CHAPTER 10

Supporting Camera Background

After completing this chapter, you will be able to:

- Implement background tiling with any image in any given camera WC bounds
- Understand parallax and simulate motion parallax with parallax scrolling
- Appreciate the need for layering objects in 2D games and support layered drawing

Introduction

By this point your game engine is capable of illuminating 2D images to generate highlights and shadows and of simulating basic physical behaviors. To complete the engine development, this chapter focuses on the general support for creating the game world environment with background tiling and parallax and relieving the game programmers from having to manage draw ordering. Background images or objects are included to decorate the game world to further engage the players; this often requires being vast in scale with subtle visual complexities. For example, in a side-scrolling game, the background must always be present, and simple motion parallax can create the sense of depth and further capture the players' interests.

Tiling, in the context of computer graphics and video games, refers to the duplication of an image or pattern along the x and y directions. In video games, images used for tiling are usually strategically constructed to ensure content continuation across the duplicating boundaries. Figure 10-1 shows an example of a strategically drawn background image tiled three times in the x direction and two times in the y direction. Notice the perfect continuation across the duplication boundaries. Proper tiling conveys a sense of complexity in a boundless game world by creating only a single image.

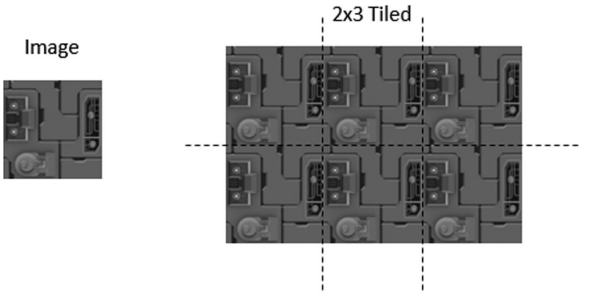

Figure 10-1. *Tiling of a strategically drawn background image*

Parallax is the apparent displacements of objects when the objects are viewed from different positions. Figure 10-2 shows an example of the parallax of a shaded circle. When viewed from the middle eye position, the center shaded circle appears to be covering the center rectangular block. However, this same shaded circle appears to be covering the top rectangular block when viewed from the bottom eye position. Motion parallax is the observation that when one is in motion, nearby objects move quicker than those in the distance. This is a fundamental visual cue that informs depth perception. In 2D games, the simulation of motion parallax is a straightforward approach to introduce depth complexity to further captivate the players.

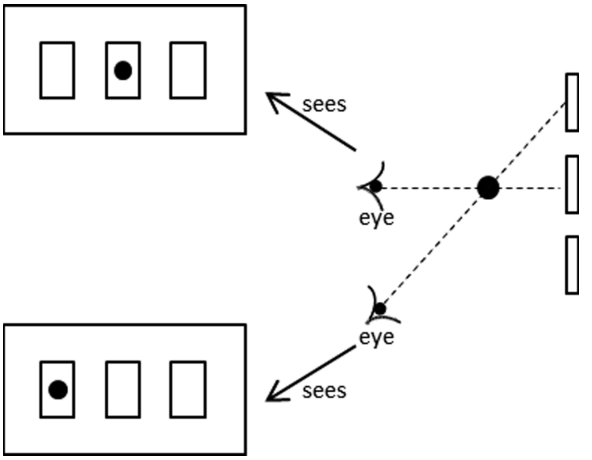

Figure 10-2. *Parallax: seeing different positions of objects from different viewpoints*

This chapter presents a general algorithm for tiling the camera WC bounds and describes an abstraction for hiding the details of parallax scrolling. With the increase in visual complexity of the background, this chapter discusses the importance of and creates a layer manager to alleviate game programmers from the details of draw ordering.

Tiling of the Background

When tiling the background in a 2D game, it is important to recognize that only the tiles result in covering the camera WC bounds need to be drawn. This is illustrated in Figure 10-3. In this example, the background object to be tiled is defined at the WC origin with its own width and height. However, in this case, the camera WC bounds do not intersect with the defined background object. Figure 10-3 shows that the background object needs to be tiled six times to cover the camera WC bounds. Notice that since it is not visible through the camera, the player-defined background object that located at the origin does not need to be drawn.

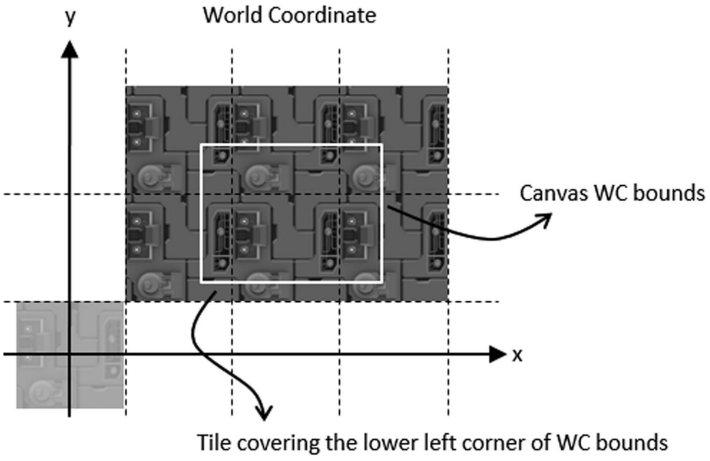

Figure 10-3. *Generating tiled background for camera WC bounds*

There are many ways to compute the required tiling for a given background object and the camera WC bounds. A simple approach is to determine the tile position that covers the lower-left corner of the WC bound and tile in the positive x and y directions.

The Tiled Objects Project

This project demonstrates how to implement simple background tiling. You can see an example of this project running in Figure 10-4. The source code to this project is defined in the `Chapter10/10.1.TiledObjects` folder.

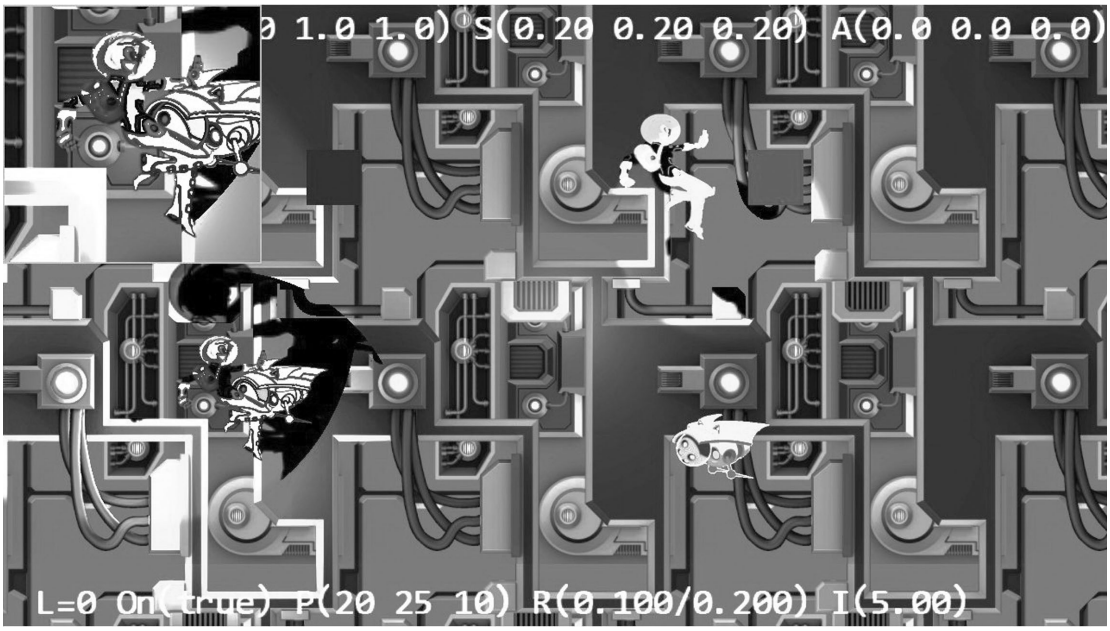

Figure 10-4. *Running the Tiled Objects project*

The controls of the project are as follows:

- *WASD keys*: Move the Dye character (the hero) to pan the WC window bounds

The goals of the project are as follows:

- To experience working with multiple layers of background
- To implement the tiling of background objects for camera WC window bounds

You can find the following external resources in the `assets` folder: the `fonts` folder that contains the default system fonts and six texture images (`minion_sprite.png`, `minion_sprite_normal.png`, `bg.png`, `bg_normal.png`, `byLayer.png`, and `bgLayer_normal.png`). The `Hero` and `Minion` objects are represented by sprite elements in the `minion_sprite.png` image, and `bg.png` and `bgLayer.png` are two layers of background images. The corresponding `_normal` files are the normal maps.

Define TiledGameObject

Recall that a `GameObject` abstracts the basic behavior of an object in the game where its appearance is determined by the `Renderable` object that it references. A `TiledGameObject` is a `GameObject` that is capable of tiling the referenced `Renderable` object to cover the WC bounds of a given `Camera` object.

1. Create a new file in the `src/Engine/GameObjects/` folder and name it `TiledGameObject.js`. Add the following code to construct the object:

```
function TiledGameObject(renderableObj) {
    this.mShouldTile = true;  // can switch this off if desired
    GameObject.call(this, renderableObj);
}
gEngine.Core.inheritPrototype(TiledGameObject, GameObject);
```

The `TiledGameObject` object has only one variable, `mShouldTile`, which is a Boolean that determines whether the object should tile.

2. Define the getter and setter functions for `mShouldTile`.

```
TiledGameObject.prototype.setIsTiled = function (t) {
    this.mShouldTile = t;
};

TiledGameObject.prototype.shouldTile = function () {
    return this.mShouldTile;
};
```

3. Define the function to tile and draw the `Renderable` object to cover the WC bounds of the `aCamera` object.

```
TiledGameObject.prototype._drawTile = function(aCamera) {
    // Step A: Compute the positions and dimensions of tiling object.
    var xf = this.getXform();
    var w = xf.getWidth();
    var h = xf.getHeight();
    var pos = xf.getPosition();
```

```
var left = pos[0] - (w/2);
var right = left + w;
var top = pos[1] + (h/2);
var bottom = top - h;

// Step B: Get the world positions and dimensions of the drawing camera.
var wcPos = aCamera.getWCCenter();
var wcLeft = wcPos[0] - (aCamera.getWCWidth() / 2);
var wcRight = wcLeft + aCamera.getWCWidth();
var wcBottom = wcPos[1] - (aCamera.getWCHeight() / 2);
var wcTop = wcBottom + aCamera.getWCHeight();

// Step C: Determine the offset to the camera window's lower left corner.
var dx = 0, dy = 0; // offset to the lower left corner
// left/right boundary?
if (right < wcLeft) { // left of WC left
    dx = Math.ceil((wcLeft - right)/w) * w;
} else {
    if (left > wcLeft) { // not touching the left side
        dx = -Math.ceil((left-wcLeft)/w) * w;
    }
}
// top/bottom boundary
if (top < wcBottom) { // Lower than the WC bottom
    dy = Math.ceil((wcBottom - top)/h) * h;
} else {
    if (bottom > wcBottom) {  // not touching the bottom
        dy = -Math.ceil((bottom - wcBottom)/h) * h;
    }
}

// Step D: Save the original position of the tiling object.
var sX = pos[0];
var sY = pos[1];

// Step E: Offset tiling object and modify the related position variables.
xf.incXPosBy(dx);
xf.incYPosBy(dy);
right = pos[0] + (w/2);
top = pos[1] + (h/2);

// Step F: Determine the number of times to tile in the x and y directions.
var nx = 1, ny = 1; // number of times to draw in the x and y directions
nx = Math.ceil((wcRight - right) / w);
ny = Math.ceil((wcTop - top) / h);

// Step G: Loop through each location to draw a tile.
var cx = nx;
var xPos = pos[0];
while (ny >= 0) {
    cx = nx;
```

```
            pos[0] = xPos;
            while (cx >= 0) {
                this.mRenderComponent.draw(aCamera);
                xf.incXPosBy(w);
                --cx;
            }
            xf.incYPosBy(h);
            --ny;
        }

        // Step H: Reset the tiling object to its original position.
        pos[0] = sX;
        pos[1] = sY;
    };
```

The `_drawTile()` function computes and repositions the `Renderable` object to cover the lower-left corner of the camera WC bounds and tiles the object in the positive x and y directions. Note the following:

a. Steps A and B compute the position and dimension of the tiling object and the camera WC bounds.

b. Step C computes the `dx` and `dy` offsets that will translate the `Renderable` object with bounds that cover the lower-left corner of the `aCamera` WC bounds. The calls to the `Math.ceil()` function ensure that the computed `nx` and `ny` are integers.

c. Step D saves the original position of the `Renderable` object before offsetting and drawing it. Step E offsets the `Renderable` object to cover the lower-left corner of the camera WC bounds.

d. Step F computes the number of repeats required, and step G tiles the `Renderable` object in the positive x and y directions until the results cover the entire camera WC bounds.

e. Step H resets the position of the tiled object to the original location.

4. Override the `draw()` function to call the `_drawTile()` function when tiling is true.

```
TiledGameObject.prototype.draw = function (aCamera) {
    if (this.isVisible()) {
        if (this.shouldTile()) {
            // find out where we should be drawing
            this._drawTile(aCamera);
        } else {
            this.mRenderComponent.draw(aCamera);
        }
    }
};
```

Modify MyGame to Test Tiled Objects

MyGame should test for the correctness of object tiling. To test multiple layers of tiling, two separate instances of TiledGameObject and Camera are created. The two TiledGameObject instances are located at different distances from the cameras (z-depth) and are illuminated by different combinations of light sources. The newly added camera is focused on one of the Hero objects.

Only the creation of the TiledGameObject instance is of interest. This is because, once created, a TiledGameObject instance can be handled in the same manner as a GameObject instance. For this reason, only the MyGame initialize() function is examined in detail here. The rest of the MyGame functions are largely similar to previous projects and are not listed here to avoid unnecessary distraction.

```
MyGame.prototype.initialize = function () {
    // Step A: set up the cameras
    this.mHeroCam = new Camera(
        vec2.fromValues(20, 30.5),  // position of the camera
        14,                         // width of camera
        [0, 420, 300, 300],         // viewport (orgX, orgY, width, height)
        2
    );
    this.mHeroCam.setBackgroundColor([0.5, 0.5, 0.9, 1]);

    this.mCamera = new Camera(
        vec2.fromValues(50, 37.5),  // position of the camera
        100,                        // width of camera
        [0, 0, 1280, 720]           // viewport (orgX, orgY, width, height)
    );
    this.mCamera.setBackgroundColor([0.8, 0.8, 0.8, 1]);
            // sets the background to gray

    // Step B: the lights
    this._initializeLights();       // defined in MyGame_Lights.js

    // Step C: the far Background
    var bgR = new IllumRenderable(this.kBg, this.kBgNormal);
    bgR.setElementPixelPositions(0, 1024, 0, 1024);
    bgR.getXform().setSize(30, 30);
    bgR.getXform().setPosition(0, 0);
    bgR.getMaterial().setSpecular([0.2, 0.1, 0.1, 1]);
    bgR.getMaterial().setShininess(50);
    bgR.getXform().setZPos(-20);
    bgR.addLight(this.mGlobalLightSet.getLightAt(1));  // only directional light
    this.mBg = new TiledGameObject(bgR);

    // Step D: the closer Background
    var i;
    var bgR1 = new IllumRenderable(this.kBgLayer, this.kBgLayerNormal);
    bgR1.getXform().setSize(30, 30);
    bgR1.getXform().setPosition(0, 0);
    bgR1.getXform().setZPos(-10);
    for (i = 0; i < 4; i++)
        bgR1.addLight(this.mGlobalLightSet.getLightAt(i));   // all the lights
    bgR1.getMaterial().setSpecular([0.2, 0.2, 0.5, 1]);
```

```
    bgR1.getMaterial().setShininess(10);
    this.mBgL1 = new TiledGameObject(bgR1);
    this.mBgL1.setSpeed(0.1);
    this.mBgL1.setCurrentFrontDir([-1, 0]);

    // Initialize the other objects in the scene
    //
    // ... code not shown because of similarity to previous projects ...
    //
};
```

In the previous code, the two cameras are first created in step A, followed by the creation and initialization of all the light sources (in `MyGame_Lights.js`, not shown because of similarity to previous projects). Step C defines `bgR` as an `IllumRenderable` object that is being illuminated by one light source and creates a `TiledGameObject` instance based on `bgR`. Step D defines the second `IllumRenderable` object that is being illuminated by four light sources and again creates a `TiledGameObject` instance based on the `Renderable` object. Notice that the second tile object initializes its speed to 0.1 and front direction to point toward the negative x-axis. This object will move toward the left continuously. Since the `TileGameObject` `mShouldTile` variable defaults to true, both of the tile objects will tile the camera that they are drawing to.

You can now run the project and move the `Hero` object with the WASD keys. As expected, the two layers of tiled backgrounds are clearly visible with the front layer moving continuously toward the left. Move the `Hero` object to pan the cameras to verify that the tiling and the background movement behaviors are correct in both of the cameras.

Simulating Motion Parallax with Parallax Scrolling

Parallax scrolling simulates motion parallax by defining and scrolling objects at different speeds to convey the sense that these objects are located at different distances from the camera. Figure 10-5 illustrates this idea with a top view showing the conceptual distances of objects from the camera. Since this is a bird's-eye view, the width of the camera WC bounds is shown as a horizontal line at the bottom. The `Hero` object is closest to the camera in front of two layers of backgrounds, `Layer1` and `Layer2`. For typical 2D games, the vast majority of objects in the game will be located at this default distance from the camera. The background objects are located farther from the camera, behind the default distance. The distance perception can be conveyed by strategic drawings on the background objects (for example, grass fields for `Layer1` and distant mountains for `Layer2`) accompanied with appropriate scroll speeds. Take note that positions P_1 and P_2 on background objects `Layer1` and `Layer2` are directly behind the `Hero` object.

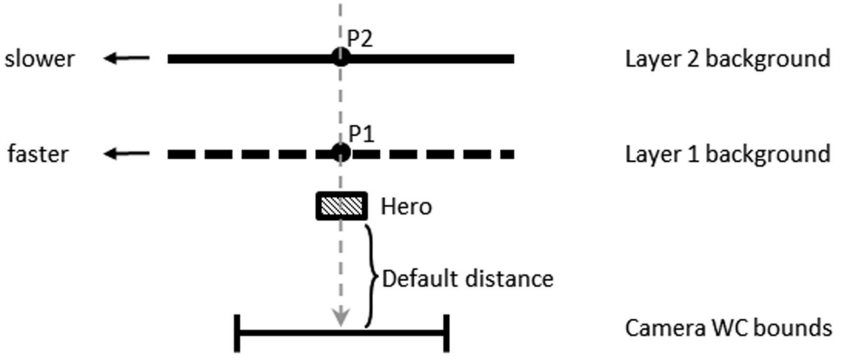

Figure 10-5. *Top view of a scene with two background objects at different distances*

Figure 10-6 shows the results of leftward parallax scrolling with a stationary camera. With `Layer1` scrolling at a faster speed than `Layer2`, position P_1 has a greater displacement than P_2 from their original positions. A continuous scrolling will move `Layer1` faster than `Layer2` and properly convey the sense that it is closer than `Layer2`. In parallax scrolling, objects that are closer to the camera always have a greater scroll speed than objects that are farther.

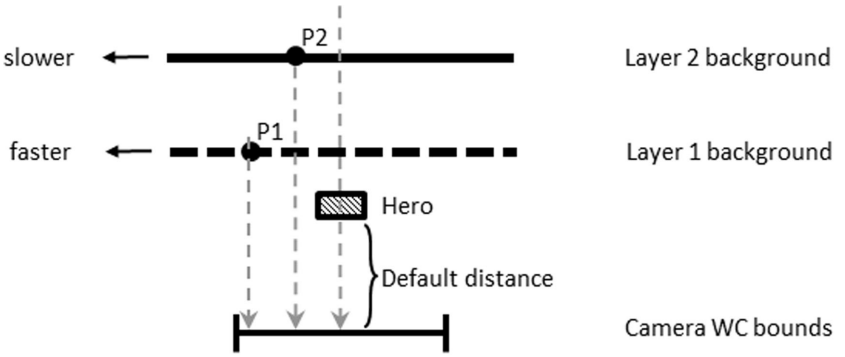

Figure 10-6. *Top view of parallax scrolling with stationary camera*

In the case when the camera is in motion, relative speeds of objects must be considered when implementing parallax scrolling. Figure 10-7 illustrates, with a top view, the situation of a moving camera with stationary objects. In this example, the camera WC bounds have moved rightward by d units. Since the movement is in the camera, all stationary objects in the camera view will appear to have been displaced by the inverse of the camera movement. For example, the stationary `Hero` object is displaced from the center leftward to the left edge of the new WC bounds. To properly simulate motion parallax, the two backgrounds, `Layer1` and `Layer2`, must be displaced by different relative distances. In this case, relative distances must be computed such that farther objects will appear to move slower. At the end of the camera movement, in the new WC bounds, the `Hero` object that is closest to the camera will appear to have been displaced leftward by d units, the `Layer1` object by 0.75d, and the `Layer2` object by 0.25d. In this way, the displacements of the objects reflect their relative distances from the camera. To achieve this, the translation of the `Hero` object is zero, and the `Layer1` and `Layer2` objects must be translated rightward by 0.25d and 0.75d, respectively. Notice that the backgrounds are translated rightward by amounts that are less than that of the camera movement, and as a result the backgrounds are actually moving leftward. For example, although the `Layer1` object is translated rightward by 0.25d, when viewed from the camera that has been moved rightwards by d, the resulting relative movement is such that the `Layer1` object has been displaced leftward by 0.75d.

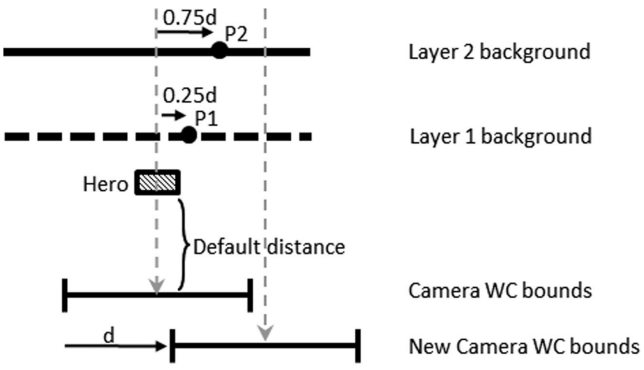

Figure 10-7. *Top view of parallax scrolling with the camera in motion*

It is important to note that in the described approach to implement parallax scrolling for a moving camera, stationary background objects are displaced. There are two limitations to this implementation. First, the object locations are changed for the purpose of conveying visual cues and do not reflect any specific game state logic. This can create challenging conflicts if the game logic requires the precise control of the movements of the background objects. Fortunately, background objects are usually designed to serve the purposes of decorating the environment and engaging the players. Background objects typically do not participate in the actual gameplay logic. The second limitation is that the stationary background objects are actually in motion and will appear so when viewed from cameras other than the one causing the motion parallax. When views from multiple cameras are necessary in the presence of motion parallax, it is important to carefully coordinate them to avoid player confusion.

The ParallaxObjects Project

This project demonstrates parallax scrolling. You can see an example of this project running in Figure 10-8. The source code to this project is defined in the `Chapter10/10.2.ParallaxObjects` folder.

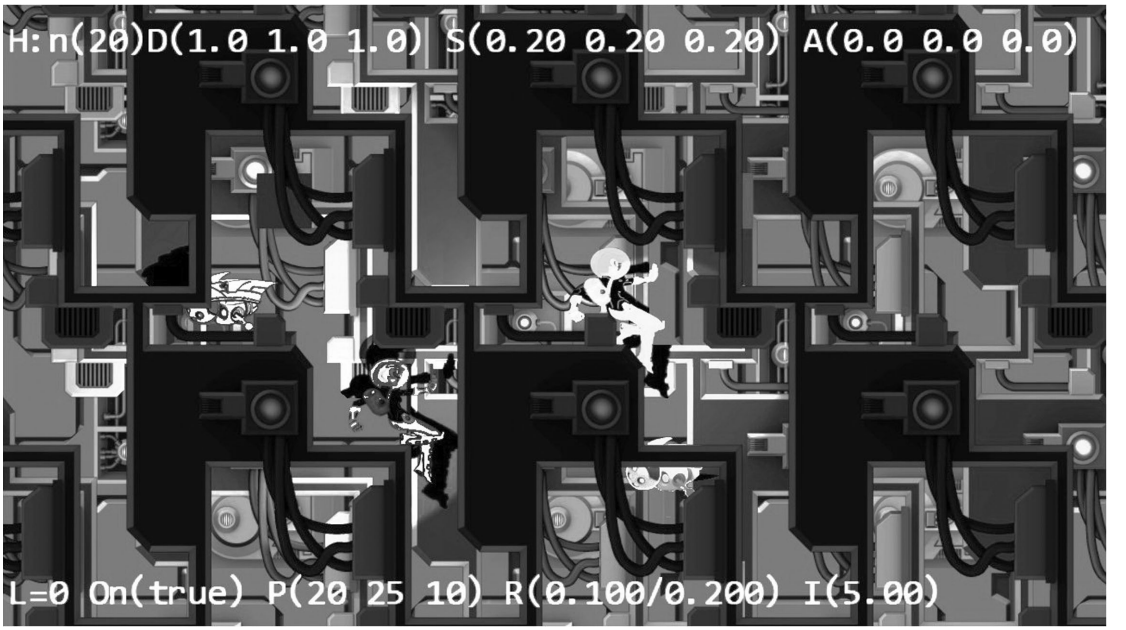

Figure 10-8. *Running the Parallax Objects project*

The controls of the project are as follows:

- *P key*: Toggles the drawing of parallax camera to provide a zoomed view of object parallax
- *WASD keys*: Move the Dye character (the hero) to pan the WC window bounds

The goals of the project are as follows:

- To understand and appreciate motion parallax
- To simulate motion parallax with parallax scrolling

Define ParallaxGameObject to Implement Parallax Scrolling

Parallax scrolling involves the continuous scrolling of objects, and `TiledGameObject` provides a convenient platform for never-ending scrolling. For this reason, `ParallaxGameObject` is defined as a subclass to the `TiledGameObject` class.

1. Create a new file in the `src/Engine/GameObjects/` folder and name it `ParallaxGameObject.js`. Add the following code to construct the object:

```
function ParallaxGameObject(renderableObj, scale, aCamera) {
    this.mRefCamera = aCamera;
    this.mCameraWCCenterRef = vec2.clone(this.mRefCamera.getWCCenter());
    this.mParallaxScale = 1;
    this.setParallaxScale(scale);
    TiledGameObject.call(this, renderableObj);
}
gEngine.Core.inheritPrototype(ParallaxGameObject, TiledGameObject);
```

The `ParallaxGameObject` object maintains `mRefCamera`, a reference to `aCamera` and `mCameraWCCenterRef`, the current WC bounds center. These values are used to compute relative movements based on the motion of the referenced camera to support parallax scrolling. The `scale` parameter is a positive value. A `scale` value of 1 represents that the object is located at the default distance, and values of less than 1 convey that the object is in front of the default distance. A `scale` of greater than 1 represents objects that are behind the default distance. The larger the `scale` value, the farther the object is from the camera.

2. Define the getter and setter functions for `mParallaxScale`, and notice that `mParallaxScale` is the inverse of the object distance.

```
ParallaxGameObject.prototype.getParallaxScale = function () {
    return this.mParallaxScale;
};

ParallaxGameObject.prototype.setParallaxScale = function(s) {
    if (s <= 0) {
        this.mParallaxScale = 1;
    } else {
        this.mParallaxScale = 1/s;
    }
};
```

3. Override the `update()` function to implement parallax scrolling.

```
ParallaxGameObject.prototype.update = function () {
    // simple default behavior
    this._refPosUpdate(); // check to see if the camera has moved
    var pos = this.getXform().getPosition();  // our own xform
    vec2.scaleAndAdd(pos, pos, this.getCurrentFrontDir(),
                     this.getSpeed() * this.mParallaxScale);
};
```

The _refPosUpdate() function is the one that computes a relative displacement based on the reference camera's WC center position. The vec2.scaleAndAdd() function moves the current object at a speed that is scaled by the mParallaxScale.

4. Define the _refPosUpdate() function.

```
ParallaxGameObject.prototype._refPosUpdate = function () {
    // now check for reference movement
    var deltaT = vec2.fromValues(0, 0);
    vec2.sub(deltaT, this.mCameraWCCenterRef, this.mRefCamera.getWCCenter());
    this.setWCTranslationBy(deltaT);
    // now update WC center ref position
    vec2.sub(this.mCameraWCCenterRef, this.mCameraWCCenterRef, deltaT);
};
```

The deltaT variable records the relative displacements, and setWCTranslationBy() moves the object to simulate parallax scrolling.

5. Define the function to translate the object to implement parallax scrolling.

```
ParallaxGameObject.prototype.setWCTranslationBy = function (delta) {
    var f = (1-this.mParallaxScale);
    this.getXform().incXPosBy(-delta[0] * f);
    this.getXform().incYPosBy(-delta[1] * f);
};
```

Testing ParallaxGameObject in MyGame

The testing of ParallaxGameObject involves testing for the correctness of parallax scrolling with the stationary camera, testing for the motion camera with an object in front of and behind the default distance, and observing the ParallaxGameObject from an alternative camera. The MyGame-level source code is largely similar to that from the previous project, and the details are not listed. The relevant part of the initialize() function is listed for the purpose of demonstrating how to create the ParallaxGameObject instances.

```
MyGame.prototype.initialize = function () {
    // Step A: set up the cameras
    // creating the two cameras and initializing the lights
    //
    // ... code not shown because of similarity to previous projects ...
    //

    // Step C: the far Background
    var bgR = new IllumRenderable(this.kBg, this.kBgNormal);
    bgR.setElementPixelPositions(0, 1024, 0, 1024);
    bgR.getXform().setSize(30, 30);
    bgR.getXform().setPosition(0, 0);
    bgR.getMaterial().setSpecular([0.2, 0.1, 0.1, 1]);
    bgR.getMaterial().setShininess(50);
    bgR.getXform().setZPos(-10);
```

```
        bgR.addLight(this.mGlobalLightSet.getLightAt(1));   // only the directional light
        this.mBg = new ParallaxGameObject(bgR, 5, this.mCamera);
        this.mBg.setCurrentFrontDir([0, -1, 0]);
        this.mBg.setSpeed(0.1);

        // Step D: the closer Background
        var i;
        var bgR1 = new IllumRenderable(this.kBgLayer, this.kBgLayerNormal);
        bgR1.getXform().setSize(25, 25);
        bgR1.getXform().setPosition(0, 0);
        bgR1.getXform().setZPos(0);
        bgR1.addLight(this.mGlobalLightSet.getLightAt(1));   // the directional light
        bgR1.addLight(this.mGlobalLightSet.getLightAt(2));   // the hero spotlight light
        bgR1.addLight(this.mGlobalLightSet.getLightAt(3));   // the hero spotlight light
        bgR1.getMaterial().setSpecular([0.2, 0.2, 0.5, 1]);
        bgR1.getMaterial().setShininess(10);
        this.mBgL1 = new ParallaxGameObject(bgR1, 3, this.mCamera);
        this.mBgL1.setCurrentFrontDir([0, -1, 0]);
        this.mBgL1.setSpeed(0.1);

        // Step E: the front layer
        var f = new TextureRenderable(this.kBgLayer);
        f.getXform().setSize(30, 30);
        f.getXform().setPosition(0, 0);
        this.mFront = new ParallaxGameObject(f, 0.9, this.mCamera);

        // ... Identical to previous project ...
};
```

The mBg object is created as a ParallaxGameObject with a scale value of 5, mBgL1 with a scale of 3, and mFront with a scale of 0.9. Recall that scale is the second parameter of the ParallaxGameObject constructor, and it signifies the object distance from the camera, with values greater than 1 being farther from the default distance and less than 1 being closer than the default distance. In this case, mBg is the furthest and mBgL1 is closer, and both are behind the default distance. The mFront object is actually closer to the camera than the default distance. It is in front of the Hero object. Notice that mBg and mBgL1 are both defined to be continuously scrolling downward (setCurrentFrontDir([0, -1, 0])) with a speed of 0.1.

You can now run the project and observe the darker foreground layer partially blocking the Hero and Minions objects. You can see the two background layers continuously scrolling downward but with different speeds. The mBg object is farther away and thus scrolls slower than the mBgL1 object. Move the Hero object with WASD and pan the camera. You will notice the front-layer parallax scrolls at a faster speed than all other objects, and as a result, panning the camera reveals different parts of the animated Minion objects.

Press the P key to enable the drawing of the second camera. Notice that the view in this camera is as expected. The Hero object is stationary, and the two background layers scroll continuously downward. Now, if you move the Hero object to pan the main camera, note the foreground and background objects in the second camera view are also moving and exhibit motion parallax even though the second camera is not moving! As game designer, it is important to ensure this side effect does not cause player confusion.

Layer Management

Although the engine you are developing is for supporting 2D games, you have worked with a few situations where depth ordering and drawing orders are important. For example, the shadow receiver must always be defined behind the shadow casters, and as discussed in the previous example, foreground and background parallax objects must be carefully defined and drawn in the order of their depth ordering. It is convenient for the game engine to provide a utility manager to help game programmers manage and work with the depth layering. A typical 2D game can have the following layers, in the order of the distance from the camera, from nearest to furthest:

- *Heads-up display (HUD) layer*: Typically closest to the camera displaying essential user interface information
- *Foreground or front layer*: The layer in front of the game objects for decorative or partial occlusion of the game objects
- *Actor layer*: The default distance layer in Figure 10-5, where all game objects reside
- *Shadow Receiver layer*: The layer behind the actor layer to receive potential shadows
- *Background layer*: The decorative background

Each layer will reference all objects defined for that layer, and these objects will be drawn in the order they were inserted into the layer, with the last inserted drawn last and covering objects before it. This section presents the `LayerManager` engine component to support the described five layers to relieve game programmers from the details of managing updates and drawings the objects. Note that the number of layers a game engine should support is determined by the kinds of games that the engine is designed to build. The five layers presented are logical and convenient for simple games. You may choose to expand the number of layers in your own game engine.

The Layer Manager Project

This project demonstrates how to develop a utility component to assist in managing layers for the game programmers. You can see an example of this project running in Figure 10-9. The source code to this project is defined in the `Chapter10/10.3.LayerManager` folder.

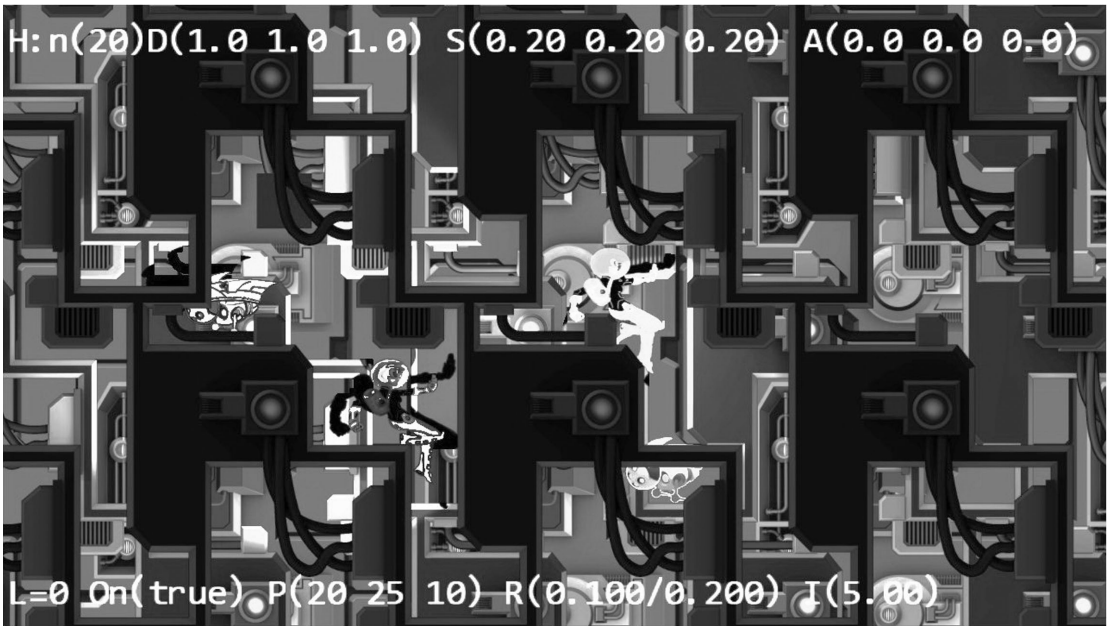

Figure 10-9. *Running the Layer Manager project*

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

- *P key*: Toggles the drawing of parallax camera, to provide a zoomed view of object parallax
- *WASD keys*: Move the Dye character (the hero) to pan the WC window bounds

The goals of the project are as follows:

- To appreciate the importance of layering in 2D games
- To develop a layer manager engine component

Layer Management in the Engine

Follow the pattern of defining engine components.

1. Create a new file in the `src/Engine/Core/` folder and name it `Engine_LayerManager.js`. This file will implement the `LayerManager` engine component.
2. Define enumerators for the layers.

```
gEngine.eLayer = Object.freeze({
    eBackground: 0,
    eShadowReceiver: 1,
    eActors: 2,
    eFront: 3,
    eHUD: 4
});
```

3. Define the LayerManager component following the JavaScript Module Pattern as follows:

```
var gEngine = gEngine || { };
    // initialize the variable while ensuring it is not redefined

gEngine.LayerManager = (function () {
    var mPublic = { };
    return mPublic;
}() );
```

4. Within the LayerManager component, define appropriate constants and instance variables to keep track of the layers.

```
var kNumLayers = 5;
var mAllLayers = [];
```

mAllLayers is an array of GameObjectSet instances representing each of the five layers.

5. Define an initialize() function to create the array of GameObjectSet instances.

```
var initialize = function() {
    mAllLayers[gEngine.eLayer.eBackground] = new GameObjectSet();
    mAllLayers[gEngine.eLayer.eShadowReceiver] = new GameObjectSet();
    mAllLayers[gEngine.eLayer.eActors] = new GameObjectSet();
    mAllLayers[gEngine.eLayer.eFront] = new GameObjectSet();
    mAllLayers[gEngine.eLayer.eHUD] = new GameObjectSet();
};
```

6. Define a cleanUp() function to reset the mAllLayer array.

```
var cleanUp = function() {
    initialize();
};
```

7. Define functions to add to, remove from, and query the layers.

```
var addToLayer = function(layerEnum, obj) {
    mAllLayers[layerEnum].addToSet(obj);
};

var removeFromLayer = function(layerEnum, obj) {
    mAllLayers[layerEnum].removeFromSet(obj);
};

var layerSize = function(layerEnum) {
    return mAllLayers[layerEnum].size();
};
```

```
var addAsShadowCaster = function(obj) {
    var i;
    for (i = 0; i<mAllLayers[gEngine.eLayer.eShadowReceiver].size(); i++)
        mAllLayers[gEngine.eLayer.eShadowReceiver].getObjectAt(i).
            addShadowCaster(obj);
};
```

Note the `addAsShadowCaster()` function assumes that the shadow receiver objects are already inserted into the `eShadowReceiver` layer. Additionally, this function adds the casting object to all receivers in the layer.

8. Define functions to draw a specific layer or all the layers, from the furthest to the nearest to the camera.

```
var drawLayer = function(layerEnum, aCamera) {
    mAllLayers[layerEnum].draw(aCamera);
};

var drawAllLayers = function(aCamera) {
    var i;
    for (i=0; i<kNumLayers; i++)
        mAllLayers[i].draw(aCamera);
};
```

9. Define a function to move a specific object such that it will be drawn last (on top).

```
var moveToLayerFront = function(layerEnum, obj) {
    mAllLayers[layerEnum].moveToLast(obj);
};
```

10. Define functions to update a specific layer or all the layers.

```
var updateLayer = function(layerEnum) {
    mAllLayers[layerEnum].update();
};

var updateAllLayers = function() {
    var i;
    for (i=0; i<kNumLayers; i++)
        mAllLayers[i].update();
};
```

11. Remember to add all the functions to the public interface.

```
var mPublic = {
  initialize: initialize,
  drawAllLayers: drawAllLayers,
  updateAllLayers: updateAllLayers,
  cleanUp: cleanUp,
```

```
            drawLayer: drawLayer,
            updateLayer: updateLayer,
            addToLayer: addToLayer,
            addAsShadowCaster: addAsShadowCaster,
            removeFromLayer: removeFromLayer,
            moveToLayerFront: moveToLayerFront,
            layerSize: layerSize
          };
          return mPublic;
```

Modify Engine Components and Objects

You must modify the rest of the game engine slightly to integrate the new `LayerManager` component.

Enhance the GameObjectSet Functionality

Add the following two functions to support removing membership and moving objects to the end of a set array:

```
GameObjectSet.prototype.removeFromSet = function (obj) {
    var index = this.mSet.indexOf(obj);
    if (index > -1)
        this.mSet.splice(index, 1);
};

// Ensures the given obj is the last element (when drawn, appears on top)
GameObjectSet.prototype.moveToLast = function (obj) {
    this.removeFromSet(obj);
    this.addToSet(obj);
};
```

Initialize LayerManager in gEngine_Core

Modify the engine's `Core` component to initialize the `LayerManager` component in the `initializeEngineCore()` function.

```
// initialize all of the EngineCore components
var initializeEngineCore = function (htmlCanvasID, myGame) {
    _initializeWebGL(htmlCanvasID);
    gEngine.VertexBuffer.initialize();
    gEngine.Input.initialize(htmlCanvasID);
    gEngine.AudioClips.initAudioContext();
    gEngine.Physics.initialize();
    gEngine.LayerManager.initialize();

    // Inits DefaultResources, when done,
    // invoke the anonymous function to call startScene(myGame).
    gEngine.DefaultResources.initialize(function () { startScene(myGame); });
};
```

Define the Update Function for Layer Membership Objects

Define update functions for objects that may appear as members in the LayerManager layers: Renderable, FontRenderable, and ShadowReceiver.

Modify MyGame to Work with LayerManager

The MyGame level implements the same functionality as in the previous project. The only difference is the delegation of layer management to the LayerManager component. The following description focuses only on how the LayerManager functions are called.

1. Modify the unloadScene() function to clean up the LayerManager.

```
MyGame.prototype.unloadScene = function () {
    gEngine.LayerManager.cleanUp();

    gEngine.Textures.unloadTexture(this.kMinionSprite);
    gEngine.Textures.unloadTexture(this.kBg);
    gEngine.Textures.unloadTexture(this.kBgNormal);
    gEngine.Textures.unloadTexture(this.kBgLayer);
    gEngine.Textures.unloadTexture(this.kBgLayerNormal);
    gEngine.Textures.unloadTexture(this.kMinionSpriteNormal);
};
```

2. Modify the initialize() function to add the game objects to the corresponding layers in the LayerManager component.

```
MyGame.prototype.initialize = function () {
    // ... Identical to previous project ...

    // add to layer managers ...
    gEngine.LayerManager.addToLayer(gEngine.eLayer.eBackground, this.mBg);
    gEngine.LayerManager.addToLayer(gEngine.eLayer.eShadowReceiver, this.mBgShadow1);

    gEngine.LayerManager.addToLayer(gEngine.eLayer.eActors, this.mIllumMinion);
    gEngine.LayerManager.addToLayer(gEngine.eLayer.eActors, this.mLgtMinion);
    gEngine.LayerManager.addToLayer(gEngine.eLayer.eActors, this.mIllumHero);
    gEngine.LayerManager.addToLayer(gEngine.eLayer.eActors, this.mLgtHero);

    gEngine.LayerManager.addToLayer(gEngine.eLayer.eFront, this.mBlock1);
    gEngine.LayerManager.addToLayer(gEngine.eLayer.eFront, this.mBlock2);
    gEngine.LayerManager.addToLayer(gEngine.eLayer.eFront, this.mFront);
    gEngine.LayerManager.addToLayer(gEngine.eLayer.eHUD, this.mMsg);
    gEngine.LayerManager.addToLayer(gEngine.eLayer.eHUD, this.mMatMsg);
};
```

3. Modify the `draw()` function to rely on the `LayerManager` component for the actual drawings.

```
MyGame.prototype.draw = function () {
    // Step A: clear the canvas
    gEngine.Core.clearCanvas([0.9, 0.9, 0.9, 1.0]); // clear to light gray

    this.mCamera.setupViewProjection();
    gEngine.LayerManager.drawAllLayers(this.mCamera);

    if (this.mShowHeroCam) {
        this.mParallaxCam.setupViewProjection();
        gEngine.LayerManager.drawAllLayers(this.mParallaxCam);
    }
};
```

4. Modify the `update()` function to rely on the `LayerManager` component for the actual update of all game objects.

```
MyGame.prototype.update = function () {
    this.mCamera.update();  // to ensure proper interploated movement effects
    this.mParallaxCam.update();

    gEngine.LayerManager.updateAllLayers();

    // ... Identical to previous project ...
};
```

You can now run the project and observe the same output and interactions as the previous project. The important observation for this project is in the implementation. By inserting game objects to the proper `LayerManager` layers during `initialize()`, the `draw()` and `update()` functions of a game level can be much cleaner. The simpler and cleaner `MyGame update()` function is of special importance. Instead of being crowded with mundane game object `update()` function calls, the `MyGame update()` function can now focus on implementing the game logic and controlling the interactions between game objects.

Summary

This chapter explained the need for tiling and introduced the `TileGameObject` to implement a simple algorithm that tiles and covers a given camera WC bounds. The basics of parallax and approaches to simulate motion parallax with parallax scrolling were introduced. Motion parallax with stationary and motioned camera were examined, and solutions were derived and implemented. You learned that computing movements relative to the camera motions to displace background objects results in visually pleasing motion parallax but may cause player confusion when viewed from different cameras. With shadow computations introduced earlier and now parallax scrolling, game programmers must dedicate code and attention to coordinate the drawing order of different types of objects. To facilitate the programmability of the game engine, the `LayerManager` engine component is presented as a utility tool to alleviate game programmers from managing the drawing of the layers.

Your game engine is now completed, and it can draw objects with texture maps, sprite animations, and even illumination by various light sources. The engine defines proper abstractions for simple behaviors, mechanisms to approximate and accurately compute the collisions, and simulates the physical behaviors of these objects. Views from multiple cameras can conveniently be displayed over the same game screens with manipulation functionality that is smoothly interpolated. Keyboard/mouse input is supported, and now background objects can scroll without bounds and simulate motion parallax.

The important next step is to go through a simple game design process and implement a game based on your new completed game engine.

Game Design Considerations

In previous "Game Design Considerations" sections, you've explored how developing one simple mechanic from the ground up can lead in many directions and be applied to a variety of game types. Creative teams in game design studios frequently debate which elements of game design take the lead in the creative process. Writers often believe story comes first, while many designers believe that story and everything else must be secondary to gameplay. There's no right or wrong answer, of course. The creative process is a chaotic system, and every team and studio is unique; some creative directors want to tell a particular story and search for mechanics and genres that are best suited to supporting specific narratives, while others are gameplay purists and completely devoted to a culture of "gameplay first, next, and last." The decision often comes down to understanding your audience. If you're creating an AAA first-person-shooter experience, for example, consumers will have specific expectations for many of the core elements of play, and it's usually a smart move to ensure that gameplay drives the design. If you're creating an indie adventure game designed to provide players with new experiences and unexpected twists, however, story and setting might lead the way.

Many game designers (and this includes seasoned veterans as well as those new to the discipline) begin new projects by designing complex experiences that are relatively minor variations on existing well-understood mechanics; and while there are sound reasons for this approach (as in the case of AAA studios developing content for particularly demanding audiences or a desire to work with mechanics that have proven to be successful across many titles), it tends to significantly limit exploration into new territory and is one reason why many gamers complain about creative stagnation and a lack of gameplay diversity between games within the same genre. Many of us who are professional game designers grew up enjoying certain kinds of games and dreamed about creating new experiences based on the mechanics we know and love, and several decades of that culture has solidified much of the industry around a comparatively few number of similar mechanics and conventions. The good news is that a rapidly growing independent and small studio community has boldly begun throwing long-standing genre convention to the wind in recent years, and new distribution platforms like mobile app stores and Valve's Steam have opened opportunities for a wide range of never-before-seen mechanics and experiences to flourish.

If you continue exploring game design, you'll realize there are relatively few completely unique core mechanics, but there are endless opportunities for innovating as you intentionally and logically build those elemental interactions into more complex causal chains, adding unique flavor and texture through elegant integration with the other elements of game design. Some of the most groundbreaking games in recent memory were created through exercises very much like the mechanic exploration you've done in the "Game Design Considerations" sections. Valve's Portal, for example, is based on the same kind of "escape the room" sandbox you explored and is designed around a similarly simple base mechanic. So, what made Portal such a breakthrough hit? While many things need to come together to create a hit game, Portal undoubtedly benefitted from a design team that started building the experience from the most basic mechanic and smartly increased complexity as they became increasingly fluent in its unique structure and characteristics, instead of starting at the 10,000-foot level with a codified genre and a host of expectations and overly familiar mechanics.

Of course, nobody talks about Portal without also mentioning the rogue artificial intelligence character GLaDOS and her Aperture Laboratories playground: setting, narrative, and audiovisual design are as important to the Portal experience as the portal-launching mechanic, and it's hard to separate gameplay from the narrative given how skillfully intertwined they are. The projects in this chapter provide a good opportunity to begin similarly situating the mechanic from the "Game Design Considerations" sections in a unique setting and context. You've probably noticed many of the projects throughout this book are building toward a sci-fi visual theme, with a spacesuit-wearing hero character, a variety of flying robots, and now in Chapter 10 the introduction of large-scale parallaxing environments. And while you're not building a game with the same degree of environment and interaction complexity as Portal, that doesn't mean you don't have the same opportunity to develop a highly engaging game setting, context, and cast of characters.

The first thing you should notice about the Tiled Objects project is the dramatic impact on environment experience and scale compared to earlier projects. The factors enhancing presence in this project are the three independently moving layers (hero, moving wall, and stationary wall) and the seamless tiling of the two background layers. Compare the Tiled Objects project to the Shadow Shaders project from Chapter 8; notice the difference in presence when the environment is broken into multiple layers that appear to move in an analogous (if not physically accurate) way to how you experience movement in the physical world. The sense of presence is further strengthened when you add multiple background layers of parallaxing movement in the Parallax Objects project; as you move through the physical world, the environment appears to move at different speeds, with closer objects seeming to pass-by quickly while objects toward the horizon appear to move slowly. Parallaxing environment objects simulate this effect, adding considerable depth and interest to game environments. The Layer Manager project pulls things together and begins to show the potential for a game setting to immediately engage the imaginations of players. With just a few techniques, you're able to create the impression of a massive environment that might be the interior of an ancient alien machine, the outside of a large space craft, or anything else you might care to create. Try using different kinds of image assets with this technique: exterior landscapes, underwater locations, abstract shapes, and the like would all be interesting to explore. You'll often find inspiration for game settings by experimenting with just a few basic elements, as you did in Chapter 10.

Pairing game environment design (both audio and visual) with interaction design (and occasionally the inclusion of haptic feedback–like controller vibrations) is the mechanism you use to create and enhance presence, and the relationship that environments and interactions have with the game mechanic contributes the majority of what players experience in games. Environment design and narrative context create the game setting, and as previously mentioned, the most successful and memorable games achieve an excellent harmony between game setting and player experience. At this point, the mechanic from the "Game Design Considerations" section in Chapter 9 has been intentionally devoid of any game setting context, and you've only briefly considered the interaction design, leaving you free to explore any setting that captures your interest. In Chapter 11, you'll pair and further evolve the sci-fi setting and image assets used in the main chapter projects with the unlocking mechanic from the "Game Design Considerations" section to create a fairly advanced 2D platformer game-level prototype.