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 11-1 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 duplicating boundaries. Proper tiling conveys a sense of complexity in a boundless game world by creating only a single image.

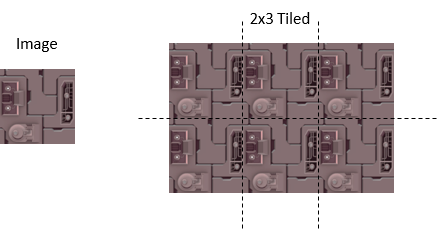


Figure 11-1. Tiling of a strategically drawn background image

Parallax is the apparent displacements of objects when they are viewed from different positions. Figure 11-2 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 appear to 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.

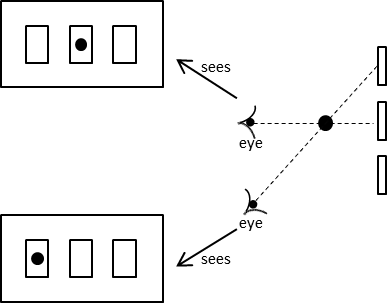


Figure 11-2. Parallax: objects appearing at different positions when observed from different viewpoints

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 11-3. 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 11-3 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 at the origin does not need to be drawn.

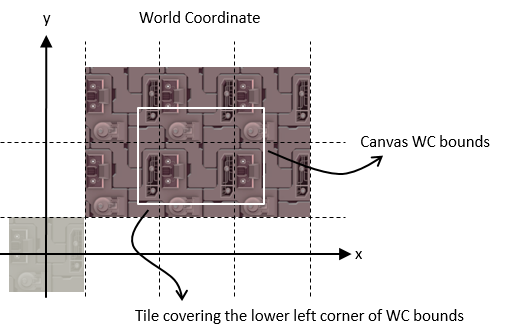


Figure 11-3. Generating tiled background for camera WC bounds

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 11‑4. The source code to this project is defined in the chapter11/11.1.tiled\_objects folder.

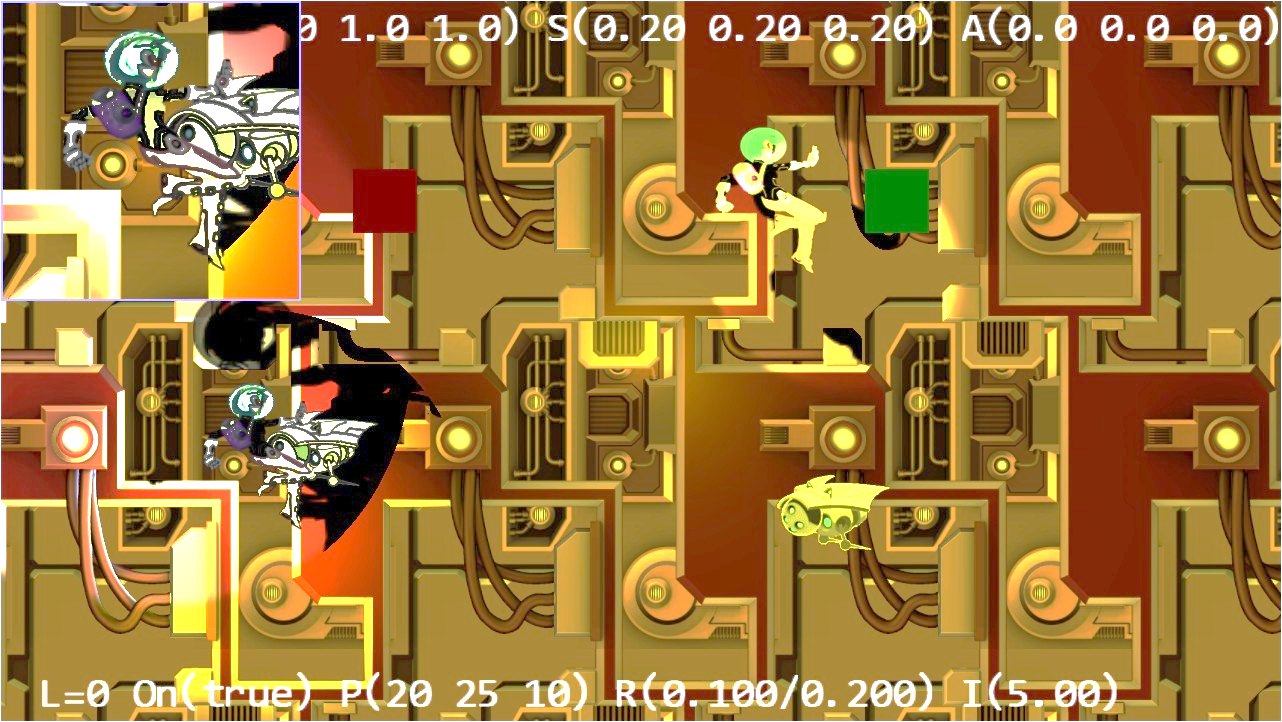


Figure 11-4. Running the Tiled Objects project

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, bg\_layer.png, and bg\_layer\_normal.png. The Hero and Minion objects are represented by sprite elements in the minion\_sprite.png image, and bg.png and bg\_layer.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/game\_objects folder and name it tiled\_game\_object.js. Add the following code to construct the object. The mShouldTile variable provides the option to stop the tiling process.

class TiledGameObject extends GameObject {

constructor(renderableObj) {

super(renderableObj);

this.mShouldTile = true; // can switch this off if desired

}

… implementation to follow …

export deafult TiledGameObject;

1. Define the getter and setter functions for mShouldTile

setIsTiled(t) { this.mShouldTile = t; }

shouldTile() { return this.mShouldTile; }

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

\_drawTile(aCamera) {

// Step A: Compute the positions and dimensions of tiling object.

let xf = this.getXform();

let w = xf.getWidth();

let h = xf.getHeight();

let pos = xf.getPosition();

let left = pos[0] - (w / 2);

let right = left + w;

let top = pos[1] + (h / 2);

let bottom = top - h;

// Step B: Get the world positions and dimensions of the drawing camera.

let wcPos = aCamera.getWCCenter();

let wcLeft = wcPos[0] - (aCamera.getWCWidth() / 2);

let wcRight = wcLeft + aCamera.getWCWidth();

let wcBottom = wcPos[1] - (aCamera.getWCHeight() / 2);

let wcTop = wcBottom + aCamera.getWCHeight();

// Step C: Determine the offset to the camera window's lower left corner.

let 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.

let sX = pos[0];

let 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.

let 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

let cx = nx;

let 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:

1. Steps A and B compute the position and dimension of the tiling object and the camera WC bounds.
2. 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 dx and dy are integer multiples of the Renderable width and height. This is essential to ensure there is no overlaps or gaps during tiling.
3. 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.
4. 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. The calls to the Math.ceil() function ensure that the computed nx and ny, the number of times to tile in the x and y directions, are integers.
5. Step H resets the position of the tiled object to the original location.
6. Override the draw() function to call the \_drawTile() function when tiling is enabled.

draw(aCamera) {

if (this.isVisible() && (this.mDrawRenderable)) {

if (this.shouldTile()) {

// find out where we should be drawing

this.\_drawTile(aCamera);

} else {

this.mRenderComponent.draw(aCamera);

}

}

}

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

### 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 init() function of the MyGame class is examined in detail. The rest of the MyGame functions are largely similar to previous projects and are not listed here to avoid unnecessary distraction.

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, 1280, 720] // viewport (orgX, orgY, width, height)

);

this.mCamera.setBackgroundColor([0.8, 0.8, 0.8, 1]);

// sets the background to gray

this.mHeroCam = new engine.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]);

// Step B: the lights

this.\_initializeLights(); // defined in MyGame\_Lights.js

// Step C: the far Background

let bgR = new engine.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(-5);

bgR.addLight(this.mGlobalLightSet.getLightAt(1)); // only the directional light

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

// Step D: the closer Background

let i;

let bgR1 = new engine.IllumRenderable(this.kBgLayer, this.kBgLayerNormal);

bgR1.getXform().setSize(30, 30);

bgR1.getXform().setPosition(0, 0);

bgR1.getXform().setZPos(-2);

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 engine.TiledGameObject(bgR1);

// Initialize the other objects in the scene

//

… code not shown because of similarity to previous projects …

//

}

In the listed code, the two cameras are first created in step A, followed by the creation and initialization of all the light sources (in my\_game\_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. Since the mShouldTile variable of the TileGameObject class 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. 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.

An interesting observation is that while the two layers of backgrounds are located at different distances from the camera, when the camera pans the two background images scroll synchronously. If not for the differences in light source illumination, it would appear as though the background is actually a single image. This example illustrates the importance of simulating motion parallax.

# 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 11-5 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 the 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 and on background objects Layer1 and Layer2 are directly behind the Hero object.

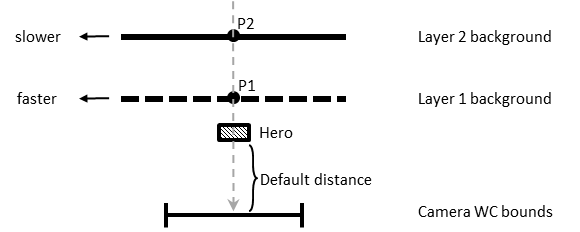


Figure 11-5. Top view of a scene with two background objects at different distances

Figure 11-6 shows the results of leftward parallax scrolling with a stationary camera. With Layer1 scrolling at a faster speed than Layer2, position has a greater displacement than 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.

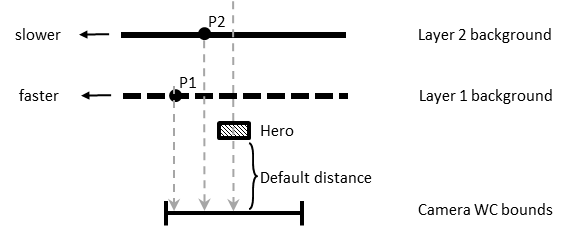


Figure 11-6. Top view of parallax scrolling with stationary camera

In the case when the camera is in motion, relative speeds of objects must be considered when implementing parallax scrolling. Figure 11-7 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.

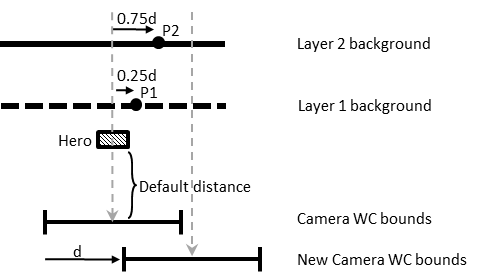


Figure 11-7. Top view of parallax scrolling with the camera in motion

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 Parallax Objects Project

This project demonstrates parallax scrolling. You can see an example of this project running in Figure 11-8. The source code to this project is defined in the chapter11/11.2.parallax\_objects folder.

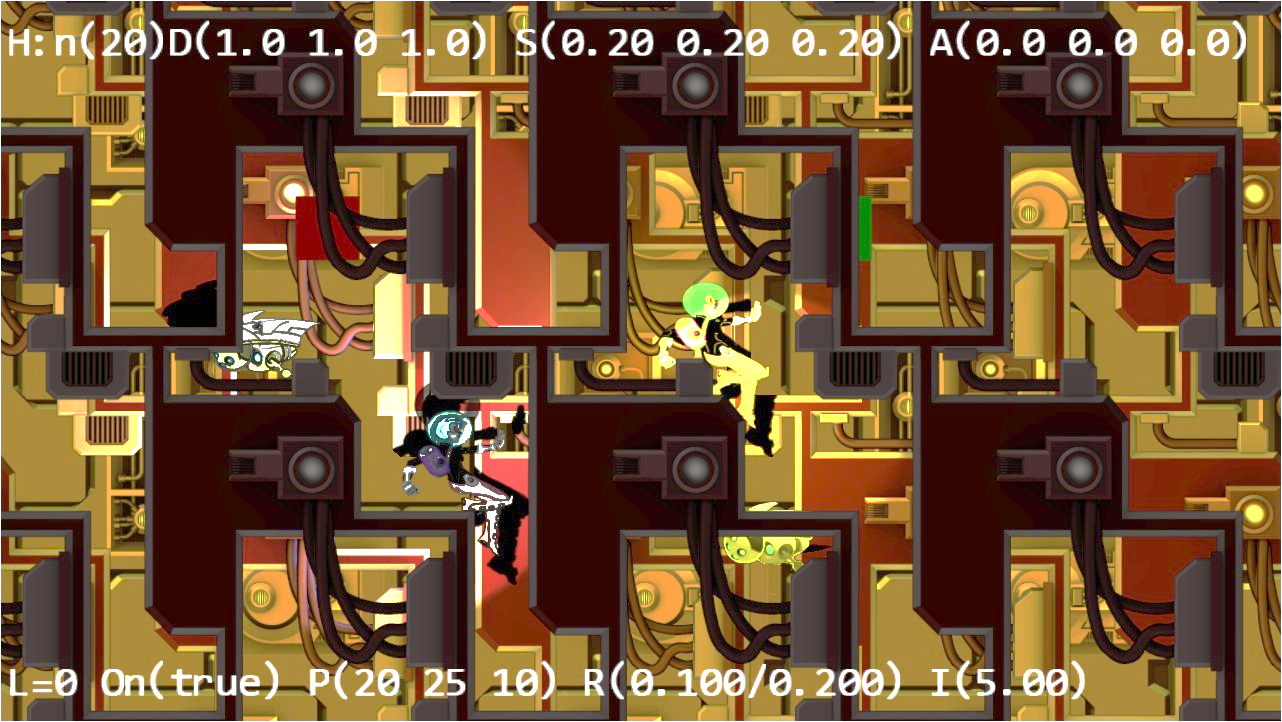


Figure 11-8. Running the Parallax Objects project

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/game\_objects folder and name it parallax\_game\_object.js. Add the following code to construct the object:

import TiledGameObject from "./tiled\_game\_object.js";

class ParallaxGameObject extends TiledGameObject {

constructor(renderableObj, scale, aCamera) {

super(renderableObj);

this.mRefCamera = aCamera;

this.mCameraWCCenterRef = vec2.clone(this.mRefCamera.getWCCenter());

this.mParallaxScale = 1;

this.setParallaxScale(scale);

}

… implementation to follow …

export default ParallaxGameObject;

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.

1. Define the getter and setter functions for mParallaxScale. Notice the clamping of negative values; this variable must be positive in value.

getParallaxScale() { return this.mParallaxScale; }

setParallaxScale(s) {

this.mParallaxScale = s;

if (s <= 0) {

this.mParallaxScale = 1;

}

}

1. Override the update() function to implement parallax scrolling.

update() {

// simple default behavior

this.\_refPosUpdate(); // check to see if the camera has moved

super.update();

}

The \_refPosUpdate() function is the one that computes a relative displacement based on the reference camera’s WC center position.

1. Define the \_refPosUpdate() function.

\_refPosUpdate() {

// now check for reference movement

let deltaT = vec2.fromValues(0, 0);

vec2.sub(deltaT, this.mCameraWCCenterRef, this.mRefCamera.getWCCenter());

this.setWCTranslationBy(deltaT);

// update WC center ref position

vec2.sub(this.mCameraWCCenterRef, this.mCameraWCCenterRef, deltaT);

}

The deltaT variable records the movement of the camera and setWCTranslationBy() moves the object to simulate parallax scrolling.

1. Define the function to translate the object to implement parallax scrolling. The negative delta is designed to move the object in the same direction as that of the camera. Notice the variable f is 1 minus the inverse of mParallaxScale.

setWCTranslationBy(delta) {

let f = (1 – (1/this.mParallaxScale));

this.getXform().incXPosBy(-delta[0] \* f);

this.getXform().incYPosBy(-delta[1] \* f);

}

When mParallaxScale is less than 1, the inverse is greater than 1 and f becomes a negative number. In this case, when the camera moves, the object will move in the opposite direction and thus create the sensation that the object is in front of the default distance.

Conversely, when mParallaxScale is greater than 1, its inverse will be less than 1 and result in a positive f with a value of less than 1. In this case, the object will be moving in the same direction as the camera, only slower. A larger mParallaxScale would correspond to f value being closer to 1, and the movement of the object will be closer to that of the camera, or the object will appear to be at a further distance from the camera.

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

### 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 alternate camera. The source code of the MyGame level is largely similar to that from the previous project, and the details are not listed. The relevant part of the init() function is listed for the purpose of demonstrating how to create the ParallaxGameObject instances.

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, 1280, 720] // viewport (orgX, orgY, width, height)

);

this.mCamera.setBackgroundColor([0.8, 0.8, 0.8, 1]);

// sets the background to gray

this.mParallaxCam = new engine.Camera(

vec2.fromValues(40, 30), // position of the camera

45, // width of camera

[0, 420, 600, 300], // viewport (orgX, orgY, width, height)

2

);

this.mParallaxCam.setBackgroundColor([0.5, 0.5, 0.9, 1]);

// Step B: the lights

this.\_initializeLights(); // defined in MyGame\_Lights.js

// Step C: the far Background

let bgR = new engine.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(-5);

// only the directional light

bgR.addLight(this.mGlobalLightSet.getLightAt(1));

this.mBg = new engine.ParallaxGameObject(bgR, 5, this.mCamera);

// Step D: the closer Background

let i;

let bgR1 = new engine.IllumRenderable(this.kBgLayer, this.kBgLayerNormal);

bgR1.getXform().setSize(25, 25);

bgR1.getXform().setPosition(0, -15);

bgR1.getXform().setZPos(0);

// the directional light

bgR1.addLight(this.mGlobalLightSet.getLightAt(1));

// the hero spotlight light

bgR1.addLight(this.mGlobalLightSet.getLightAt(2));

// the hero spotlight light

bgR1.addLight(this.mGlobalLightSet.getLightAt(3));

bgR1.getMaterial().setSpecular([0.2, 0.2, 0.5, 1]);

bgR1.getMaterial().setShininess(10);

this.mBgL1 = new engine.ParallaxGameObject(bgR1, 3, this.mCamera);

// Step E: the front layer

let f = new engine.TextureRenderable(this.kBgLayer);

f.getXform().setSize(50, 50);

f.getXform().setPosition(-3, 2);

this.mFront = new engine.ParallaxGameObject(f, 0.9, this.mCamera);

… identical to previous code …

}

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. This parameter 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, mBgL1 being closer, and both are behind the default distance. The mFront object is closer to the camera than the default distance. It is in front of the Hero object.

You can now run the project and observe the darker foreground layer partially blocking the Hero and Minions objects. You can move the Hero object to pan the camera and observe the two background layers scrolling at different speeds. The mBg object is farther away and thus scrolls slower than the mBgL1 object. You will also 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 stationary Minion objects.

Press the P key to enable the drawing of the second camera. Notice that when the Hero is stationary, the view in this camera is as expected, not moving. 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 designers, 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 11-5, 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 Layer 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 11-9. The source code to this project is defined in the chapter11/11.3.layer\_manager folder.



Figure 11-9. 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 an engine component, for example, similar to that of physics and particle systems.

1. Create a new file in the src/engine/components folder and name it layer.js. This file will implement the Layer engine component.
2. Define enumerators for the layers.

const eBackground = 0;

const eShadowReceiver = 1;

const eActors = 2;

const eFront = 3;

const eHUD = 4;

1. Define appropriate constants and instance variables to keep track of the layers. The mAllLayers variable is an array of GameObjectSet instances representing each of the five layers.

let kNumLayers = 5;

let mAllLayers = [];

1. Define an init() function to create the array of GameObjectSet instances.

function init() {

mAllLayers[eBackground] = new GameObjectSet();

mAllLayers[eShadowReceiver] = new GameObjectSet();

mAllLayers[eActors] = new GameObjectSet();

mAllLayers[eFront] = new GameObjectSet();

mAllLayers[eHUD] = new GameObjectSet();

}

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

function cleanUp() {

init();

}

1. Define functions to add to, remove from, and query the layers. Note the addAsShadowCaster() function assumes that the shadow receiver objects are already inserted into the eShadowReceiver layer and adds the casting object to all receivers in the layer.

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

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

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

function addAsShadowCaster(obj) {

let i;

for (i = 0; i < mAllLayers[eShadowReceiver].size(); i++) {

mAllLayers[eShadowReceiver].getObjectAt(i).addShadowCaster(obj);

}

}

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

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

function drawAllLayers(aCamera) {

let i;

for (i = 0; i < kNumLayers; i++) {

mAllLayers[i].draw(aCamera);

}

}

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

function moveToLayerFront(layerEnum, obj) {

mAllLayers[layerEnum].moveToLast(obj);

}

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

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

function updateAllLayers() {

let i;

for (i = 0; i < kNumLayers; i++) {

mAllLayers[i].update();

}

}

1. Remember to export all the defined functionality.

export {

// array indices

eBackground, eShadowReceiver, eActors, eFront, eHUD,

// init and cleanup

init, cleanUp,

// draw/update

drawLayer, drawAllLayers,

updateLayer, updateAllLayers,

// layer-specific support

addToLayer, addAsShadowCaster,

removeFromLayer, moveToLayerFront,

layerSize

}

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

### Modify Engine Components and Objects

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

#### Enhance the GameObjectSet Functionality

Add the following function to support moving objects to the end of a set array:

moveToLast(obj) {

this.removeFromSet(obj);

this.addToSet(obj);

}

#### Initialize Layer in index.js

In addition to import/export the Layer component, modify the engine init() and clearnUp() functions in index.js to initialize and cleanup the component.

… identical to previous code …

function init(htmlCanvasID) {

glSys.init(htmlCanvasID);

vertexBuffer.init();

input.init(htmlCanvasID);

audio.init();

shaderResources.init();

defaultResources.init();

layer.init();

}

function cleanUp() {

layer.cleanUp();

loop.cleanUp();

shaderResources.cleanUp();

defaultResources.cleanUp();

audio.cleanUp();

input.cleanUp();

vertexBuffer.cleanUp();

glSys.cleanUp();

}

#### Define the Update Function for Layer Membership Objects

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

### Modify MyGame to Work with the Layer Component

The MyGame level implements the same functionality as in the previous project. The only difference is the delegation of layer management to the Layer component. The following description focuses only on function calls relevant to layer management.

1. Modify the unload() function to clean up the Layer.

unload() {

engine.layer.cleanUp();

engine.texture.unload(this.kMinionSprite);

engine.texture.unload(this.kBg);

engine.texture.unload(this.kBgNormal);

engine.texture.unload(this.kBgLayer);

engine.texture.unload(this.kBgLayerNormal);

engine.texture.unload(this.kMinionSpriteNormal);

}

1. Modify the init() function to add the game objects to the corresponding layers in the Layer component.

init() {

… identical to previous code …

// add to layer managers ...

engine.layer.addToLayer(engine.layer.eBackground, this.mBg);

engine.layer.addToLayer(engine.layer.eShadowReceiver, this.mBgShadow1);

engine.layer.addToLayer(engine.layer.eActors, this.mIllumMinion);

engine.layer.addToLayer(engine.layer.eActors, this.mLgtMinion);

engine.layer.addToLayer(engine.layer.eActors, this.mIllumHero);

engine.layer.addToLayer(engine.layer.eActors, this.mLgtHero);

engine.layer.addToLayer(engine.layer.eFront, this.mBlock1);

engine.layer.addToLayer(engine.layer.eFront, this.mBlock2);

engine.layer.addToLayer(engine.layer.eFront, this.mFront);

engine.layer.addToLayer(engine.layer.eHUD, this.mMsg);

engine.layer.addToLayer(engine.layer.eHUD, this.mMatMsg);

}

1. Modify the draw() function to rely on the Layer component for the actual drawings.

draw() {

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

this.mCamera.setViewAndCameraMatrix();

engine.layer.drawAllLayers(this.mCamera);

if (this.mShowParallaxCam) {

this.mParallaxCam.setViewAndCameraMatrix();

engine.layer.drawAllLayers(this.mParallaxCam);

}

}

1. Modify the update() function to rely on the Layer component for the actual update of all game objects.

update() {

this.mCamera.update(); // to ensure proper interpolated movement effects

this.mParallaxCam.update();

engine.layer.updateAllLayers();

… identical to previous code …

}

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 layers of the Layer component during init(), the draw() and update() functions of a game level can be much cleaner. The simpler and cleaner update() function is of special importance. Instead of being crowded with mundane game object update() function calls, this 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 moving cameras 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 Layer engine component is presented as a utility tool to relieve 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 be conveniently 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