

uniform vec4 uLightColor;

uniform vec3 uLightPosition; // in pixel space!

uniform float uLightRadius; // in pixel space!

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

// interpolated and thus varies.

varying vec2 vTexCoord;

1. To complete the shader, implement the main() function to do the following:
2. Sample the texture color and apply the ambient color and intensity.
3. Determine whether the current fragment should be illuminated by the light source. To do this, first check whether the light is on; if it is, compute the distance between the light’s position (in pixel space) and the current fragment’s position (in pixel space) that is defined in the GLSL-provided variable gl\_FragCord.xyz. If the distance is less than that of the light’s radius (again in pixel space), then accumulate the light’s color.
4. The last step is to apply the tint and to set the final color via gl\_FragColor.

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

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

if (dist <= uLightRadius)

lgtResults += uLightColor;

}

lgtResults \*= textureMapColor;

// tint the textured area, and leave transparent area as defined by the texture

vec3 r = vec3(lgtResults) \* (1.0-uPixelColor.a) + vec3(uPixelColor) \* uPixelColor.a;

vec4 result = vec4(r, textureMapColor.a);

gl\_FragColor = result;

}

### Creating a Light Object

With the GLSL light\_fs shader defined; you can now create an object to encapsulate a point light source.

1. Create a new folder called lights under the src/engine folder. In the lights folder, add a new file and name it lights.js.
2. In lights.js, create a new class called Light and add a simple constructor that initializes a color, position, radius, and on-off variable for the light. Set their initial values as follows:

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

// 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; }

### Creating the LightShader Object

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

1. Under the src/engine/shaders folder, create a new file and name it light\_shader.js.
2. Now create a class called LightShader and add a constructor in order to initialize the references for the light’s variables and to obtain their reference locations within the shader. Remember to inherit from the SpriteShader object.

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

setCameraAndLight(c, l) {

this.mCamera = c;

this.mLight = l;

}

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

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

\_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);

}

}

### Creating the LightRendererable Object

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

1. Begin by creating a new file in the src/engine/renderables folder and naming it light\_rendererable.js.
2. Next, create a new class called LightRenderable which extends SpriteAnimateRenderable and add a constructor to call the superclass’s constructor to set the corresponding light shader and to create a variable for the light that will illuminate this object.

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. Add a draw function that passes the illuminating light source to the LightShader object for communicating with the GLSL fragment shader.

draw(camera) {

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

super.draw(camera);

}

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

getLight() {

return this.mLight;

}

addLight(l) {

this.mLight = l;

}

### Defining a Default LightShader Instance

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

1. Begin by adding a variable for the light shader in the shader\_resources.js file located in the src/engine/core/resources folder. Also, define an accessor function as shown here:

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

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

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. In the cleanUp() function, add the following lines of code to unload the file when it is no longer needed:

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. TEMP TEXT

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 (for example, origin of the Camera) that do not change during each rendering invocation. To avoid repeated computation of these values, a per-render invocation cache should be defined for the Camera object.

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

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

1. TEMP TEXT

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

export default Camera;

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

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

constructor(wcCenter, wcWidth, viewportArray, bound) {

…

// background color

this.mBGColor = [0.8, 0.8, 0.8, 1]; // RGB and Alpha

// 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 setupViewProjection() function by adding step B4 to calculate and set the cache using the existing Camera viewport width, world width, and world height.

setViewAndCameraMatrix() {

let gl = glSys.get();

…

// 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);

}

#### Adding Camera Transform Functions

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

1. Under the src/engine/cameras folder, create a new file/module and name it camera\_xform.js.

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

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

... implementation to follow …

export default Camera;

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

Camera.prototype.wcPosToPixel = function (p) { // p is a vec3, fake Z

// Convert the position to pixel space

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, provide 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;

}

### Modifiy index.js stuff

### Testing the Light

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

#### Modifying the Hero and Minion

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

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

constructor(spriteTexture) {

super(null);

this.kDelta = 0.3;

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

…

}

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

constructor(spriteTexture, atX, atY) {

super(null);

this.kDelta = 0.2;

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

…

}

#### Modifying the MyGame Object

With the implementation of the light completed and the game objects properly updated, you can now modify the MyGame level to display and test the light source. Because of the simplistic and repetitive nature of the code in the 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 the illuminated circle decreases (smaller intersection area) and increases in size. Alternatively, you can press the C or V key to increase or decrease the point light radius to increase or decrease the volume of illumination, and observe the corresponding changes in the illuminated circle radius. Another observation to take note of is that the light source illuminates the left minion, the hero, and the background but not the other three objects in the scene. This is because the right minion and the two blocks are not LightRenderable objects and thus cannot be illuminated by the defined light source.

# Multiple Light Sources and Distance Attenuation

In the previous project, a single point light source was defined with the capability of illuminating a spherical volume. This type of light source is useful in many games, but it is restrictive to limit a game to only a single light source. The engine should support the illumination from multiple light sources to fulfill the design needs of different games. This shortcoming is remedied in the next project with general support for multiple light sources. The implementation principle for multiple lights remains the same as the previous project, with the modification of replacing the single light source with an array of lights. As illustrated in Figure 8-5, a new Light object will be defined, while the LightRenderable object will be modified to support an array of the Light objects. The LightShader object will define an array of ShaderLightAtindex objects that are capable of communicating light source information to the uLights array in the GLSL 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 the illuminated circle disappears abruptly with a sharp illuminated bright boundary. This sudden disappearance of illumination results does not reflect real life where effects from a given light source decrease gradually over distance instead of switching off abruptly. A more visually pleasing light illumination result should show an illuminated circle where the illumination results at the boundary disappear gradually. This gradual decrease of light illumination effect over distance is referred to as distance attenuation. It is a common practice to approximate distant attenuation with quadratic functions because they produce effects that resemble the real world. In general, distance attenuation can be approximated in many ways, and it is often refined to suit the needs of the game. In addition to distance attenuation, you will also implement a near cutoff distance and a far cutoff distance (that is, two distances from the light source at which the distant attenuation effect will begin and end). These two values give you control over a light source to show a fully illuminated center area with illumination drop-off occurring only at a specified distance. Lastly, a light intensity will be defined to allow dramatically different effects. For example, you can have a soft, barely noticeable light that covers a wide area or an oversaturated glowing light that is concentrated over a small area in the scene.

## The Multiple Lights Project

This project demonstrates how to implement multiple point lights within a single scene. It also demonstrates how to increase the complexity of your point lights so that they are more flexible to serve a wider variety of purposes. You can see an example of this project running in Figure 8-6. The source code of this project is located in the chapter8/8.3.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 in GLSL shaders
* To understand and examine the distance attenuation effects of light
* To experience controlling and manipulating multiple light sources in a scene

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

### Modifying the GLSL Light Fragment Shader

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

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

**Note** You can define as many lights as the hardware can support. For example, you can try increasing the number of lights to 50 and then test and measure its performance.

1. Next add a function called LightEffect() that takes a light parameter and returns a color as a result. This function calculates the distance between the light and the current fragment and determines whether it lies within the near radius, in between near and far radii, or farther than the far radius. If the fragment’s current position lies within the near radius, there is no attenuation, so a value of 1 is applied. If the fragment’s current position lies in between the near and far radii, then a quadratic attenuation is applied. A distance of greater than the far radius will result in no illumination from the corresponding light source, or a 0 value will be applied. You can see how this is achieved in the following code:
2. The main function iterates through all the defined light sources and calls the LightEffect() function to calculate and accumulate the contribution from the corresponding light in the array.

### Modifying the Light Object

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

1. Modify the Lights.js constructor to include variables for near and far attenuation and intensity. You can see this achieved in the following code:
2. Define the get and set accessors for the new variables. Note that the radius variable has been generalized and replaced by the near and far cutoff distances.

### Creating the LightSet Object

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

### Creating the ShaderLightAtIndex Object

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

1. Under the src/Engine/Shaders folder, create a new file and name it ShaderLightAtIndex.js. Remember to load this new source file in index.html.
2. Define a constructor to set the light property references to a specific index in the uLights array in the fragment shader.
3. Implement the loadToShader() function to push the light’s properties to the GLSL shader. Notice that this function is similar to the previous loadToShader() function defined in the LightShader.js file.
4. Provide a function to update the shader’s on/off property for the light.

### Modifying the LightShader Object

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

1. Begin by editing the LightShader.js file and removing the loadToShader() function because the actual loading of light information to GLSL shaders will be handled by the newly defined ShaderLightAtIndex objects.
2. Modify the constructor to define mLights, which is an array of ShaderLightAtIndex objects to correspond to the uLights array defined in the LightFS fragment shader. It is important to note that the mLights and uLights arrays must be the exact same size. You can see this in the following code:
3. Modify the activateShader() function to iterate and load the contents of each ShaderLightAtIndex object to the LightFS shader by calling the corresponding loadToShader() function. Recall that the GLSL fragment shader requires the for loop control variable to be a constant. This implies that all elements of the uLights array will be processed on each LightFS invocation, and thus it is important to ensure all unused lights are switched off. This is ensured by the last while loop in the following code:
4. Make a simple modification to the setLight() function so that it becomes setLights() and handles the array rather than the single light.

### Modifying the LightRenderable Object

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

1. In the LightRenderable constructor, replace the single light reference variable with an array.
2. Make sure to update the draw function to reflect the change to multiple light sources.
3. Define the corresponding accessor functions for the light array.

### Modifiy index.js stuff

### Testing the Light Sources with MyGame

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

1. Modify the MyGame.js file in the src/MyGame folder to reflect the changes to the constructor, initialize function, draw function, and update function. All these changes revolve around handling multiple lights through a light set.
2. In the src/MyGame folder, create the new file MyGame\_Lights.js to instantiate and initialize the lights. Remember to load this new source file in index.html.
3. In the src/MyGame folder, create the new file MyGame\_lightControls.js to implement the controls of the lights. Remember to load this new source file in index.html.

### Observations

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

# Diffuse Reflection and Normal Mapping

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

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



Figure 8-7. Surface normal vectors of an object

A human’s observation of light illumination is the result of visible energy from the light sources reflecting off object surfaces and reaching the eyes. A diffuse, or Lambertian, surface reflects light energy uniformly in all directions. Examples of diffuse surfaces include typical printer papers or matte painted surfaces. Figure 8-8 shows a light source illuminating three diffuse surface element positions, A, B, and C. First, notice that the direction from the position being illuminated toward the light source is defined as the position’s light vector, . It is important to note that the direction of the vector is always toward the light source and that this is a normalized vector with a magnitude of 1. Figure 8-8 illustrates the diffuse illumination, or magnitude of diffuse reflection, with examples. Position A cannot receive any energy from the given light source because its normal vector,, 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 light vector, or .



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

Displaying the computation results, or the diffuse component, cues 3D shape contours for the human vision system. For example, Figure 8-9 shows a sphere and torus (doughnut shape object) with (the left image) and without (the right image) the corresponding diffuse components. Clearly, in both cases, the 3D contour of the objects is captured by the left versions of the image with the computation.



Figure 8-9. 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: the vector in the z direction. This lack of distinct normal vectors for objects implies that it is not possible to compute the diffuse component of objects. Fortunately, similar to how texture mapping addresses the limitation of each geometry having only a single color, normal mapping can resolve the problem of each geometry having only a single normal vector. Figure 8-10 shows the idea behind normal mapping where in addition to the color texture image a corresponding normal texture image is required. The left image of Figure 8-10 is a typical color texture image, and the two right images are zoomed images of the highlighted square on the left image. Notice once again that two images are involved in normal mapping: the color texture image where the RGB channels of the texture record the color of objects (bottom of the right image of Figure 8-10) and a corresponding normal texture image where the RGB channels record the x, y, and z values of the normal vector for the corresponding object in the color texture (top of the right image).



Figure 8-10. Normal mapping with two texture images: the normal and the color texture

Figure 8-11 captures the view of the three corresponding positions labeled on the right images of Figure 8-10 (the positions 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 8-11 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 8-11 shows that the RGB components of the normal texture records the normal vector xyz values of objects at the corresponding color texture positions. The top layer of Figure 8-11 shows that when illuminated by a light source with the term properly computed and displayed, the human vision system will perceive a sloped contour.



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

In summary, a normal texture map or a normal map is a texture map that stores normal vector information rather than the usual color information. Each texel of a normal map encodes the xyz values of a normal vector in the RGB channels. In lieu of displaying the normal map texels as you would with a color texture, the texels are used purely for calculating how the surface would interact with light. In this way, instead of a constant normal vector pointing in the z direction, when a square is normal mapped, the normal vector of each pixel being rendered will be defined by texels from the normal map and used for computing the diffuse component. When computing light illumination, the computation is referred to as *diffuse* computation, lighting, or illumination. The mathematic term is referred to as the diffuse term.

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

## The Normal Maps and Illumination Shaders Project

This project demonstrates how to integrate normal mapping into your game engine and use the results to compute the diffuse component of objects. You can see an example of this project running in Figure 8-12. The source code of this project is located in the chapter8/8.4.normal\_maps\_and\_illumination\_shaders folder.



Figure 8-12. Running the Normal Maps and Illumination Shaders 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: Increases/decreases the light z position
* C/V and B/N keys: Increases/decreases the near and far cutoff distances of the selected light
* K/L key: Increases/decreases the intensity of the selected light
* H key: Toggles the selected light on/off

The goals of the project are as follows:

* To understand and work with normal maps
* To implement normal maps as textures in the game engine
* To implement GLSL shaders that support diffuse component illumination
* To examine the diffuse component illumination of

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.

### Creating the GLSL Illumination Fragment Shader

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

1. Begin by copying and pasting LightFS.glsl and naming the new file IllumFS.glsl within the src/GLSLShaders folder.
2. Edit the IllumFS.glsl file and add a sampler2D object, uNormalSampler, to sample the normal map.
3. Modify the LightEffect() function to receive a normal vector parameter, N. This normal vector N is assumed to be normalized with a magnitude of 1 and will be used in the diffuse component computation. Include code to compute the vector, remember to normalize the vector, and use the result of to scale the light attenuation accordingly as follows:
4. Edit the main function to sample from both the color texture with uSampler and the normal texture with uNormalSampler. Remember that the normal map provides you with a vector that represents the normal vector direction of the surface element at that given point. Because the xyz normal vector values are stored in the 0 to 1 RGB color format, the sampled normal map results must be scaled and offset to the -1 to 1 range. In addition, recall that texture uv coordinates can be defined with the v direction increasing upward or downward. In this case, depending on the v direction of the normal map, you may also have to flip the y direction of the sampled normal map values. The normalized normal vector, N, is then passed on to the LightEffect() function for the illumination calculations.

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

### Creating the IllumShader Object

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

1. Under the src/Engine/Shaders folder, create a new file and name it IllumShader.js. Remember to load this new source file in index.html.
2. Create the constructor by first subclassing from the LightShader to take advantage of the functionality related to light sources and define a variable, mNormalSamplerRef, to maintain a reference to the normal sampler in the GLSL shader.
3. Override and extend the activateShader() function by binding the normal texture sampler reference to WebGL texture unit 1. So far, you have been working with the color texture sampler that is bounded to the default texture unit of 0. In this way, the WebGL texture system can work with two active textures: units 0 and 1. As will be discussed, the TextureShader object must now explicitly bind the color texture to unit 0, and in gEngine.Textures, it is important to configure the WebGL to activate the appropriate texture units for the corresponding purpose: color versus normal texture mapping.

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

### Modifying the TextureShader Object

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

1. Modify the constructor to include and initialize a reference to the sampler location, as shown in the following code:
2. Add a line to the activateShader() function to bind the texture to unit 0, as shown in the following code:

### Creating the IllumRenderable Object

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

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

### Modifying the Engine

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

#### Defining the Default in Engine\_DefaultResources

You must modify the Engine\_DefaultResources.js file to define the default instance of the IllumShader object.

1. Define a constant file path and variable for the newly created fragment shader.
2. Modify the \_createShaders() function to instantiate an IllumShader object.
3. Add a simple accessor for the illumination shader.
4. Modify the initialize() function to load the text file that defines the illumination shader.
5. Modify the cleanUp() function to unload the text file that defines the illumination shader.
6. Export the get accessor in the public function list.

#### Configuring WebGL Texture Units in Engine\_Textures

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

1. Add a line to the activateTexture() function to specify the texture unit 0. Recall that this function activates the color texture mapping functionality. The gl.TEXTURE0 constant informs WebGL to bind to the texture unit 0.
2. Add a function to activate the normal map texture and bind this texture to WebGL texture unit 1 with the TEXTURE1 constant.
3. Remember to export the activation of the normal map in the public function list.

### Testing the Normal Map

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

#### Modifying the Hero and the Minion

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

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

#### Modifying MyGame

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

### Observations

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

# Specular Reflection and Materials

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



Figure 8-13. Specularity: the reflection of the light source

From real-life experience, you know that specular highlights are visible even when the eye’s viewing direction is not perfectly aligned with the reflection direction of the light source. As illustrated in Figure 8-14, where the 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 , the less likely you will observe the light reflection. In fact, you know that when α, the angle between and , 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 8-14. The Phong specularity model

The Phone illumination model simulates the characteristic 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 will roll off. 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 8-15, the right sphere has a value of 0, the middle sphere has a value of 5, and the right sphere’s value is 30.



Figure 8-15. Specularity and shininess (n)

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 8-16, is the halfway vector, which is the vector halfway between 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 8-16. The halfway vector

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

* The ambient term:
* 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 IllumFS 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 IllumFS GLSL shader with the engine support in the IllumShader/IllumRenderable object pair.



Figure 8-17. The Phone illumination model

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

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



Figure 8-18. Support for material

## The Material and Specularity Project

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



Figure 8-19. Running the Material and Specularity 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
* Z/X key: Increases/decreases the light z position
* C/V and B/N keys: Increases/decreases the near and far cutoff distances of the selected light
* K/L key: Increases/decreases the intensity of the selected light
* H key: Toggles the selected light on/off

Material property controls:

* *Number keys 5 and 6*: Select the left minion and the hero
* *Number keys 7, 8, and 9*: Select the : , , 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 shaders
* To understand and experience the control of Material of illuminated objects
* To examine specular highlights in illuminated images

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

### Modifying the GLSL Illumination Fragment Shader