CHAPTER 7

Manipulating the Camera

After completing this chapter, you will be able to:

- Implement operations that are commonly employed by manipulating a camera
- Interpolate values between old and new to create a smooth transition
- Understand how some motions or behaviors can be described by simple mathematic formulations
- Build games with multiple camera views
- Transform positions from the Canvas Coordinate space to the World Coordinate (WC) space
- Program with mouse input in a game environment with multiple cameras

Introduction

Your game engine is now capable of representing and drawing objects. With the basic abstraction mechanism introduced in the previous chapter, the engine can also support the interactions and behaviors of these objects. This chapter refocuses the attention on controlling and interacting with the `Camera` object that abstracts and facilitates the presentation of the game objects on the canvas. In this way, your game engine will be able to control and manipulate the presentation of visually pleasant game objects with well-structured behaviors.

Figure 7-1 presents a brief review of the `Camera` object abstraction that was introduced in Chapter 3. The `Camera` object allows the game programmer to define a World Coordinate (WC) window of the game world to be displayed into a viewport on the HTML canvas. The WC window is the bounds defined by a WC center and a dimension of $W_{wc} \times H_{wc}$. A viewport is a rectangular area on the HTML canvas with the lower-left corner located at (V_x, V_y) and a dimension of $W_v \times H_v$. The `Camera` object's `setUpViewProjection()` function encapsulates the details and enables the drawing of all game objects inside the WC window bounds to be displayed in the corresponding viewport.

Note In this book, the WC window or WC bounds are used to refer to the WC window bounds.

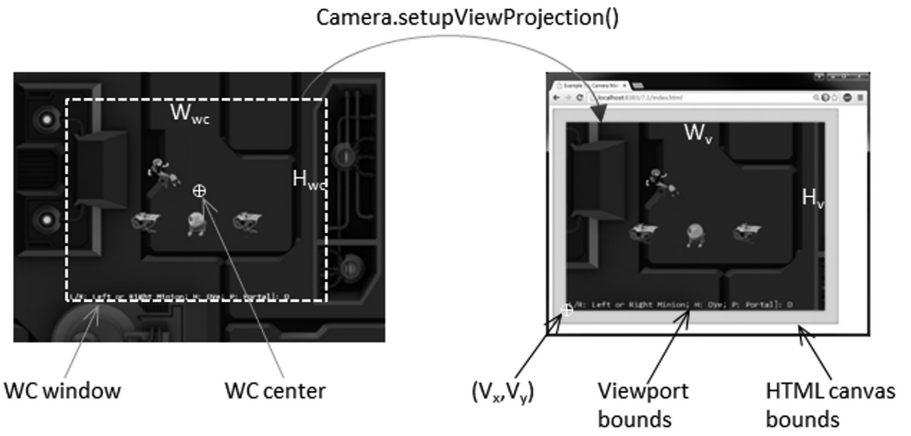

Figure 7-1. *Review of WC parameters that define a Camera object*

The `Camera` object abstraction allows the game programmer to ignore the details of WC bounds and the HTML canvas and focus on designing a fun and entertaining gameplay experience. Programming with a `Camera` object in a game level should reflect the use of a physical video camera in the real world. For example, you may want to pan the video camera to show your audiences the environment, you may want to attach the video camera on an actress and share her journey with your audience, or you may want to play the role of director and instruct the actors in your scene to stay within the visual ranges of the video camera. The distinct characteristics of these examples, such as panning or following a character's view, are the high-level functional specifications. Notice that in the real world you do not specify coordinate positions or bounds of windows.

This chapter introduces some of the most commonly encountered camera manipulation operations including clamping, panning, and zooming. Solutions in the form of interpolation will be derived to alleviate annoying or confusing abrupt transitions resulting from the manipulation of cameras. You will also learn about supporting multiple camera views in the same game level and working with mouse input.

Camera Manipulations

In a 2D world, you may want to clamp or restrict the movements of objects to be within the bounds of a camera, to pan or move the camera, or to zoom the camera into or away from specific areas. These high-level functional specifications can be realized by strategically changing the parameters of the `Camera` object: the WC center and the $W_{wc} \times H_{wc}$ of the WC window. The key is to create convenient functions for the game developers to manipulate these values in the context of the game. For example, instead of increasing/decreasing the width/height of the WC windows, zoom functions can be defined for the programmer.

The Camera Manipulations Project

This project demonstrates how to implement intuitive camera manipulation operations by working with the WC center, width, and height of the `Camera` object. You can see an example of this project running in Figure 7-2. The source code to this project is defined in the `Chapter7/7.1.CameraManipulations` folder.

Figure 7-2. *Running the Camera Manipulations project*

The controls of the project are as follows:

- *WASD keys*: Move the `Dye` character (the `Hero` object). Notice that the camera WC window updates to follow the `Hero` object when it attempts to move beyond 90 percent of the WC bounds.
- *Arrow keys*: Move the `Portal` object. Notice that the `Portal` object cannot move beyond 80 percent of the WC bounds.
- *L/R/P/H keys*: Select the `Left` minion, `Right` minion, `Portal` object, or `Hero` object to be the object in focus; the L/R keys also set the camera to focus on the `Left` or `Right` minion.
- *N/M keys*: Zoom into or away from the center of the camera.
- *J/K keys*: Zoom into or away while ensuring the constant relative position of the currently in-focus object. In other words, as the camera zooms, the positions of all objects will change except that of the in-focus object.

The goals of the project are as follows:

- To experience some of the common camera manipulation operations
- To understand the mapping from manipulation operations to the corresponding camera parameter values that must be altered
- To implement camera manipulation operations

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

Organize the Source Code

To accommodate the increase in functionality and the complexity of the `Camera` object, you will create a separate folder for storing the `Camera` object implementation and all supporting operations' source code files.

- Create a new folder called `Cameras` in `src/Engine`. Move the `Camera.js` file into this folder, and remember to update the reference path in `index.html`.

Support Clamping to Camera WC Bounds

Edit `Camera.js` and define a function to clamp the bounds associated with a `Transform` object to the camera WC bound.

```
Camera.prototype.clampAtBoundary = function (aXform, zone) {
    var status = this.collideWCBound(aXform, zone);
    if (status !== BoundingBox.eboundCollideStatus.eInside) {
        var pos = aXform.getPosition();
        if ((status & BoundingBox.eboundCollideStatus.eCollideTop) !== 0)
            pos[1] = (this.getWCCenter())[1] + (zone * this.getWCHeight() / 2)
                                             - (aXform.getHeight() / 2);
        if ((status & BoundingBox.eboundCollideStatus.eCollideBottom) !== 0)
            pos[1] = (this.getWCCenter())[1] - (zone * this.getWCHeight() / 2)
                                             + (aXform.getHeight() / 2);
        if ((status & BoundingBox.eboundCollideStatus.eCollideRight) !== 0)
            pos[0] = (this.getWCCenter())[0] + (zone * this.getWCWidth() / 2)
                                             - (aXform.getWidth() / 2);
        if ((status & BoundingBox.eboundCollideStatus.eCollideLeft) !== 0)
            pos[0] = (this.getWCCenter())[0] - (zone * this.getWCWidth() / 2)
                                             + (aXform.getWidth() / 2);
    }
    return status;
};
```

The `aXform` object can be the `Transform` of a `GameObject` or `Renderable` object. The `clampAtBoundary()` function ensures that the bounds of the `aXform` remain inside the WC bounds of the camera by clamping the `aXform` position. Once again, the `zone` variable defines a percentage of clamping for the WC bounds. For example, a 1.0 would mean clamping to the exact WC bounds, while a 0.9 means clamping to a bound that is 90 percent of the current WC window size. It is important to note that the `clampAtBoundary()` function operates only on bounds that collide with the camera WC bounds. For example, if the `aXform` object has its bounds that are completely outside of the camera WC bounds, it will remain outside.

Define Camera Manipulation Operations in Camera_Manipulation.js File

As discussed in the previous chapter, to maintain the readability of source code files, related functions of objects are grouped into separate source code files.

1. Create a new file in the `src/Engine/Cameras` folder and name it `Camera_Manipulation.js`.

2. Edit this file to define functions to pan, or move, the camera by an offset and to a new WC center.

```
Camera.prototype.panBy = function (dx, dy) {
    this.mWCCenter[0] += dx;
    this.mWCCenter[1] += dy;
};

Camera.prototype.panTo = function (cx, cy) {
    this.setWCCenter(cx, cy);
};
```

3. Define a function to pan the camera based on the bounds of a `Transform` object.

```
Camera.prototype.panWith = function (aXform, zone) {
    var status = this.collideWCBound(aXform, zone);
    if (status !== BoundingBox.eboundCollideStatus.eInside) {
        var pos = aXform.getPosition();
        var newC = this.getWCCenter();
        if ((status & BoundingBox.eboundCollideStatus.eCollideTop) !== 0)
            newC[1] = pos[1] + (aXform.getHeight() / 2) -
                      (zone * this.getWCHeight() / 2);
        if ((status & BoundingBox.eboundCollideStatus.eCollideBottom) !== 0)
            newC[1] = pos[1] - (aXform.getHeight() / 2) +
                      (zone * this.getWCHeight() / 2);
        if ((status & BoundingBox.eboundCollideStatus.eCollideRight) !== 0)
            newC[0] = pos[0] + (aXform.getWidth() / 2) -
                      (zone * this.getWCWidth() / 2);
        if ((status & BoundingBox.eboundCollideStatus.eCollideLeft) !== 0)
            newC[0] = pos[0] - (aXform.getWidth() / 2) +
                      (zone * this.getWCWidth() / 2);
    }
};
```

The `panWidth()` function is complementary to the `clampAtBoundary()` function, where instead of changing the `aXform` position, the camera is moved to ensure the proper inclusion of the `aXform` bounds. As in the case of the `clampAtBoundary()` function, the camera will not be changed if the `aXform` bounds are completely outside the tested WC bounds area.

4. Define functions to zoom the camera with respect to the center or a target position.

```
Camera.prototype.zoomBy = function (zoom) {
    if (zoom > 0)
        this.mWCWidth *= zoom;
};

Camera.prototype.zoomTowards = function (pos, zoom) {
    var delta = [];
    vec2.sub(delta, pos, this.mWCCenter);
    vec2.scale(delta, delta, zoom - 1);
    vec2.sub(this.mWCCenter, this.mWCCenter, delta);
    this.zoomBy(zoom);
};
```

The `zoomBy()` function zooms with respect to the center of the camera, and the `zoomTowards()` function zooms with respect to a world coordinate position. If the `zoom` variable is greater than 1, the WC window size becomes larger, and you will see more of the world and, thus, zoom out. The `zoom` value of less than 1 zooms in. Figure 7-3 shows the results of `zoom=0.5` for zooming with respect to the center of WC and with respect to the position of the `Hero` object.

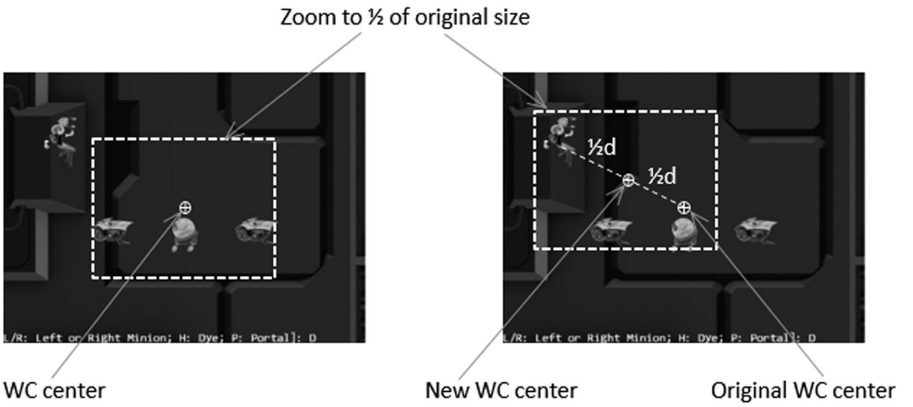

Figure 7-3. *Zooming toward the WC Center and toward a target position*

Manipulating the Camera in MyGame

There are two important functionality to be tested: panning and zooming. Once again, with the exception of the `update()` function, the majority of the code in the `MyGame.js` file is similar to the previous projects and are not repeated. The `update()` function is modified from the previous project to manipulate the camera.

```
MyGame.prototype.update = function () {
    // ... identical code to previous project ...
```

```
    // Pan camera to object
    if (gEngine.Input.isKeyClicked(gEngine.Input.keys.L)) {
        this.mFocusObj = this.mLMinion;
        this.mChoice = 'L';
        this.mCamera.panTo(this.mLMinion.getXform().getXPos(),
            this.mLMinion.getXform().getYPos());
    }
    if (gEngine.Input.isKeyClicked(gEngine.Input.keys.R)) {
        this.mFocusObj = this.mRMinion;
        this.mChoice = 'R';
        this.mCamera.panTo(this.mRMinion.getXform().getXPos(),
            this.mRMinion.getXform().getYPos());
    }
    if (gEngine.Input.isKeyClicked(gEngine.Input.keys.P)) {
        this.mFocusObj = this.mPortal;
        this.mChoice = 'P';
    }
    if (gEngine.Input.isKeyClicked(gEngine.Input.keys.H)) {
        this.mFocusObj = this.mHero;
        this.mChoice = 'H';
    }

    // zoom
    if (gEngine.Input.isKeyClicked(gEngine.Input.keys.N))
        this.mCamera.zoomBy(1 - zoomDelta);
    if (gEngine.Input.isKeyClicked(gEngine.Input.keys.M))
        this.mCamera.zoomBy(1 + zoomDelta);
    if (gEngine.Input.isKeyClicked(gEngine.Input.keys.J))
        this.mCamera.zoomTowards(this.mFocusObj.getXform().getPosition(), 1 - zoomDelta);
    if (gEngine.Input.isKeyClicked(gEngine.Input.keys.K))
        this.mCamera.zoomTowards(this.mFocusObj.getXform().getPosition(), 1 + zoomDelta);

    // interaction with the WC bounds
    this.mCamera.clampAtBoundary(this.mBrain.getXform(), 0.9);
    this.mCamera.clampAtBoundary(this.mPortal.getXform(), 0.8);
    this.mCamera.panWith(this.mHero.getXform(), 0.9);

    this.mMsg.setText(msg + this.mChoice);
};
```

In the previous code, the first four `if` statements select the in-focus object, where L and R keys also recenter the camera by calling the `panTo()` function with the appropriate WC positions. The second four `if` statements control the zoom, whether toward the WC center or toward the current in-focus object. The function ends with clamping the `Brain` and `Portal` objects to within 90 percent and 80 percent of the WC bounds, respectively, and panning the camera based on the transform (or position) of the `Hero` object.

You can now run the project and move the `Hero` object with the WASD keys. Move the `Hero` object toward the WC bounds to observe the camera being pushed. Continue pushing the camera with the `Hero` object; notice that because of the `clampAtBoundary()` function call, the `Portal` object will in turn be pushed such that it never leaves the camera WC bounds. Now press the L/R key to observe the camera center switching to the center on the `Left` or `Right` minion. The N/M keys demonstrate straightforward zooming with respect to the center. To experience zooming with respect to a target, move the `Hero` object toward the

top left of the canvas and then press the H key to select it as the zoom focus. Now, with your mouse pointer pointing at the head of the `Hero` object, you can press the K key to zoom out first and then the J key to zoom back in. Notice that as you zoom, all objects in the scene change positions except the areas around the `Hero` object. This is a convenient functionality to support zooming into a desired region of your game. You can experience moving the `Hero` object around while zooming into/away from it.

Interpolation

It is now possible to manipulate the camera based on high-level functions such as pan or zoom. However, the results are often sudden or even incoherent changes to the rendered image, which may result in annoyance or confusion. For example, in the previous project, the L or R key causes the camera to re-center with a simple assignment of new WC center values. The abrupt change in camera position results in the sudden appearance of a seemingly new game world. This sudden appearance of a completely different world is not only visually unpleasant; it can also cause player confusion.

When new values for camera parameters are available, instead of assigning them and causing an abrupt change, it is desirable to morph the values gradually from the old to the new over time, or *interpolate* the values. For example, as illustrated in Figure 7-4, at time t_1 a parameter with the old value is to be assigned a new one. In this case, instead of updating the value abruptly, an interpolation will change the value gradually over time. It will compute the intermediate results with decreasing values and complete the change to the new value at a later time t_2.

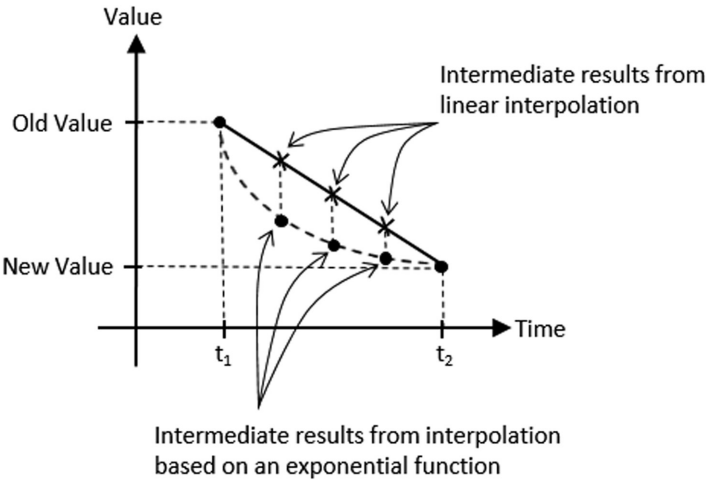

Figure 7-4. *Interpolating values based on linear and exponential functions*

Figure 7-4 shows that there is more than one way to interpolate values over time. For example, linear interpolation computes intermediate results according to the slope of the line connecting the old and new values. In contrast, an exponential function may compute intermediate results based on percentages from previous values. Linear interpolation interpolates from the old to the new values in constant steps, whereas the shown exponential function will result in rapid changes initially followed by much slower changes. In this way, with linear interpolation, a camera position would move from an old to new position with a constant speed similar to a constant-speed camera that is panning. In comparison, the interpolation based on the given exponential function would move the camera position rapidly at the beginning, with the motion slowing down quickly over time giving a sensation of focusing the camera on a new target.

This section introduces the `Interpolate` and `InterpolateVec2` utility objects to support smooth and gradual camera movements resulting from camera manipulation operations.

The Camera Interpolations Project

This project demonstrates the smoother and visually more pleasing interpolated results from camera manipulation operations. You can see an example of this project running in Figure 7-5. The source code to this project is defined in the `Chapter7/7.2.CameraInterpolations` folder.

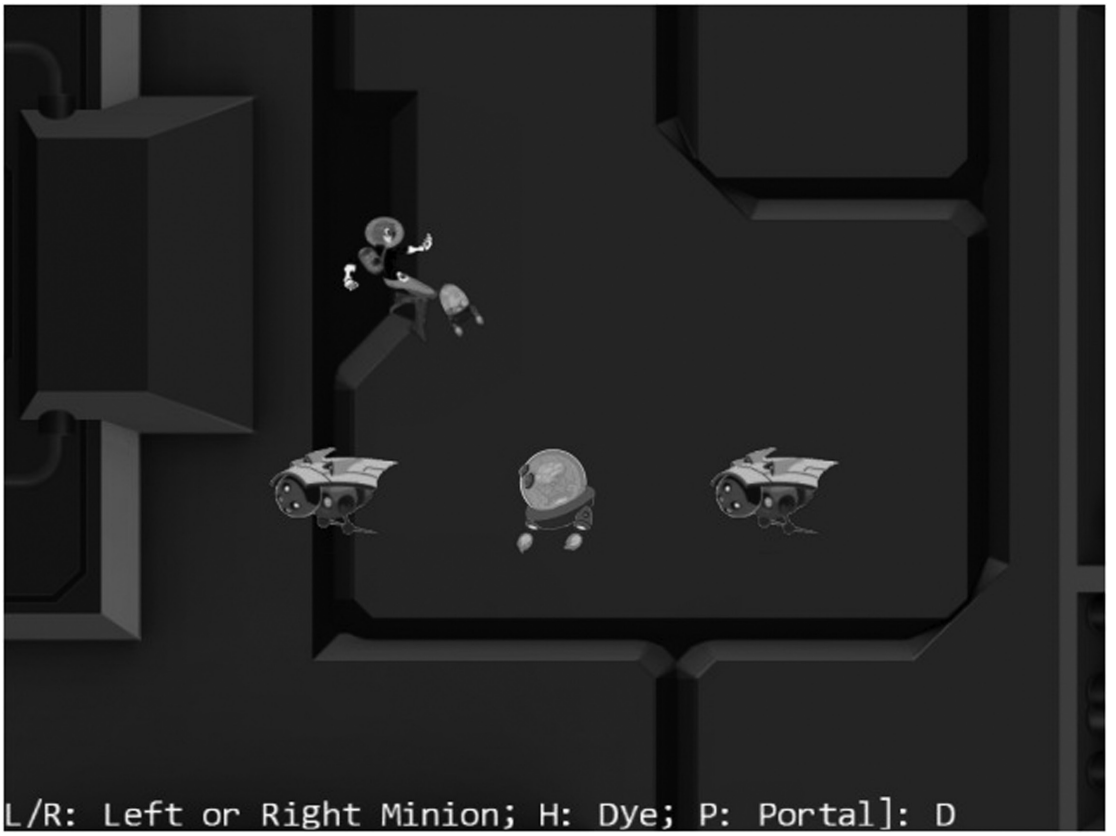

Figure 7-5. *Running the Camera Interpolations project*

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

- *WASD keys*: Move the `Dye` character (the `Hero` object). Notice that the camera WC window updates to follow the `Hero` object when it attempts to move beyond 90 percent of the WC bounds.
- *Arrow keys*: Move the `Portal` object. Notice that the `Portal` object cannot move beyond 80 percent of the WC bounds.

- *L/R/P/H keys*: Select the `Left` minion, `Right` minion, `Portal` object, or `Hero` object to be the object in focus; the L/R keys also set the camera to focus on the `Left` or `Right` minion.
- *N/M keys*: Zoom into or away from the center of the camera.
- *J/K keys*: Zoom into or away while ensuring constant relative position of the currently in-focus object. In other words, as the camera zooms, the positions of all objects will change except that of the in-focus object.

The goals of the project are as follows:

- To understand interpolated results between given values
- To implement interpolation supporting gradual camera parameter changes
- To experience interpolated changes in camera parameters

You can find the same external resource files as in the previous project in the `assets` folder.

Interpolation as an Utility

Similar to `Transform` objects supporting transformation functionality and `BoundingBox` objects supporting collision detection, an `Interpolate` object can be defined to support interpolation of values. To maintained source code organization, a new folder should be defined to store these utilities objects.

- Create the new folder called `src/Engine/Utils` and move the `Transform.js` and `BoundingBox.js` files into this folder. Remember to update the reference path in `index.html`.

The Interpolate Object

Define the `Interpolate` object to compute interpolation between two values.

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

```
function Interpolate(value, cycles, rate) {
    this.mCurrentValue = value;    // begin value of interpolation
    this.mFinalValue = value;      // final value of interpolation
    this.mCycles = cycles;
    this.mRate = rate;

    // if there is a new value to interpolate to, number of cycles
    left for interpolation
    this.mCyclesLeft = 0;
}
```

This object interpolates from `mCurrentValue` to `mFinalValue` in `mCycles`. During each update, intermediate results are computed based on the `mRate` increment on the difference between `mCurrentValue` and `mFinalValue`, as shown next.

2. Define the function that computes the intermediate results.

```
Interpolate.prototype._interpolateValue = function () {
    this.mCurrentValue = this.mCurrentValue +
                         this.mRate * (this.mFinalValue - this.mCurrentValue);
};
```

Note that the `_interpolateValue()` function computes a result that linearly interpolates between `mCurrentValue` and `mFinalValue`. However, since during each iteration the `mCurrentValue` is updated to the computed intermediate result, over the entire interpolation cycle `mCurrentValue` changes to the `mFinalValue` following an exponential function.

3. Define relevant getter and setter functions.

```
Interpolate.prototype.getValue = function () { return this.mCurrentValue; };
Interpolate.prototype.configInterpolation = function (stiffness, duration) {
    this.mRate = stiffness;
    this.mCycles = duration;
};

Interpolate.prototype.setFinalValue = function (v) {
    this.mFinalValue = v;
    this.mCyclesLeft = this.mCycles;     // will trigger interpolation
};
```

4. Define the function to trigger the computation of each intermediate result.

```
Interpolate.prototype.updateInterpolation = function () {
    if (this.mCyclesLeft <= 0)
        return;
    this.mCyclesLeft--;
    if (this.mCyclesLeft === 0)
        this.mCurrentValue = this.mFinalValue;
    else
        this._interpolateValue();
};
```

The InterpolateVec2 Object

Since many of the camera parameters are `vec2` objects (for example, the WC center position), it is important to generalize the `Interpolate` object to support the interpolation of `vec2` objects.

1. Create a new file in the `src/Engine/Utils/` folder and name it `InterpolateVec2.js`. Add the following to construct the object as a child of `Interpolate`:

```
function InterpolateVec2(value, cycle, rate) {
    Interpolate.call(this, value, cycle, rate);
}
gEngine.Core.inheritPrototype(InterpolateVec2, Interpolate);
```

2. Override the `_interpolateValue()` function to compute intermediate results for `vec2`.

```
InterpolateVec2.prototype._interpolateValue = function () {
    vec2.lerp(this.mCurrentValue, this.mCurrentValue, this.mFinalValue,
        this.mRate);
};
```

The `vec2.lerp()` function, defined in the `gl-matrix.js` file, computes for each of the x and y components of `vec2` with identical calculations as the `_interpolateValue()` function in the `Interpolate` object.

Represent Interpolated Intermediate Results with the CameraState Object

The state of a `Camera` object must be generalized to support gradual changes of interpolated intermediate results. The `CameraState` object is introduced to accomplish this purpose.

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

```
function CameraState(center, width) {
    this.kCycles = 300;  // number of cycles to complete the transition
    this.kRate = 0.1;    // rate of change for each cycle
    this.mCenter = new InterpolateVec2(center, this.kCycles, this.kRate);
    this.mWidth = new Interpolate(width, this.kCycles, this.kRate);
}
```

Observe that `mCenter` and `mWidth` are the only variables required to support camera panning (changing of `mCenter`) and zooming (changing of `mWidth`). Both of these variables are instances of the `Interpolate` class and are capable of interpolating and computing intermediate results that achieve gradual changes.

2. Define the setting and getting functions for the center and width.

```
CameraState.prototype.getCenter = function ()
    { return this.mCenter.getValue(); };
CameraState.prototype.getWidth = function ()
    { return this.mWidth.getValue(); };

CameraState.prototype.setCenter = function (c)
    { this.mCenter.setFinalValue(c); };
CameraState.prototype.setWidth = function (w)
    { this.mWidth.setFinalValue(w); };
```

3. Define the update function to trigger the interpolation computation.

```
CameraState.prototype.updateCameraState = function () {
    this.mCenter.updateInterpolation();
    this.mWidth.updateInterpolation();
};
```

4. Define the function to configure the interpolation.

```
CameraState.prototype.configInterpolation = function (stiffness, duration) {
    this.mCenter.configInterpolation(stiffness, duration);
    this.mWidth.configInterpolation(stiffness, duration);
};
```

The `stiffness` variable defines how quickly the interpolated intermediate results should converge to the final value. This is a number between 0 to 1, where a 0 means the convergence will never happen and a 1 means instantaneous convergence. The `duration` variable defines in how many update cycles the convergence should take place. This must be a positive integer value.

Integrate Interpolation into Camera Manipulation Operations

The `Camera` object must be modified to represent the WC center and width using the newly defined `CameraState` object.

1. Modify the `Camera` constructor to replace the center and width variables with an instance of `CameraState`.

```
function Camera(wcCenter, wcWidth, viewportArray) {
    // WC and viewport position and size
    this.mCameraState = new CameraState(wcCenter, wcWidth);
    this.mViewport = viewportArray;  // [x, y, width, height]
    this.mNearPlane = 0;
    this.mFarPlane = 1000;

    // transformation matrices
    this.mViewMatrix = mat4.create();
    this.mProjMatrix = mat4.create();
    this.mVPMatrix = mat4.create();

    // background color
    this.mBgColor = [0.8, 0.8, 0.8, 1]; // RGB and Alpha
}
```

The rest of the changes in the `Camera.js` file are trivial replacements of accessing and setting the center and width through the `this.mCameraState` instance variable and are not shown.

2. Now, edit the `Camera_Manipulation.js` file to define the functions to update and configure the interpolation functionality of the `CameraState` object.

```
Camera.prototype.update = function () {
    this.mCameraState.updateCameraState();
};
Camera.prototype.configInterpolation = function (stiffness, duration) {
    this.mCameraState.configInterpolation(stiffness, duration);
};
```

3. Modify the `panBy()` camera manipulation function to support the `CameraState` object as follows:

```
Camera.prototype.panBy = function (dx, dy) {
    var newC = vec2.clone(this.getWCCenter());
    this.mWCCenter[0] += dx;
    this.mWCCenter[1] += dy;
    this.mCameraState.setCenter(newC);
};
```

Similar to the `panBy()` function, the rest of the changes in the `Camera_Manipulation.js` file are trivial replacements to access and set WC window center and width via the `this.mCameraState` instance variable and are not shown.

Testing Interpolations in MyGame

Recall that the user controls of this project are identical to that from the previous project. The only difference is that in this project you can expect gradual and smooth transitions between different camera settings. To observe the proper interpolated results, the camera `update()` function must be invoked at each game scene update.

- Modify the `MyGame update()` function.

```
MyGame.prototype.update = function () {
    var zoomDelta = 0.05;
    var msg = "L/R: Left or Right Minion; H: Dye; P: Portal]: ";

    this.mCamera.update();  // for smoother camera movements

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

The previous call to update the camera for computing interpolated intermediate results is the only change in the `MyGame.js` file. You can now run the project and experiment with the smooth and gradual changes resulting from camera manipulation operations. Notice that the uninterrupted interpolated results mean the rendered image never abruptly changes and the sense of continuation in space from before and after the user's camera manipulation commands is preserved. You can try changing the `stiffness` and `duration` variables to better appreciate the different rates of interpolation convergence.

Camera Shake Effect

In video games, shaking the camera can be a convenient way to convey the significance or mightiness of events, such as the appearance of an enemy boss or the collisions between large objects. Similar to the interpolation of values, the camera shake movement can also be modeled by straightforward mathematical formulations.

Consider how a camera shake may occur in a real-life situation. For instance, while shooting with a video camera, say you are surprised or startled by someone or something could collide with you. Your reaction will probably be slight disorientation followed by quickly refocusing on the original targets of shooting. From the perspective of the camera, this reaction can be described as initial large displacements from the original camera center followed by quick adjustments to recenter the camera. Mathematically, as illustrated in Figure 7-6, damped simple harmonic motions, which can be represented with the damping of trigonometric functions, can be used to describe these types of displacements. However, in contrast to the slightly chaotic and unpredictable randomness of human reactions, straight mathematic formulation is precise, with perfect predictability. For this reason, a pseudorandom simple harmonic function will be introduced to model the camera shake effect.

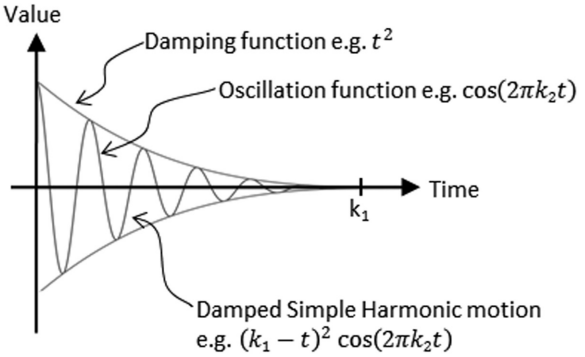

Figure 7-6. *The displacements of a damped simple harmonic motion*

The Camera Shake Project

This project demonstrates how to implement camera shake as a pseudorandom damped simple harmonic motion. You can see an example of this project running in Figure 7-7. This project is identical to the previous project except for the added command to create the camera shake effect. The source code to this project is defined in the `Chapter7/7.3.CameraShake` folder.

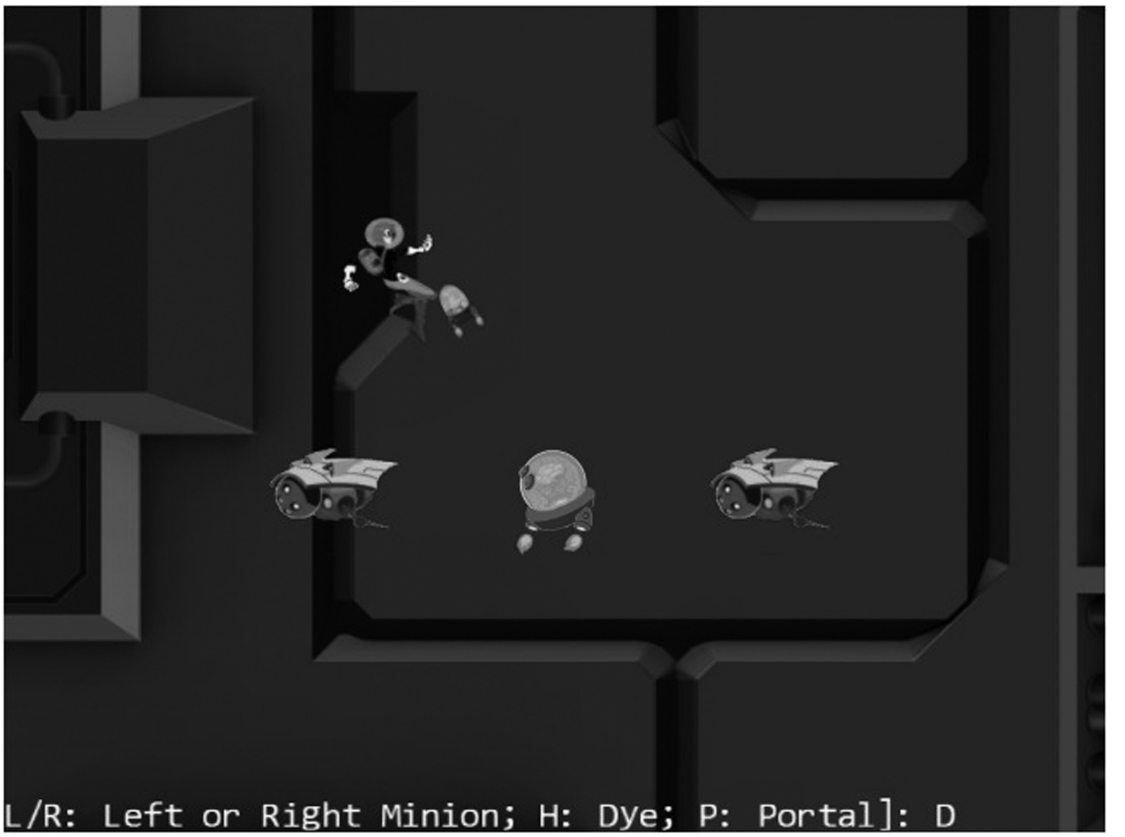

Figure 7-7. *Running the Camera Shake project*

The following is the new control of this project:

- *Q key*: Initiates the camera shake effect

The following controls are identical to the previous project:

- *WASD keys*: Move the `Dye` character (the `Hero` object). Notice that the camera WC window updates to follow the `Hero` object when it attempts to move beyond 90 percent of the WC bounds.
- *Arrow keys*: Move the `Portal` object. Notice that the `Portal` object cannot move beyond 80 percent of the WC bounds.
- *L/R/P/H keys*: Select the `Left` minion, `Right` minion, `Portal` object, or `Hero` object to be the object in focus; the L/R keys also set the camera to focus on the `Left` or `Right` minion.
- *N/M keys*: Zoom into or away from the center of the camera.
- *J/K keys*: Zoom into or away while ensuring constant relative position of the currently in-focus object. In other words, as the camera zooms, the positions of all objects will change except that of the in-focus object.

The goals of the project are as follows:

- To gain some insights into modeling displacements with simple mathematical functions
- To experience with the camera shake effect
- To implement camera shake as a pseudorandom damped simple harmonic motion

You can find the same external resource files as in the previous project in the `assets` folder.

Create the ShakePosition Class to Model the Shaking of a Position

As discussed, the shaking of a camera is described by the movements of its center position. For this reason, the implementation of shake effects should begin with modeling the shaking of a position.

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

```
// xDelta, yDelta: how large a shake
// shakeFrequency: how much movement
// shakeDuration: for how long in number of cycles
function ShakePosition(xDelta, yDelta, shakeFrequency, shakeDuration) {
    this.mXMag = xDelta;
    this.mYMag = yDelta;
    this.mCycles = shakeDuration; // number of cycles to complete the transition
    this.mOmega = shakeFrequency * 2 * Math.PI; // Converts frequency to radians
    this.mNumCyclesLeft = shakeDuration;
}
```

The `xDelta` and `yDelta` variables represent the initial displacements before damping, in WC space. The `shakeFrequency` parameter specifies how much to oscillate with a value of 1 representing one complete period of a cosine function. The `shakeDuration` parameter defines how long to shake, in units of game loop updates.

2. Define the damped simple harmonic motion.

```
ShakePosition.prototype._nextDampedHarmonic = function () {
    // computes (Cycles) * cos(  Omega * t )
    var frac = this.mNumCyclesLeft / this.mCycles;
    return frac * frac * Math.cos((1 - frac) * this.mOmega);
};
```

The `frac` variable is a ratio of the number of cycles left in the shake (`mNumCyclesLeft`) to the total number of cycles the camera should shake (`mCycles`). This value decreases from 1 to 0 as `mNumCyclesLeft` decreases from `mCycles` to 0. Figure 7-8 illustrates the damped simple harmonic motion that governs the camera shake displacements.

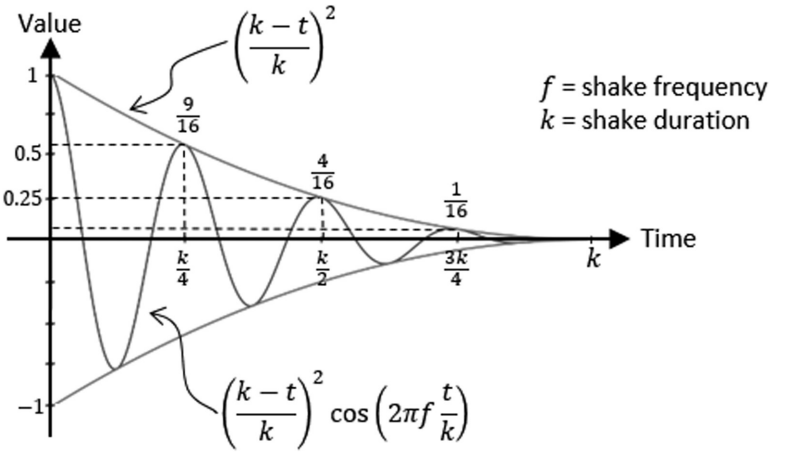

Figure 7-8. *The damped simple harmonic motion that specifies camera shake*

3. Define a function to check for the status of camera shake.

```
ShakePosition.prototype.shakeDone = function () {
    return (this.mNumCyclesLeft <= 0);
};
```

4. Define a function to trigger the calculation of shake displacements.

```
ShakePosition.prototype.getShakeResults = function () {
    this.mNumCyclesLeft--;
    var c = [];
    var fx = 0;
    var fy = 0;
    if (!this.shakeDone()) {
        var v = this._nextDampedHarmonic();
        fx = (Math.random() > 0.5) ? -v : v;
        fy = (Math.random() > 0.5) ? -v : v;
    }
    c[0] = this.mXMag * fx;
    c[1] = this.mYMag * fy;
    return c;
};
```

For the same `mNumCyclesLeft`, the call to the `_nextDampedHarmonic()` function will return the same value. The call to the `Math.random()` function introduces pseudorandomness into the x and y displacements.

Define the CameraShake Object to Abstract Camera Shaking Effect

With the defined ShakePoint object, it is convenient to trace the displacements of a pseudorandom damped simple harmonic motion. However, the Camera object requires an additional abstraction layer.

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

```
function CameraShake(state, xDelta, yDelta, shakeFrequency, shakeDuration) {
    this.mOrgCenter = vec2.clone(state.getCenter());
    this.mShakeCenter = vec2.clone(this.mOrgCenter);
    this.mShake = new ShakePosition(xDelta, yDelta, shakeFrequency,
        shakeDuration);
}
```

The CameraShake object receives the center of a camera via the CameraState parameter (state). This center is kept as the origin of shake displacements.

2. Define the function that triggers the displacement computation for accomplishing the shaking effect.

```
CameraShake.prototype.updateShakeState = function () {
    var s = this.mShake.getShakeResults();
    vec2.add(this.mShakeCenter, this.mOrgCenter, s);
};
```

3. Define a function to check whether the shaking effect has completed.

```
CameraShake.prototype.shakeDone = function () {
    return this.mShake.shakeDone();
};
```

4. Define the getter and setting functions for the camera center.

```
CameraShake.prototype.getCenter = function () { return this.mShakeCenter; };
CameraShake.prototype.setRefCenter = function (c) {
    this.mOrgCenter[0] = c[0];
    this.mOrgCenter[1] = c[1];
};
```

Modify the Camera to Support Shake Effect

With the proper CameraShake abstraction, supporting the shaking of the camera simply means defining, using, initiating, and updating the shake effect.

1. Modify the Camera constructor to define a CameraShake object.

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

```
        this.mViewport = viewportArray;  // [x, y, width, height]
        this.mNearPlane = 0;
        this.mFarPlane = 1000;

        // transformation matrices
        this.mViewMatrix = mat4.create();
        this.mProjMatrix = mat4.create();
        this.mVPMatrix = mat4.create();

        // background color
        this.mBgColor = [0.8, 0.8, 0.8, 1]; // RGB and Alpha
    }
```

2. Modify step B1 of the `setupViewProjection()` function to use the `CameraShake` object's center if it is defined.

```
    // Initializes the camera to begin drawing
    Camera.prototype.setupViewProjection = function () {
        var gl = gEngine.Core.getGL();
        // Step A1-A4: Set up and clear the Viewport

        // Identical to previous project ...

        // Step B: Set up the View-Projection transform operator
        // Step B1: define the view matrix
        var center = [];
        if (this.mCameraShake !== null) {
            center = this.mCameraShake.getCenter();
        } else {
            center = this.getWCCenter();
        }
        mat4.lookAt(this.mViewMatrix,
            [center[0], center[1], 10],  // WC center
            [center[0], center[1], 0],   //
            [0, 1, 0]);                  // orientation

        // Identical to previous project ...

        // Step B2: define the projection matrix
        // Step B3: concatenate view and project matrices
    };
```

3. Modify the `Camera_Manipulation.js` file to add support to initiate the shake effect.

```
    Camera.prototype.shake = function (xDelta, yDelta, shakeFrequency, duration) {
        this.mCameraShake  = new CameraShake(this.mCameraState, xDelta, yDelta,
                             shakeFrequency, duration);
    };
```

4. Continue working with the Camera_Manipulation.js file and modify the Camera object update() function to trigger a camera shake update if one is defined.

```
Camera.prototype.update = function () {
    if (this.mCameraShake !== null) {
        if (this.mCameraShake.shakeDone()) {
            this.mCameraShake = null;
        } else {
            this.mCameraShake.setRefCenter(this.getWCCenter());
            this.mCameraShake.updateShakeState();
        }
    }
    this.mCameraState.updateCameraState();
};
```

Testing the Camera Shake Effect in MyGame

When comparing to the previous project, MyGame.js file adds only a few lines in the update() function: to trigger camera shake effect with the Q key.

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

    if (gEngine.Input.isKeyClicked(gEngine.Input.keys.Q))
        this.mCamera.shake(-2, -2, 20, 30);
    this.mMsg.setText(msg + this.mChoice);
};
```

You can now run the project and experience with the pseudorandom damped simple harmonic motion that simulates the camera shake effect. Notice that the displacement of the camera center position will undergo interpolation and thus result in a smoother final shake effect. You can try changing the parameters to the mCamera.shake() function to experiment with different shake configurations. Recall that the first two parameters control the initial shake displacements, and the third and fourth parameters are the shakeFrequency and shakeDuration that control the shake amplitude and how long the shake effect should last.

Multiple Cameras

Video games often present the players with multiple views into the game world to communicate vital or interesting gameplay information, such as showing a mini-map to help the player navigate the world or prompting a view of the enemy boss to warn the player of what is to come.

In your game engine, the Camera object abstracts the WC window of the game world to draw from and the viewport for the area on the canvas to draw to. This effective abstraction supports the multiple view idea with multiple Camera objects. Each view in the game can simply be handled with a separate Camera object with distinct WC window and viewport configurations.

The Multiple Cameras Project

This project demonstrates how to represent multiple views into the game world with multiple `Camera` objects. You can see an example of this project running in Figure 7-9. The source code to this project is defined in the `Chapter7/7.4.MultipleCameras` folder.

Figure 7-9. *Running the Multiple Cameras project*

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

- *Q key*: Initiates the camera shake effect.
- *WASD keys*: Move the Dye character (the `Hero` object). Notice that the camera WC window updates to follow the `Hero` object when it attempts to move beyond 90 percent of the WC bounds.
- *Arrow keys*: Move the `Portal` object. Notice that the `Portal` object cannot move beyond 80 percent of the WC bounds.
- *L/R/P/H keys*: Select the `Left` minion, `Right` minion, `Portal` object, or `Hero` object to be the object in focus; the L/R keys also set the camera to focus on the `Left` or `Right` minion.

- *N/M keys*: Zoom into or away from the center of the camera.
- *J/K keys*: Zoom into or away while ensuring the constant relative position of the currently in-focus object. In other words, as the camera zooms, the positions of all objects will change except that of the in-focus object.

The goals of the project are as follows:

- To understand the camera abstraction for presenting views into the game world
- To experience working with multiple cameras in the same game level
- To appreciate the importance of interpolation configuration for cameras with a specific purpose

You can find the same external resource files as in the previous project in the `assets` folder.

Modify the Camera

The `Camera` object will be slightly modified to allow the drawing of the viewport with a bound. This would allow easy differentiation of camera views on the canvas.

1. Modify the `Camera` constructor to allow programmers to define a `bound`-number of pixels to surround the viewport of the camera.

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

    this.mViewport = [];  // [x, y, width, height]
    this.mViewportBound = 0;
    if (bound !== undefined) {
        this.mViewportBound = bound;
    }
    this.mScissorBound = [];  // use for bounds
    this.setViewport(viewportArray, this.mViewportBound);
    this.mNearPlane = 0;
    this.mFarPlane = 1000;

    // ... Identical to previous projects ...
}
```

By default, `bound` is assumed to be zero, and the camera will draw to the entire `mViewport`. Please refer to the `setViewport()` function that follows. A nonzero `bound` instructs the camera to leave `bound`-number of pixels that surround the camera `mViewport` in the background color, thereby allowing easy differentiation of multiple viewports on a canvas.

2. Define the `setViewport()` function.

```
Camera.prototype.setViewport = function (viewportArray, bound) {
    if (bound === undefined) {
        bound = this.mViewportBound;
    }
    // [x, y, width, height]
    this.mViewport[0] = viewportArray[0] + bound;
    this.mViewport[1] = viewportArray[1] + bound;
    this.mViewport[2] = viewportArray[2] - (2 * bound);
    this.mViewport[3] = viewportArray[3] - (2 * bound);
    this.mScissorBound[0] = viewportArray[0];
    this.mScissorBound[1] = viewportArray[1];
    this.mScissorBound[2] = viewportArray[2];
    this.mScissorBound[3] = viewportArray[3];
};
```

Recall that when setting the camera viewport, you invoke the `gl.scissor()` function to define an area to be cleared and the `gl.viewport()` function to identify the target area for drawing. Previously, the scissor and viewport bounds are identical. In this case, notice that the actual `mViewport` bounds are the `bound`-number of pixels smaller than the `mScissorBound`. These settings allow the `mScissorBound` to identify the area to be cleared to background color, while the `mViewport` bounds define the actual canvas area for drawing. In this way, the `bound`-number of pixels around the viewport will remain the background color.

3. Modify the `getViewport()` function to return the actual bounds that are reserved for this camera. In this case, it is the `mScissorBound` instead of the potentially smaller viewport bounds.

```
Camera.prototype.getViewport = function () {
    var out = [];
    out[0] = this.mScissorBound[0];
    out[1] = this.mScissorBound[1];
    out[2] = this.mScissorBound[2];
    out[3] = this.mScissorBound[3];
    return out;
};
```

4. Modify the `setupViewProjection()` function to bind scissor bounds with `mScissorBound` instead of the viewport bounds.

```
Camera.prototype.setupViewProjection = function () {
    var gl = gEngine.Core.getGL();
    // ... Identical to previous projects ...
```

```
    // Step A2: set up the corresponding scissor area to limit the clear area
    gl.scissor(this.mScissorBound[0],    // x pos of bottom-left of the area to be drawn
               this.mScissorBound[1],    // y pos of bottom-left of the area to be drawn
               this.mScissorBound[2],  // width of the area to be drawn
               this.mScissorBound[3]); // height of the area to be drawn

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

Testing Multiple Cameras in MyGame

The MyGame level must create multiple cameras, configure them properly, and draw each independently. For ease of demonstration, two new Camera objects will be created, one to focus on the Hero object and one to focus on the chasing Brain object. As in the previous examples, the implementation of the MyGame level is largely identical. In this example, some portions of the initialize(), draw(), and update() functions are modified to handle the multiple Camera objects and are highlighted here:

1. Modify the initialize() function to define and configure three Camera objects.

```
MyGame.prototype.initialize = function () {
    // Step A: set up the cameras
    this.mCamera = new Camera(
        vec2.fromValues(50, 36),  // 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]);

    this.mHeroCam = new Camera(
        vec2.fromValues(50, 30),  // will be updated at each cycle to point to hero
        20,
        [490, 330, 150, 150],
        2                         // viewport bounds
    );
    this.mHeroCam.setBackgroundColor([0.5, 0.5, 0.5, 1]);
    this.mBrainCam = new Camera(
        vec2.fromValues(50, 30),  // will be updated at each cycle to point to the brain
        10,
        [0, 330, 150, 150],
        2                         // viewport bounds
    );
    this.mBrainCam.setBackgroundColor([1, 1, 1, 1]);
    this.mBrainCam.configInterpolation(0.7, 10);

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

Both the mHeroCam and mBrainCam define a 2-pixel boundary for their viewports, with the mHeroCam boundary defined to be gray (the background color) and with mBrainCam white. Notice the mBrainCam object's stiff interpolation setting informing the camera interpolation to converge to new values in ten cycles.

2. Define a helper function to draw the world that is common to all three cameras.

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

3. Modify the MyGame object draw() function to draw all three cameras.

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

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

Take note of the mMsg object only being drawn to the mCamera, the main camera. For this reason, the echo message will appear only in the viewport of the main camera.

4. Modify the update() function to pan the mHeroCam and mBrainCam with the corresponding objects and to move the mHeroCam viewport continuously.

```
MyGame.prototype.update = function () {
    var zoomDelta = 0.05;
    var msg = "L/R: Left or Right Minion; H: Dye; P: Portal]: ";

    this.mCamera.update();  // for smoother camera movements
    this.mHeroCam.update();
    this.mBrainCam.update();

    // ... Identical to previous projects ...

    // set the hero and brain cams
    this.mHeroCam.panTo(this.mHero.getXform().getXPos(),
                        this.mHero.getXform().getYPos());
    this.mBrainCam.panTo(this.mBrain.getXform().getXPos(),
                         this.mBrain.getXform().getYPos());

    // Move the hero cam viewport just to show it is possible
    var v = this.mHeroCam.getViewport();
    v[0] += 1;
```

```
        if (v[0] > 500) {
            v[0] = 0;
        }
        this.mHeroCam.setViewport(v);

        this.mMsg.setText(msg + this.mChoice);
    };
```

You can now run the project and notice the three different viewports showing on the HTML canvas. The 2-pixel-wide bounds around the `mHeroCam` and `mBrainCam` viewports allow easy visual parsing of the three views. Observe that the `mBrainCam` viewport is drawn on top of the `mHeroCam`. This is because in the `MyGame` object `draw()` function, the `mBrainCam` is drawn last. The last drawn always appears on the top. You can move the `Hero` object to observe that `mHeroCam` follows the hero and experience the smooth interpolated results of panning the camera.

Now try changing the parameters to the `mBrainCam.configInterpolation()` function to generate smoother interpolated results, such as by setting the stiffness to 0.1 and the duration to 100 cycles. Note how it appears as though the camera is constantly trying to catch up to the `Brain` object. In this case, the camera needs a stiff interpolation setting to ensure the main object remains in the center of the camera view. For a much more drastic and fun effect, you can try setting `mBrainCam` to have much smoother interpolated results, such as with a stiffness value of 0.01 and a duration of 200 cycles. With these values, the camera can never catch up to the `Brain` object and will appear as though it is wandering aimlessly around the game world.

Mouse Input Through Cameras

The mouse is a pointing input device that reports position information in the Canvas Coordinate space. Recall that the Canvas Coordinate space is simply a measurement of pixel offsets along the x/y-axes with respect to the lower-left corner of the canvas. The game engine defines and works with the WC space where all objects and measurements are specified in WC. For the game engine to work with the reported mouse position, this position must be transformed from Canvas Coordinate space to WC.

The drawing on the left of Figure 7-10 shows an example of a mouse position located at (mouseX, mouseY) on the canvas. The drawing on the right of Figure 7-10 shows that when a viewport with the lower-left corner located at (V_x,V_y) and a dimension of $W_v \times H_v$ is defined within the canvas, the same (mouseX, mouseY) position can be represented as a position in the viewport as (mouseDCX, mouseDCY) where:

- mouseDCX = mouseX–V_x
- mouseDCY = mouseY–V_y

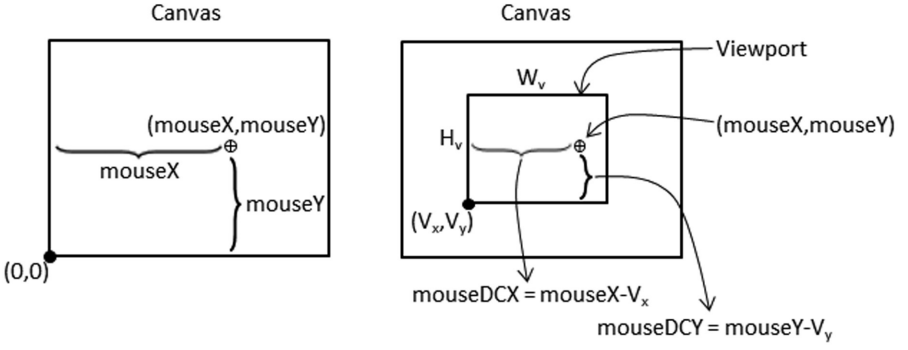

Figure 7-10. *Mouse position on canvas and viewport*

In this way, (mouseDCX, mouseDCY) is the offset from the (V_x, V_y), the lower-left corner of the viewport.

The drawing on the left of Figure 7-11 shows that the Device Coordinate (DC) space defines a pixel position within a viewport with offsets measured with respect to the lower-left corner of the viewport. For this reason, the DC space is also referred to as the pixel space. The computed (mouseDCX, mouseDCY) position is an example of a position in DC space. The drawing on the right of Figure 7-11 shows that this position can be transformed into the WC space with the lower-left corner located at (minWCX, minWCY) and a dimension of $W_{wc} \times H_{wc}$ according to these formulae:

- $\text{mouseWCX} = \text{minWCX} + \left(\text{mouseDCX} \times \frac{W_{wc}}{W_v} \right)$
- $\text{mouseWCY} = \text{minWCY} + \left(\text{mouseDCY} \times \frac{H_{wc}}{H_v} \right)$

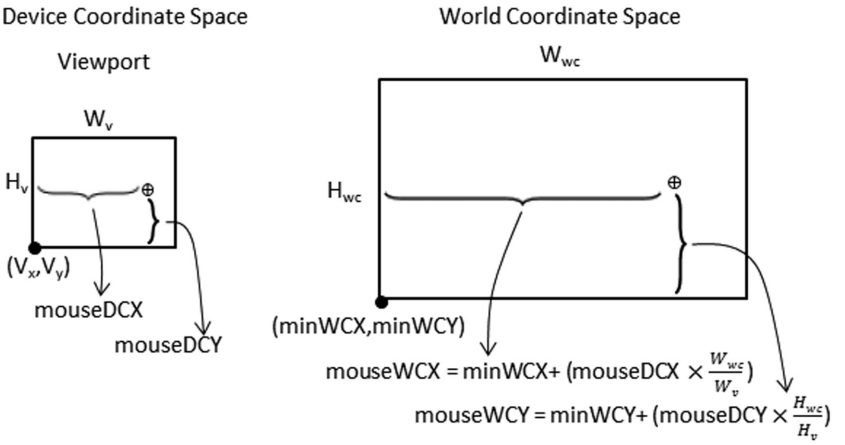

Figure 7-11. *Mouse position in viewport DC space and WC space*

With the knowledge of how to transform positions from the Canvas Coordinate space to the WC space, it is now possible to implement mouse input support in the game engine.

The Mouse Input Project

This project demonstrates mouse input support in the game engine. You can see an example of this project running in Figure 7-12. The source code to this project is defined in the `Chapter7/7.5.MouseInput` folder.

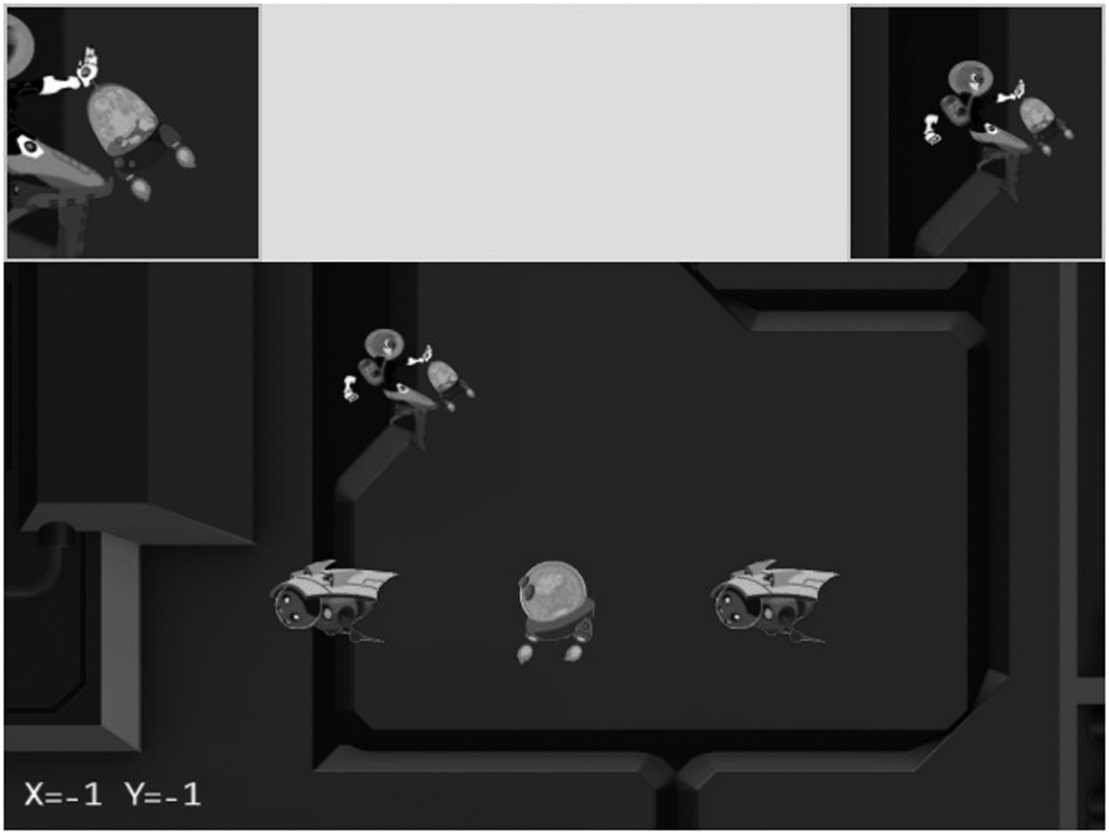

Figure 7-12. *Running the Mouse Input project*

The new controls of this project are as follows:

- *Left mouse button pressed in the main Camera view*: Drags the `Portal` object
- *Middle mouse button pressed in the HeroCam view*: Drags the `Hero` object
- *Right or middle mouse button pressed in any view*: Hides/shows the `Portal` object

The following controls are identical to the previous project:

- *Q key*: Initiates the camera shake effect
- *WASD keys*: Move the `Dye` character (the `Hero` object) and push the camera WC bounds
- *Arrow keys*: Move the `Portal` object
- *L/R/P/H keys*: Select the in-focus object with L/R keys refocusing the camera to the `Left` or `Right` minion
- *N/M and J/K keys*: Zoom into or away from the center of the camera, or the in-focus object

The goals of the project are as follows:

- To understand the Canvas Coordinate space to WC space transform
- To appreciate mouse clicks are specific to individual viewports
- To implement transformation between coordinate spaces and support mouse input

You can find the same external resource files as in the previous project in the `assets` folder.

Modify gEngine_Core to Pass Canvas ID to Input Component

To receive mouse input information, the `gEngine_Input` component needs to have access to the HTML canvas. This can be accomplished by editing the `Engine_Core.js` file and modifying the `intializeEngineCore()` function to pass the HTML canvas ID to the engine input component.

```
var initializeEngineCore = function (htmlCanvasID, myGame) {
    _initializeWebGL(htmlCanvasID);
    gEngine.VertexBuffer.initialize();
    gEngine.Input.initialize(htmlCanvasID);
    gEngine.AudioClips.initAudioContext();

    // Inits DefaultResoruces, when done, call startScene(myGame).
    gEngine.DefaultResources.initialize(function () { startScene(myGame); });
```

Implement Mouse Support in gEngine_Input

Follow these steps:

1. Edit `Engine_Input.js` and define a set of constants to represent the three |mouse buttons.

   ```
   var kMouseButton = {
       Left: 0,
       Middle: 1,
       Right: 2
   };
   ```

2. Define the variables to support mouse input.

   ```
   // Support mouse
   var mCanvas = null;
   var mButtonPreviousState = [];
   var mIsButtonPressed = [];
   var mIsButtonClicked = [];
   var mMousePosX = -1;
   var mMousePosY = -1;
   ```

Similar to keyboard input, mouse button states are arrays of three Boolean elements, each representing the state of the three mouse buttons.

3. Define the mouse movement event handler.

```
var _onMouseMove = function (event) {
    var inside = false;
    var bBox = mCanvas.getBoundingClientRect();

    // In Canvas Space now. Convert via ratio from canvas to client.
    var x = Math.round((event.clientX - bBox.left) *
        (mCanvas.width / bBox.width));
    var y = Math.round((event.clientY - bBox.top) *
        (mCanvas.width / bBox.width));

    if ((x >= 0) && (x < mCanvas.width) && (y >= 0) &&
        (y < mCanvas.height)) {
        mMousePosX = x;
        mMousePosY = mCanvas.height - 1 - y;
        inside = true;
    }
    return inside;
};
```

Notice that the mouse event handler transforms a raw pixel position into the Canvas Coordinate space by first checking whether the position is within the bounds of the canvas and then flipping the y position such that the displacement is measured with respect to the lower-left corner.

4. Define the mouse button press handler to record the button event.

```
var _onMouseDown = function (event) {
    if (_onMouseMove(event))
        mIsButtonPressed[event.button] = true;
};
```

5. Define t.e button release handler to facilitate the detection of a mouse button click event.

```
var _onMouseUp = function (event) {
    _onMouseMove(event);
    mIsButtonPressed[event.button] = false;
};
```

6. Modify the `initialize()` function to receive the `canvasID` parameter and initialize mouse event handlers.

```
var initialize = function (canvasID) {
    // Keyboard support
    // ... Identical to previous projects ...
```

```
    // Mouse support
    for (i = 0; i < 3; i++) {
        mButtonPreviousState[i] = false;
        mIsButtonPressed[i] = false;
        mIsButtonClicked[i] = false;
    }
    window.addEventListener('mousedown', _onMouseDown);
    window.addEventListener('mouseup', _onMouseUp);
    window.addEventListener('mousemove', _onMouseMove);
    mCanvas = document.getElementById(canvasID);
};
```

7. Modify the `update()` function to process mouse button state changes in a similar fashion to the keyboard.

```
var update = function () {
    var i;
    for (i = 0; i < kKeys.LastKeyCode; i++) {
        mIsKeyClicked[i] = (!mKeyPreviousState[i]) && mIsKeyPressed[i];
        mKeyPreviousState[i] = mIsKeyPressed[i];
    }
    for (i = 0; i < 3; i++) {
        mIsButtonClicked[i] = (!mButtonPreviousState[i]) &&
                              mIsButtonPressed[i];
        mButtonPreviousState[i] = mIsButtonPressed[i];
    }
};
```

8. Define the functions to retrieve mouse position and mouse button states.

```
var isButtonPressed = function (button) {
    return mIsButtonPressed[button];
};

var isButtonClicked = function (button) {
    return mIsButtonClicked[button];
};
var getMousePosX = function () { return mMousePosX; };
var getMousePosY = function () { return mMousePosY; };
```

9. When all is done, please remember to update the public interface.

Modify the Camera to Support Transformation to WC Space

The `Camera` object encapsulates the WC window and viewport and thus should be responsible for transforming mouse positions.

1. Create a new file in the `src/Engine/Camera/` folder and name it `Camera_Input.js`. This file will contain the mouse input support functionality.

2. Define functions to transform mouse positions from Canvas Coordinate space to the DC space, as illustrated in Figure 7-10.

```
Camera.prototype._mouseDCX = function () {
    return gEngine.Input.getMousePosX() - this.mViewport[Camera.eViewport.eOrgX];
};
Camera.prototype._mouseDCY = function () {
    return gEngine.Input.getMousePosY() - this.mViewport[Camera.eViewport.eOrgY];
};
```

3. Define a function to determine whether a given mouse position is within the viewport bounds of the camera.

```
Camera.prototype.isMouseInViewport = function () {
    var dcX = this._mouseDCX();
    var dcY = this._mouseDCY();
    return ((dcX >= 0) && (dcX < this.mViewport[Camera.eViewport.eWidth]) &&
            (dcY >= 0) && (dcY < this.mViewport[Camera.eViewport.eHeight]));
};
```

4. Define the functions to transform the mouse position into the WC space, as illustrated in Figure 7-11.

```
Camera.prototype.mouseWCX = function () {
    var minWCX = this.getWCCenter()[0] - this.getWCWidth() / 2;
    return minWCX + (this._mouseDCX() * (this.getWCWidth() /
                     this.mViewport[Camera.eViewport.eWidth]));
};

Camera.prototype.mouseWCY = function () {
    var minWCY = this.getWCCenter()[1] - this.getWCHeight() / 2;
    return minWCY + (this._mouseDCY() * (this.getWCHeight() /
                     this.mViewport[Camera.eViewport.eHeight]));
};
```

Testing the Mouse Input in MyGame

The main functionality to be tested includes the ability to detect which view should receive the mouse input, proper mouse button state identification, and correct transformed WC position. As in previous few examples, the `MyGame.js` implementation is largely similar to previous projects. In this case, only the `update()` function contains noteworthy changes that work with the new mouse input functionality.

```
MyGame.prototype.update = function () {

    // ... Identical to previous projects ...
```

```
    msg = "";
    // testing the mouse input
    if (gEngine.Input.isButtonPressed(gEngine.Input.mouseButton.Left)) {
        msg += "[L Down]";
        if (this.mCamera.isMouseInViewport()) {
            this.mPortal.getXform().setXPos(this.mCamera.mouseWCX());
            this.mPortal.getXform().setYPos(this.mCamera.mouseWCY());
        }
    }

    if (gEngine.Input.isButtonPressed(gEngine.Input.mouseButton.Middle)) {
        if (this.mHeroCam.isMouseInViewport()) {
            this.mHero.getXform().setXPos(this.mHeroCam.mouseWCX());
            this.mHero.getXform().setYPos(this.mHeroCam.mouseWCY());
        }
    }
    if (gEngine.Input.isButtonClicked(gEngine.Input.mouseButton.Right))
        this.mPortal.setVisibility(false);

    if (gEngine.Input.isButtonClicked(gEngine.Input.mouseButton.Middle))
        this.mPortal.setVisibility(true);

    msg += " X=" + gEngine.Input.getMousePosX() + " Y="
          + gEngine.Input.getMousePosY();
    this.mMsg.setText(msg);
};
```

The `camera.isMouseInViewport()` condition is checked when the viewport context is important, as in the case of a left mouse button press in the main camera view or a middle mouse button press in the `mHeroCam` view. This is in contrast to a right or middle mouse button click for setting the visibility of the `Portal` object. The controls will be executed no matter where the mouse position is.

You can now run the project and verify the correctness of the transformation to WC space. Press and drag with left mouse button in the main view or middle mouse button in the `mHeroCam` view to observe the accurate movement of the corresponding object centers to the mouse position. Left or middle mouse button drags in the wrong views have no effect on the corresponding objects; for example, a left mouse button drag in the `mHeroCam` or `mBrainCam` view has no effect on the `Portal` object. The right or middle mouse button click properly controls the visibility of the `Portal` object, independent of the location of the mouse pointer. Be aware that the browser maps the right mouse button click to a default pop-up menu. For this reason, you should avoid working with right mouse button clicks in your games.

Summary

This chapter was about controlling and interacting with the `Camera` object. You have learned about the most common camera manipulation operations including clamping, panning, and zooming. These operations are implemented in the game engine with utility functions that map the high-level specifications to actual WC window bound parameters. The sudden, often annoying, and potentially confusing results from camera manipulations are mitigated with the introduction of interpolation. Through the implementation of the camera shake effect, you have discovered that some movements can be modeled by simple mathematical formulations. You have experienced the importance of effective `Camera` object abstraction in supporting multiple camera views. The last section guided you through the implementation of transforming a mouse position from the Canvas Coordinate space to the WC space.

In Chapter 5, you found out how to represent and draw an object with a visually appealing image and control the animation of this object. In Chapter 6, you read about how to define an abstraction to encapsulate the behaviors of an object and the fundamental support to detect collisions between objects. This chapter is about the "directing" of these objects: what should be visible, where the focus should be, how much of the world to show, how to ensure smooth transition between foci, and how to receive input from the mouse. With these capabilities, you now have a well-rounded game engine framework, from representing and drawing objects to modeling and managing the behaviors of these objects to controlling what, where, and how to show the game world.

The following chapters will continue to examine object appearance and behavior at more advanced levels, including creating lighting and illumination effects in a 2D world and simulating and integrating behaviors based on simple classical mechanics.

Game Design Considerations

Now that you've learned the basics of object interaction, it's a good time to start thinking about creating your first simple game mechanic to begin gaining insight into the logical conditions and rules that constitute well-formed gameplay experiences. Many designers approach game creation from the top down (meaning they start with an idea for an implementation of a specific genre, like a real-time strategy, tower defense, or role-playing game), which is common in an industry such as video games where the creators typically spend quite a bit of time as content consumers before transitioning into content makers. Game studios reinforce this top-down design approach, assigning new staff to work under seasoned leads to learn best practices for whatever genre that particular studio works in. This approach has proven effective for training designers who can competently iterate on known genres, but it's not always the best path to develop well-rounded creators who can design new systems and mechanics from the ground up.

This begs the question, "What makes gameplay well-formed?" At a fundamental level, a game is an interactive experience where rules must be learned and applied to achieve a specified outcome. All games must meet this minimum criteria, including card and board, physical, video, and other game types. Taking things a step further, a *good* game is an interactive experience with rules people *enjoy* learning and applying to achieve an outcome they're *invested* in. There's quite a bit to unpack in that brief definition, of course, but as a general rule, players will enjoy a game more when the rules are discoverable, are consistent, and make logical sense and when the outcome feels like a satisfactory reward for mastering those rules. This definition applies to both individual game mechanics as well as entire game experiences. To use a metaphor, it can be helpful to think of game designs as built with letters (interactions) that form words (mechanics) that form sentences (levels) that ultimately form readable content (genres). Most new designers attempt to write novels before they know the alphabet, and everyone has played games where the mechanics and levels felt at best like sentences written with poor grammar and at worst like unsatisfying, random jumbles of unintelligible letters.

Over the next several chapters you'll learn about more advanced features in 2D game engines, including simulations of illumination and physical behaviors. You'll also be introduced to a set of design techniques enabling you to deliver a complete and well-formed game level, integrating these techniques and utilizing more of the nine elements of game design discussed in Chapter 4 in an intentional way and working from the ground up to deliver a unified experience. In the earliest stages of design exploration, it's often helpful to focus only on creating and refining the basic game mechanics and interaction model; at this stage, try to avoid thinking about setting, meta-game, systems design, and the like (these will be folded into the design as it progresses).

The first design technique is a simple exercise that allows you to start learning the game design alphabet: an "escape the room" scenario with one simple mechanic, where you must accomplish a task in order to unlock a door and claim a reward. One goal of this exercise is to begin developing insight into how to create well-formed and logical rules that are discoverable and consistent, which is much easier to accomplish when the tasks are separated into basic interactions. You've already explored the beginnings of potential rule-based scenarios in earlier projects. Recall the Keyboard Support project from Chapter 4, which suggested you might have players move a smaller square completely into the boundary of a larger square in order to trigger some kind of behavior. How might that single interaction (or "letter of the game alphabet") combine to form a game mechanic (or "word") that makes sense? Figure 7-13 sets the stage for the locked room sandbox.

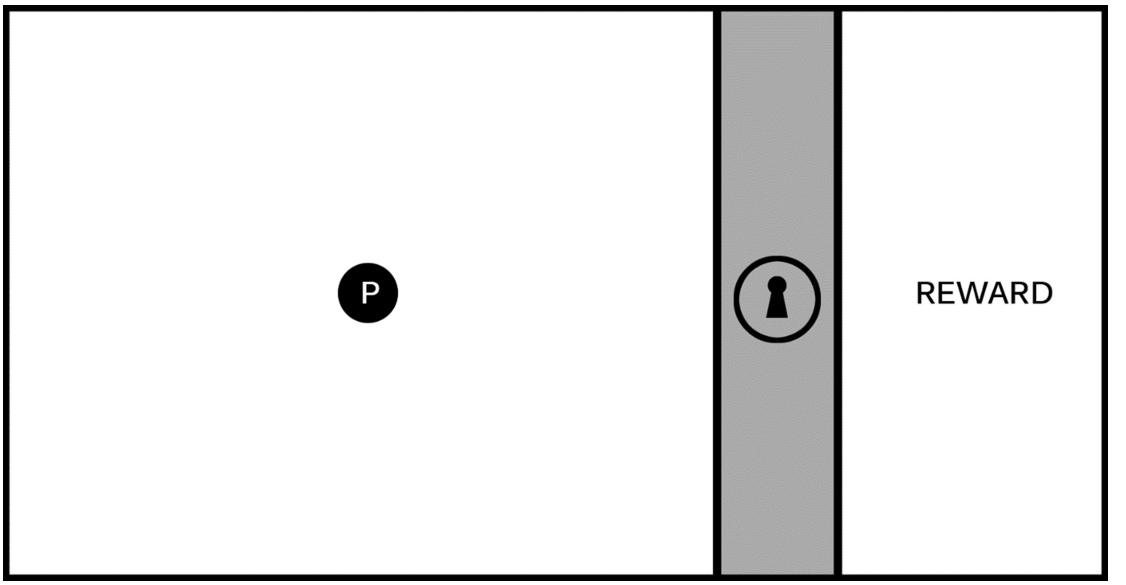

Figure 7-13. *The image represents a single game screen divided into three areas. a playable area on the left with a hero character (the circle marked with a P), an impassable barrier marked with a lock icon, and a reward area on the right*

The screen represented in Figure 7-13 is a useful starting place when exploring new mechanics. The goal for this exercise is to create one logical challenge that a player must complete to unlock the barrier and reach the reward. The specific nature of the task can be based on a wide range of elemental mechanics. It can involve jumping or shooting, puzzle solving, narrative situations, and the like. The key is to keep this first iteration *simple* (this first challenge should have a limited number of components contributing to the solution) and *discoverable* (players must be able to experiment and learn the rules of engagement so they can intentionally solve the challenge). You'll add complexity and interest to the mechanic in later iterations, and you'll see how elemental mechanics can be evolved to support many kinds of game types.

Figure 7-14 sets the stage for a logical relationship mechanic where players must interact with objects in the environment to learn the rules.

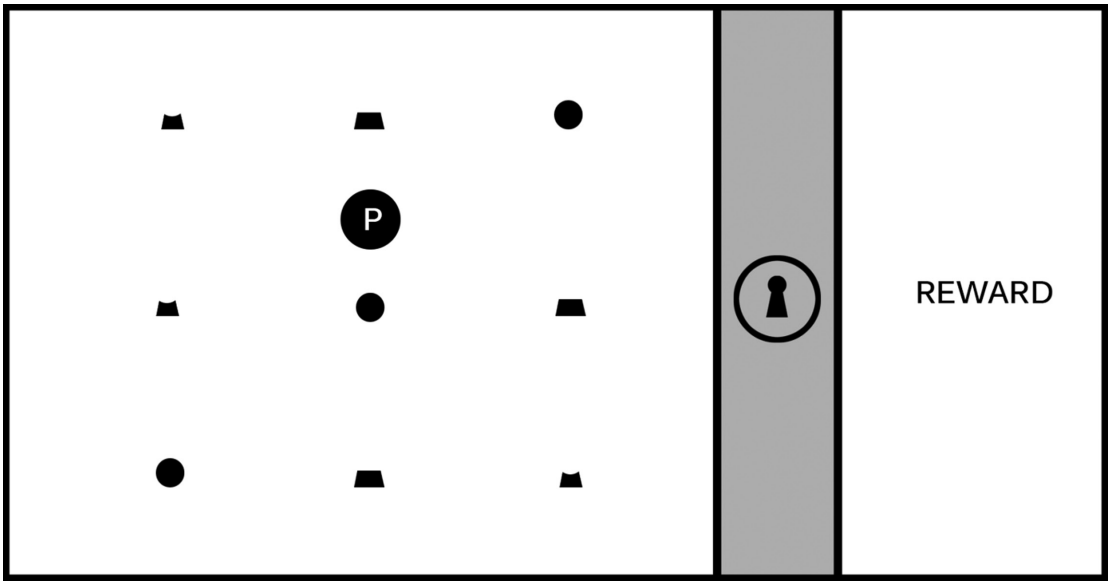

Figure 7-14. *The game screen is populated with an assortment of individual objects*

It's not immediately apparent just by looking at Figure 7-14 what the player needs to do to unlock the barrier, so they must experiment in order to learn the rules by which the game world operates; it's this experimentation that forms the core element of a game mechanic driving players forward through the level, and the mechanic will be more or less satisfying based on the discoverability and logical consistency of its rules. In this example, imagine that as the player moves around the game screen, they notice that when the hero character interacts with an object, it always "activates" with a highlight, as shown in Figure 7-15, and sometimes causes a section of the lock icon and one-third of the ring around the lock icon to glow. Some shapes, however, will not cause the lock and ring to glow when activated, as shown in Figure 7-16.

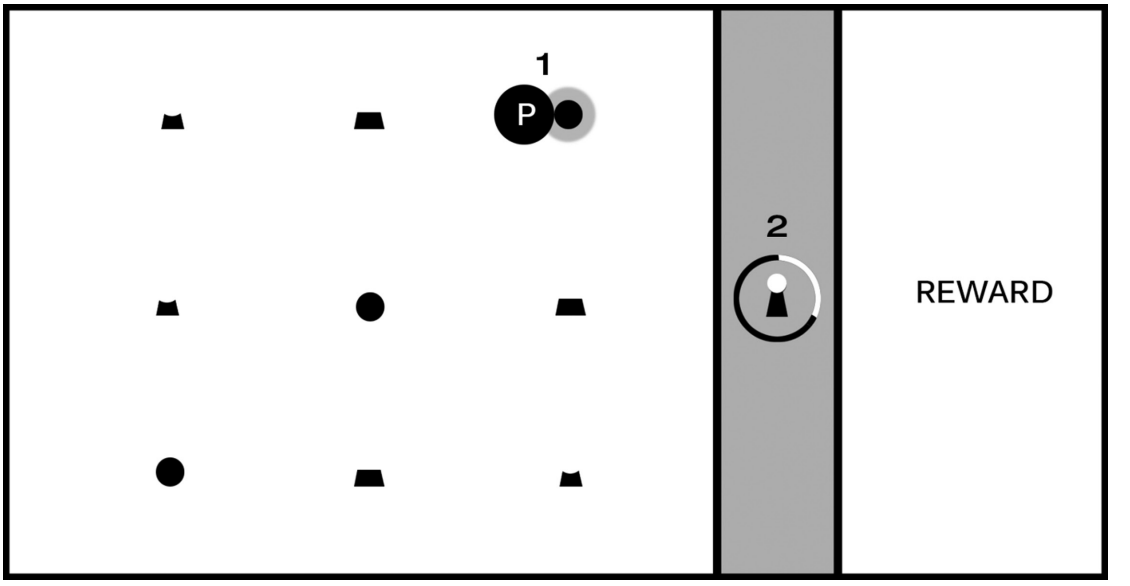

Figure 7-15. *As the player moves the hero character around the game screen, the shapes "activate" with a highlight (#1); activating certain shapes causes a section of the lock and one-third of the surrounding ring to glow (#2)*

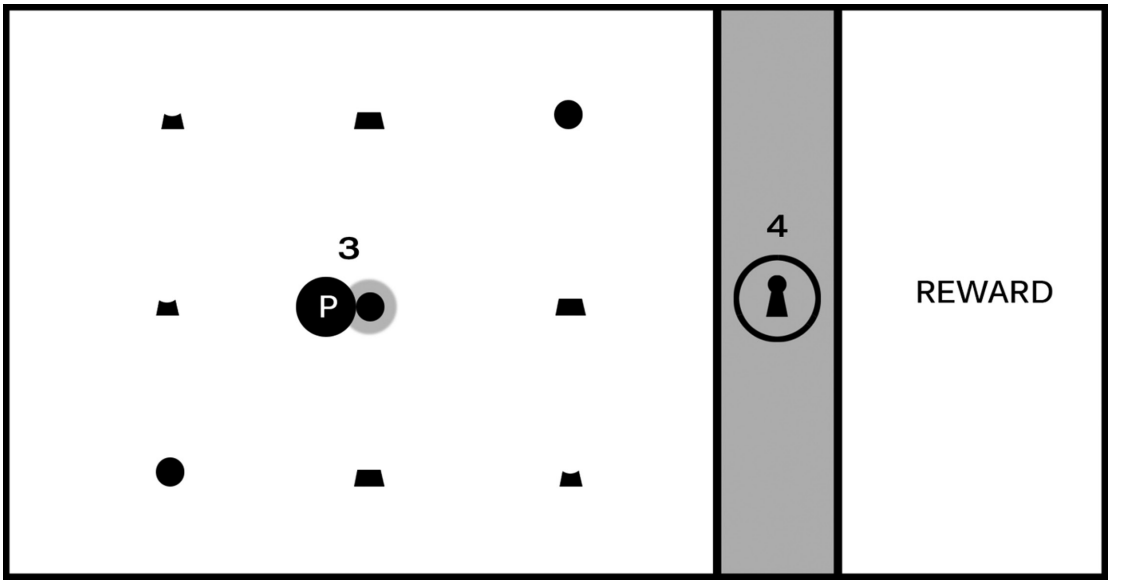

Figure 7-16. *Activating some shapes (#3) will not cause the lock and ring to glow (#4)*

Astute players will learn the rules for this puzzle fairly quickly. Can you guess what they might be just from looking at Figures 7-15 and 7-16? If you're feeling stuck, Figure 7-17 should provide enough information to solve the puzzle.

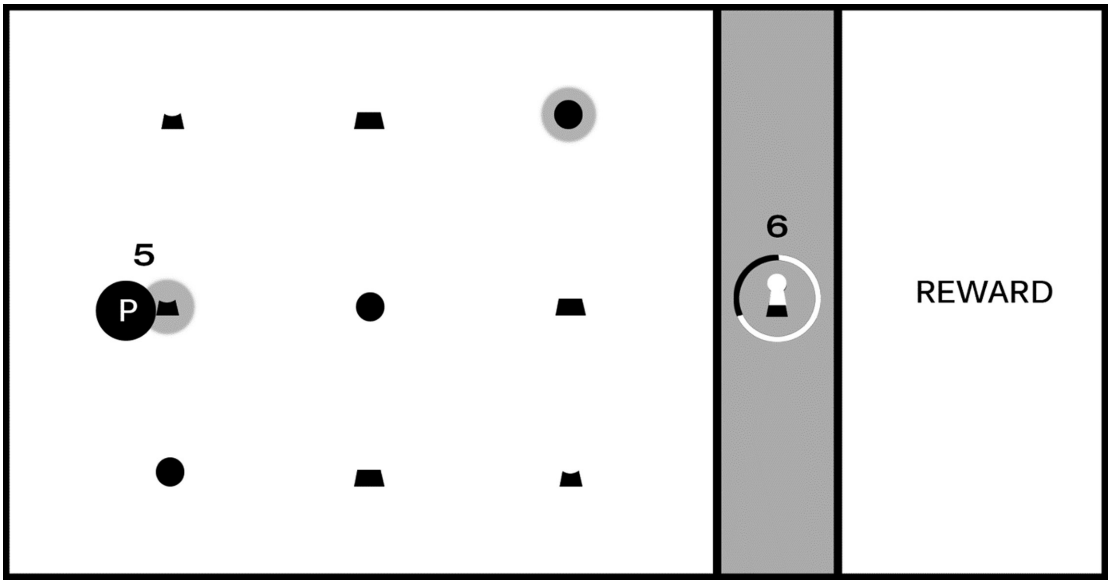

Figure 7-17. *After the first object was activated (the circle in the upper-right) and caused the top section of the lock and first third of the ring to glow, as shown in Figure 7-15, the second object in the correct sequence (#5) caused the middle section of the lock and second third of the ring to glow (#6)*

You (and players) should now have all required clues to learn the rules of this mechanic and solve the puzzle. There are three instances, each with a unique shape, that the player can interact with, and there is only one instance of each shape per row. The shapes are representations of the top, middle, and bottom of the lock icon, and as shown in Figure 7-15, activating the circle shape caused the corresponding section of the lock to glow. Figure 7-16, however, did not cause the corresponding section of the lock to glow, and the difference is the "hook" for this mechanic: sections of the lock must be activated in the correct relative position: top in the top row, middle in the middle row, bottom on the bottom (you might also choose to require that players activate them in the correct sequence starting with the top section, although that requirement is not discoverable just from looking at Figures 7-15 to 7-17).

You've now created a well-formed and logically consistent, if simple, puzzle, with all of the elements needed to build a larger and more ambitious level. This unlocking sequence is an abstract mechanic. The game screen is intentionally devoid of game setting, visual style, or genre alignment at this stage of design because I don't want to burden the exploration with any preconceived expectations. It will benefit you as a designer to spend time exploring the mechanic in its purest form before burdening it with a specific implementation, and you'll likely be surprised at the directions these simple mechanics will take you as you build them out.

Simple mechanics like the previous, which can be described as "complete a multistage task in the correct sequence to achieve a goal," are featured in many kinds of games; any game that requires players to collect parts of an object and combine them in an inventory to complete a challenge, for example, utilizes this mechanic. Individual mechanics can also be combined with other mechanics and game features to form compound elements that add complexity and flavor to the game experience.

The camera exercises in this chapter provide good examples for how you might add interest to a single mechanic. The Simple Camera Manipulations project, for example, demonstrates one method for advancing game action. Imagine in the previous example that after a player receives the reward for unlocking the barrier, the player moves the hero object to the right side of the screen and advances to a new "room" or area. Now imagine how gameplay would change if the camera advanced the screen at a fixed rate when the level started; the addition of autoscrolling changes this mechanic considerably because the player must solve the puzzle and unlock the barrier before the advancing barrier pushes the player off the screen. The first instance creates a leisurely puzzle-solving game experience, while the latter increases the tension considerably by giving the player a limited amount of time to complete each screen. In an autoscrolling implementation, how might you lay out the game screen to ensure the player had sufficient time to learn the rules and solve the puzzle?

The Multiple Cameras project can be especially useful as a mini-map that provides information about places in the game world not currently displayed on the game screen; in the case of the previous exercise, imagine that the locked barrier appeared somewhere else in the game world other than the player's current screen and that a secondary camera acting as a mini-map displayed a zoomed-out view of the entire game world map. As the game designer, you might want to let the player know when they complete a task that allows them to advance and provide information about where they need to go next, so in this case you might flash a beacon on the mini-map calling attention to the barrier that just unlocked and showing the player where to go. In the context of the "game design is like a written language" metaphor, adding additional elements like camera behavior to enhance or extend a simple mechanic is one way to begin forming words from the game design alphabet.

A game designer's primary challenge is to create scenarios that require clever experimentation while maintaining logical consistency; it's perfectly fine to frustrate players by creating devious scenarios requiring creative problem solving (we call this "good" frustration), but it's not typically sound design to frustrate players by creating scenarios that are logically inconsistent and make players feel that they succeeded in the challenge only by trial-and-error or luck ("bad" frustration). Think back to the games you've played that have resulted in bad frustration: where did they go wrong, and what might the designers have done to improve the experience?

The locked room scenario is a useful design tool because it forces you to construct basic mechanics, but you might be surprised at the variety of scenarios that can result from this exercise. Try a few different approaches to the locked room puzzle and see where the design process takes you, but keep it simple. For now, stay focused on one-step events to unlock the room that require players to learn only one rule. You'll revisit this exercise in the next chapter and begin creating more ambitious mechanics that add additional challenges.