# 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.

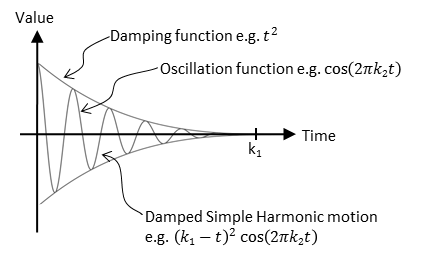


Figure 7-6. The displacements of a damped simple harmonic motion

## 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.camera\_shake 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 pseudo-random damped simple harmonic motion

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

### Abstract the Shake Behavior

The ability to shake the camera is a common yet dynamic behavior in many games and allows the engine to provide more options and functionality to the game client. However, it is important to recognize that the shake behavior can apply to more than just the camera. That is, the shaking effect on the Camera can effectively be distilled down to the shaking of the Camera’s position over time (update cycles) and thus any object in the engine with a position can potentially benefit from the shake behavior. This fact allows you to define a new set of utilities which support the shaking behavior of a position. These utilities are as follows:

* Oscillate: the abstract base class that implements simple harmonic oscillation on a value over time
* Shake: an extension of the Oscillation class that introduces randomness to the magnitudes of the oscillations to simulate the shake effect on a value
* ShakeVec2: an extension of the Shake class that expands the Shake behavior to a position or vec2 by including the y-value

#### Create the Oscillate Class to Model Simple Harmonic Motion

As discussed, the shaking of a camera is described by the movements of its center position. For this reason, the implementation of the shaking behavior starts with the ability to oscillate a value overtime.

1. Create a new file in the src/engine/utils folder and name it oscillate.js. Define a new base class named Oscillate and add the following code to construct the object:

class Oscillate {

constructor(delta, frequency, duration) {

this.mMag = delta;

this.mCycles = duration; // number of cycles to complete the transition

this.mOmega = frequency \* 2 \* Math.PI; // Converts frequency to radians

this.mNumCyclesLeft = duration;

}

// … implementation to follow …

}

export default Oscillate;

The delta variable represents the initial displacements before damping, in WC space. The frequency parameter specifies how much to oscillate with a value of 1 representing one complete period of a cosine function. The duration parameter defines how long to oscillate, in units of game loop updates.

1. Define the damped simple harmonic motion.

\_nextDampedHarmonic() {

// computes (Cycles) \* cos( Omega \* t )

let 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 oscillation (mNumCyclesLeft) to the total number of cycles the value should oscillate (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 value oscillations over time.

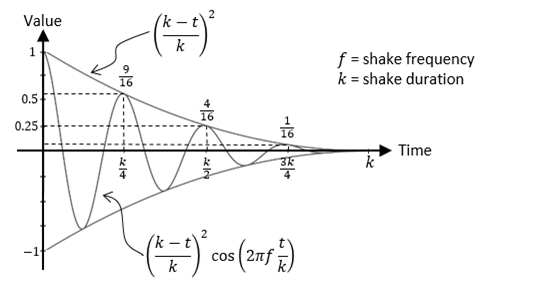


Figure 7-8. The damped simple harmonic motion that specifies value ocillation

1. TEMP TEXT

// local/protected methods

\_nextValue() {

return (this.\_nextDampedHarmonic());

}

1. Define a function to check for the status of the oscillation.

done() {

return (this.mNumCyclesLeft <= 0);

}

1. Define a function to trigger the calculation of value oscillations.

getNext() {

this.mNumCyclesLeft--;

let v = 0;

if (!this.done()) {

v = this.\_nextValue();

}

return (v \* this.mMag);

}

For the same mNumCyclesLeft, the call to the \_nextValue() function will return the same value.

1. TEMP TEXT

reStart() {

this.mNumCyclesLeft = this.mCycles;

}

#### Create the Shake Class to Model the Oscillation of a Value

With the ability to oscillate a value overtime now implemented in the Oscillate base class you can now expand this behavior to simulate the shaking effect by simply introducing pseudo-randomness into the effect.

1. Create a new file in the src/engine/utils folder and name it shake.js. Define a new class named Shake that extends the Oscillate base class and add the following code to construct the object:

"use strict";

import Oscillate from "./oscillate.js";

class Shake extends Oscillate {

constructor(delta, frequency, duration) {

super(delta, frequency, duration);

}

// … implementation to follow …

}

export default Shake;

1. Randomize the magnitude and sign of the oscillation to simulate a shake effect. This can be achieved with the following code:

\_nextValue() {

let v = this.\_nextDampedHarmonic();

let fx = (Math.random() > 0.5) ? -v : v;

return fx;

}

#### Create the ShakeVec2 Class to Model the Shaking of a Position

You can now expand the shake effect to support a position or vec2 by simply introducing two-dimensions to the pseudo-random oscillation. More specifically, the x-value and y-value.

1. Create a new file in the src/engine/utils folder and name it shake\_vec2.js. Define a new class named ShakeVec2 that extends the Shake class and add the following code to construct the object:

class ShakeVec2 extends Shake {

constructor(deltas, freqs, duration) {

super(deltas[1], freqs[1], duration); // super is shake in y-direction

this.xShake = new Shake(deltas[0], freqs[0], duration);

}

// … implementation to follow …

}

export default ShakeVec2;

1. TEMP TEXT

reStart() {

super.reStart();

this.xShake.reStart();

}

1. TEMP TEXT

getNext() {

let x = this.xShake.getNext();

let y = super.getNext();

return [x, y];

}

1. TEMP TEXT

### Define the CameraShake Object by Extending the ShakeVec2 Object

With the defined ShakeVec2 object, it is now more convenient to implement the displacements of a pseudo-random damped simple harmonic motion on the Camera. However, the Camera object requires an additional abstraction layer.

1. Create a new file in the src/engine/cameras folder and name it camera\_shake.js. Add the following code to construct the object:

import ShakeVec2 from "../utils/shake\_vec2.js";

class CameraShake {

// state is the CameraState to be shaked.

constructor(state, deltas, freqs, shakeDuration) {

this.mOrgCenter = vec2.clone(state.getCenter());

this.mShakeCenter = vec2.clone(this.mOrgCenter);

this.mShake = new ShakeVec2(deltas, freqs, shakeDuration);

}

}

export default CameraShake;

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

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

update() {

let delta = this.mShake.getNext();

vec2.add(this.mShakeCenter, this.mOrgCenter, delta);

}

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

done() {

return this.mShake.done();

}

1. TEMP TEXT

reShake() {

this.mShake.reStart();

}

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

getCenter() { return this.mShakeCenter; }

setRefCenter(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 camera\_main.js and camera\_manipulation.js to import camera\_shake.js as shown:

import CameraShake from "./camera\_shake.js";

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

constructor(wcCenter, wcWidth, viewportArray) {

this.mCameraState = new CameraState(wcCenter, wcWidth);

this.mCameraShake = null;

this.mViewport = viewportArray; // [x, y, width, height]

// Camera transform operator

this.mCameraMatrix = mat4.create();

// background color

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

}

1. Modify step B of the setViewAndCameraMatrix() function to use the CameraShake object’s center if it is defined.

setViewAndCameraMatrix() {

// … identical to previous project …

// Step B: Compute the Camera Matrix

let center = [];

if (this.mCameraShake !== null) {

center = this.mCameraShake.getCenter();

} else {

center = this.getWCCenter();

}

// … identical to previous project …

}

1. Modify the camera\_manipulation.js file to add support to initiate and reinitiate the shake effect.

Camera.prototype.shake = function (deltas, freqs, duration) {

this.mCameraShake = new CameraShake(this.mCameraState, deltas, freqs, duration);

}

// Re-cause the shake

Camera.prototype.reShake = function () {

let success = (this.mCameraShake !== null);

if (success)

this.mCameraShake.reShake();

return success;

}

1. 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.done()) {

this.mCameraShake = null;

} else {

this.mCameraShake.setRefCenter(this.getWCCenter());

this.mCameraShake.update();

}

}

this.mCameraState.update();

}

### Testing the Camera Shake Effect in MyGame

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

1. TEMP TEXT

init() {

// … identical to previous project …

// create a oscillate object to simulate motion

this.mBounce = new engine.Oscillate(2, 6, 120); // delta, freq, duration

}

1. TEMP TEXT

update() {

// … identical to previous project …

if (engine.input.isKeyClicked(engine.input.keys.Q)) {

if (!this.mCamera.reShake())

this.mCamera.shake([6, 1], [10, 3], 60);

// also re-start bouncing effect

this.mBounce.reStart();

}

if (!this.mBounce.done()) {

let d = this.mBounce.getNext();

this.mHero.getXform().incXPosBy(d);

}

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

}

You can now run the project and experience with the pseudo-random 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 frequency and duration 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.multiple\_cameras 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.

constructor(wcCenter, wcWidth, viewportArray, bound) {

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);

// Camera transform operator

this.mCameraMatrix = mat4.create();

// background color

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

}

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.

1. Define the setViewport() function.

setViewport(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.

1. Define 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.

getViewport() {

let out = [];

out[0] = this.mScissorBound[0];

out[1] = this.mScissorBound[1];

out[2] = this.mScissorBound[2];

out[3] = this.mScissorBound[3];

return out;

}

1. Modify the setViewAndCameraMatrix() function to bind scissor bounds with mScissorBound instead of the viewport bounds.

setViewAndCameraMatrix() {

let gl = glSys.get();

// … identical to previous project …

// Step A2: set up the corresponding scissor area to limit the clear area

gl.scissor(this.mScissorBound[0], // x position of bottom-left corner of the area to be drawn

this.mScissorBound[1], // y position of bottom-left corner 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.

init() {

// Step A: set up the cameras

this.mCamera = new engine.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]);

// sets the background to gray

this.mHeroCam = new engine.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 engine.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.configLerp(0.7, 10);

// … identical to previous project …

}

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.

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

\_drawCamera(camera) {

camera.setViewAndCameraMatrix();

this.mBg.draw(camera);

this.mHero.draw(camera);

this.mBrain.draw(camera);

this.mPortal.draw(camera);

this.mLMinion.draw(camera);

this.mRMinion.draw(camera);

}

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

draw() {

// Step A: clear the canvas

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

// Step B: Draw with all three cameras

this.\_drawCamera(this.mCamera);

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

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.

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

update() {

let zoomDelta = 0.05;

let 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 project …

// 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

let 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 () and a dimension of is defined within the canvas, the same (mouseX, mouseY) position can be represented as a position in the viewport as (mouseDCX, mouseDCY) where:



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

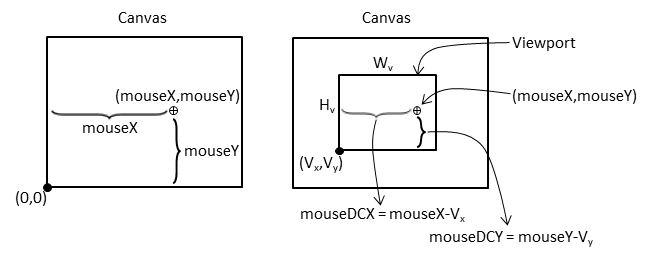


Figure 7-10. Mouse position on canvas and viewport

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 according to these formulae:



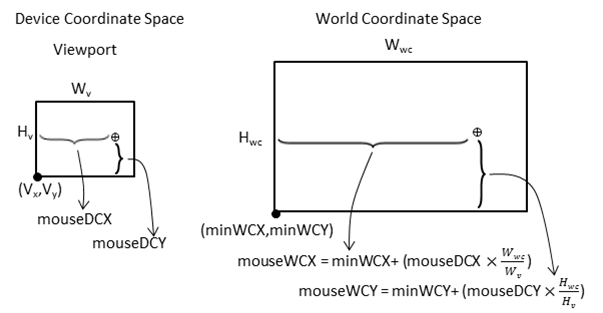


Figure 7-11. Mouse position in viewport DC space and WC space

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.mouse\_input folder.

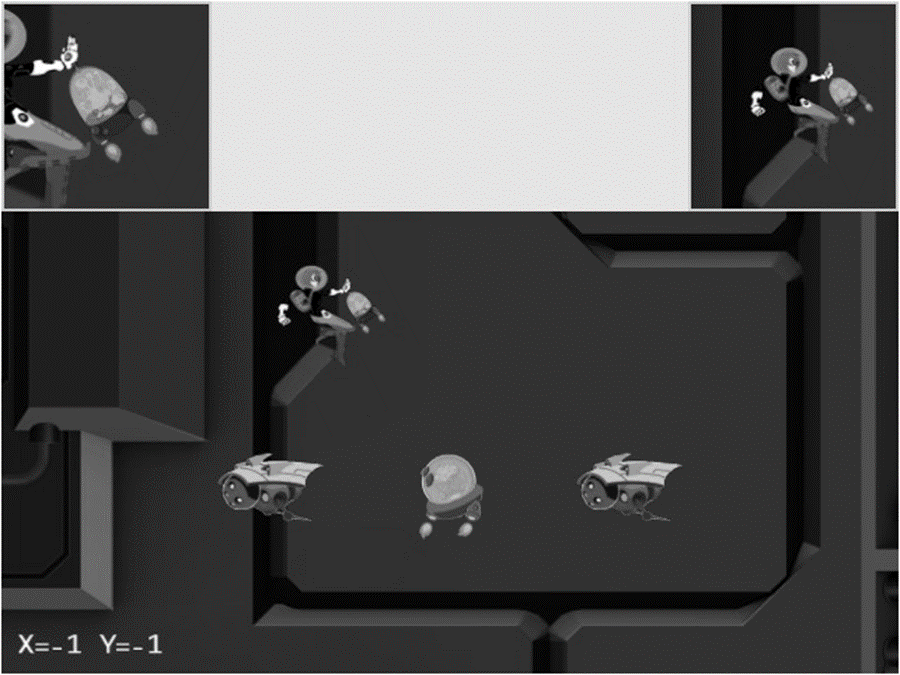


Figure 7-12. Running the Mouse Input project

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.

### Implement Mouse Support in input.js

Similar to the keyboard input you should add mouse support to the input module in input.js.

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

// mouse button enums

const eMouseButton = Object.freeze({

eLeft: 0,

eMiddle: 1,

eRight: 2

});

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

let mCanvas = null;

let mButtonPreviousState = [];

let mIsButtonPressed = [];

let mIsButtonClicked = [];

let mMousePosX = -1;

let mMousePosY = -1;

1. Define the mouse movement event handler.

function onMouseMove(event) {

let inside = false;

let bBox = mCanvas.getBoundingClientRect();

// In Canvas Space now. Convert via ratio from canvas to client.

let x = Math.round((event.clientX - bBox.left) \* (mCanvas.width / bBox.width));

let y = Math.round((event.clientY - bBox.top) \* (mCanvas.height / bBox.height));

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.

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

function onMouseDown(event) {

if (onMouseMove(event)) {

mIsButtonPressed[event.button] = true;

}

}

1. Define the mouse button release handler to facilitate the detection of a mouse button click event.

function onMouseUp(event) {

onMouseMove(event);

mIsButtonPressed[event.button] = false;

}

1. Modify the init() function to receive the canvasID parameter and initialize mouse event handlers.

function init(canvasID) {

let i;

// keyboard support

// … identical to previous project …

// 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);

}

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

function update() {

let i;

// update keyboard input state

// … Identical to previous projects …

// update mouse input state

for (i = 0; i < 3; i++) {

mIsButtonClicked[i] = (!mButtonPreviousState[i]) && mIsButtonPressed[i];

mButtonPreviousState[i] = mIsButtonPressed[i];

}

}

1. TEMP TEXT

function isButtonPressed(button) {

return mIsButtonPressed[button];

}

function isButtonClicked(button) {

return mIsButtonClicked[button];

}

function getMousePosX() { return mMousePosX; }

function getMousePosY() { return mMousePosY; }

1. TEMP TEXT

export {

keys, eMouseButton,

init, cleanUp, update,

// keyboard

isKeyClicked, isKeyPressed,

// mouse

isButtonClicked, isButtonPressed, getMousePosX, getMousePosY

}

### 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/cameras folder and name it camera\_input.js. This file will contain the mouse input support functionality.
2. TEMP TEXT

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

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

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

// …

export default Camera;

1. 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 input.getMousePosX() - this.mViewport[eViewport.eOrgX];

}

Camera.prototype.\_mouseDCY = function() {

return input.getMousePosY() - this.mViewport[eViewport.eOrgY];

}

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

Camera.prototype.isMouseInViewport = function () {

let dcX = this.\_mouseDCX();

let dcY = this.\_mouseDCY();

return ((dcX >= 0) && (dcX < this.mViewport[eViewport.eWidth]) &&

(dcY >= 0) && (dcY < this.mViewport[eViewport.eHeight]));

}

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

Camera.prototype.mouseWCX = function () {

let minWCX = this.getWCCenter()[0] - this.getWCWidth() / 2;

return minWCX + (this.\_mouseDCX() \* (this.getWCWidth() / this.mViewport[eViewport.eWidth]));

}

Camera.prototype.mouseWCY = function () {

let minWCY = this.getWCCenter()[1] - this.getWCHeight() / 2;

return minWCY + (this.\_mouseDCY() \* (this.getWCHeight() / this.mViewport[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 my\_game.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.

update() {

// … identical to previous project …

msg = "";

// testing the mouse input

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

msg += "[L Down]";

if (this.mCamera.isMouseInViewport()) {

this.mPortal.getXform().setXPos(this.mCamera.mouseWCX());

this.mPortal.getXform().setYPos(this.mCamera.mouseWCY());

}

}

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

if (this.mHeroCam.isMouseInViewport()) {

this.mHero.getXform().setXPos(this.mHeroCam.mouseWCX());

this.mHero.getXform().setYPos(this.mHeroCam.mouseWCY());

}

}

if (engine.input.isButtonClicked(engine.input.eMouseButton.eRight)) {

this.mPortal.setVisibility(false);

}

if (engine.input.isButtonClicked(engine.input.eMouseButton.eMiddle)) {

this.mPortal.setVisibility(true);

}

msg += " X=" + engine.input.getMousePosX() + " Y=" + engine.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.

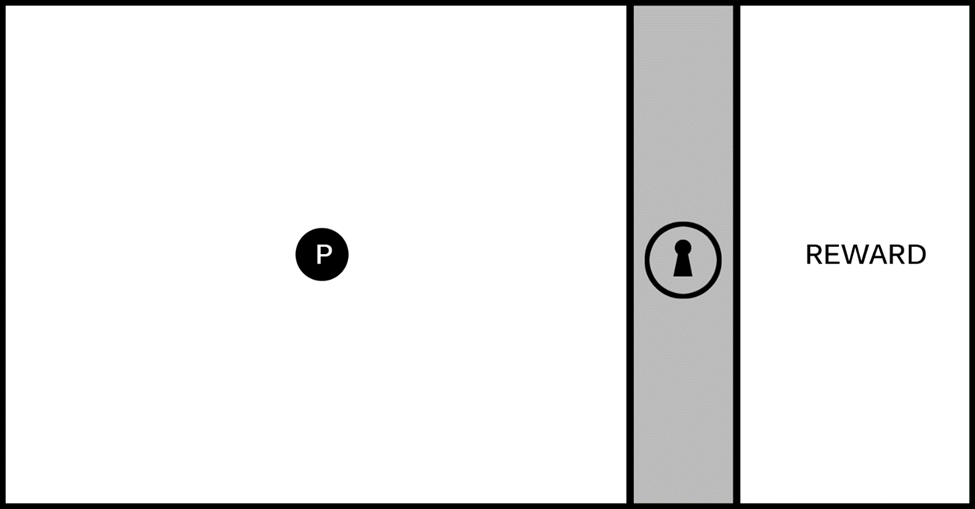


Figure 7-13. The image represents a single game screen divided into three areas. a playable area on the left with a hero character (the circle marked with a P), an impassable barrier marked with a lock icon, and a reward area on the right.

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.

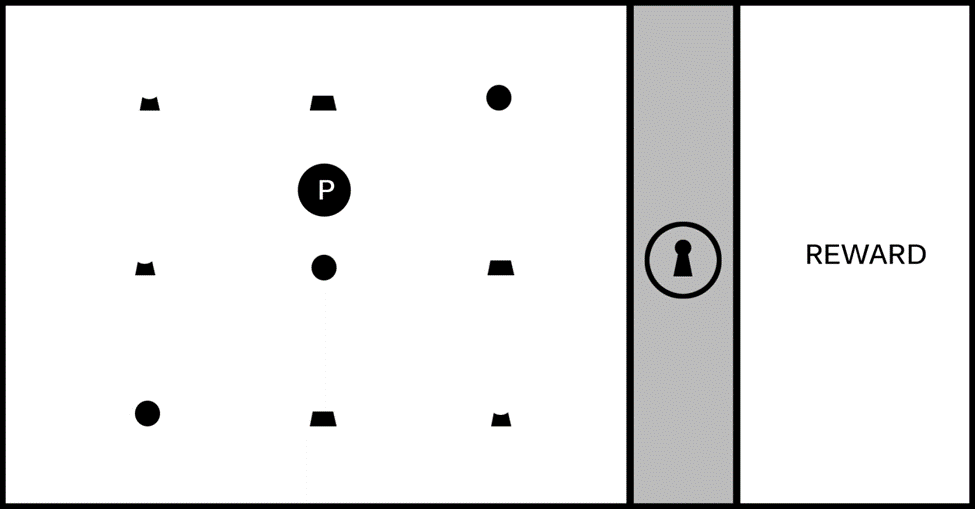


Figure 7-14. The game screen is populated with an assortment of individual objects.

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.

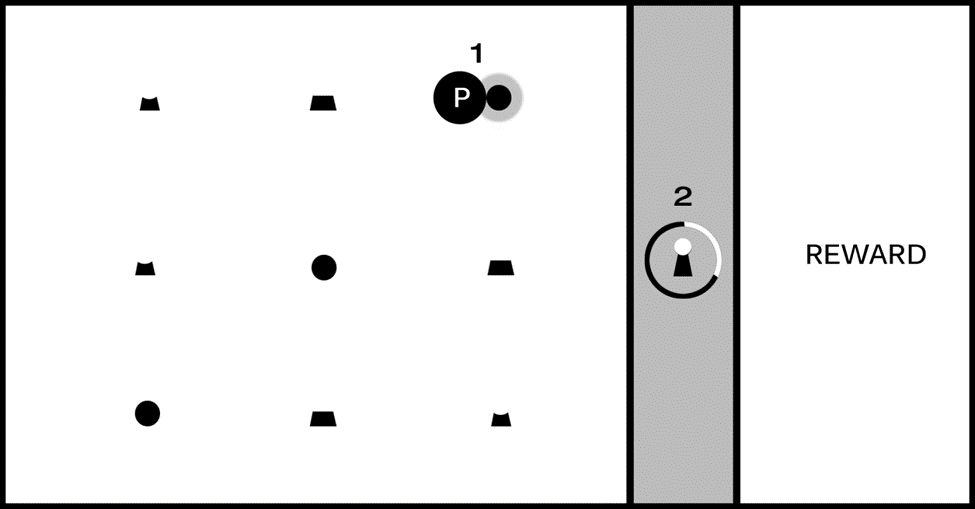


Figure 7-15. As the player moves the hero character around the game screen, the shapes “activate” with a highlight (#1); activating certain shapes causes a section of the lock and one-third of the surrounding ring to glow (#2).

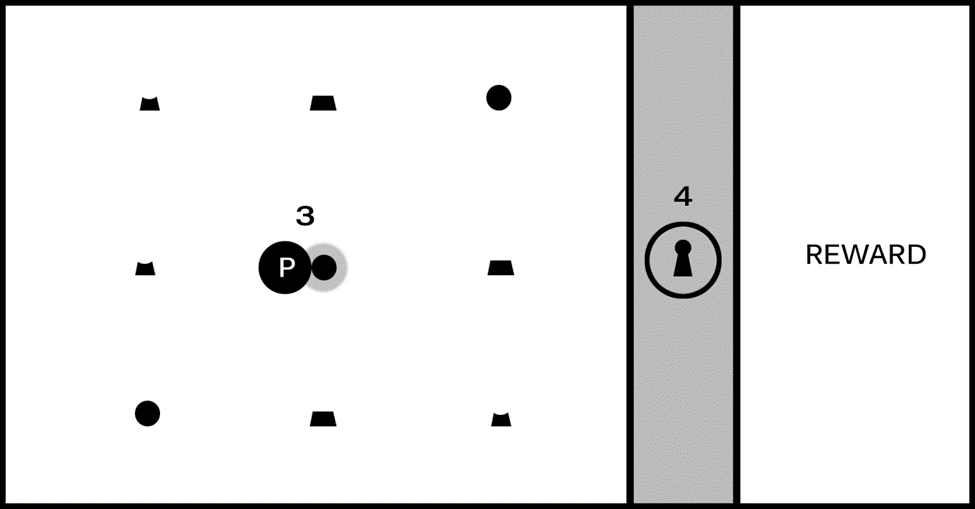


Figure 7-16. Activating some shapes (#3) will not cause the lock and ring to glow (#4).

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.

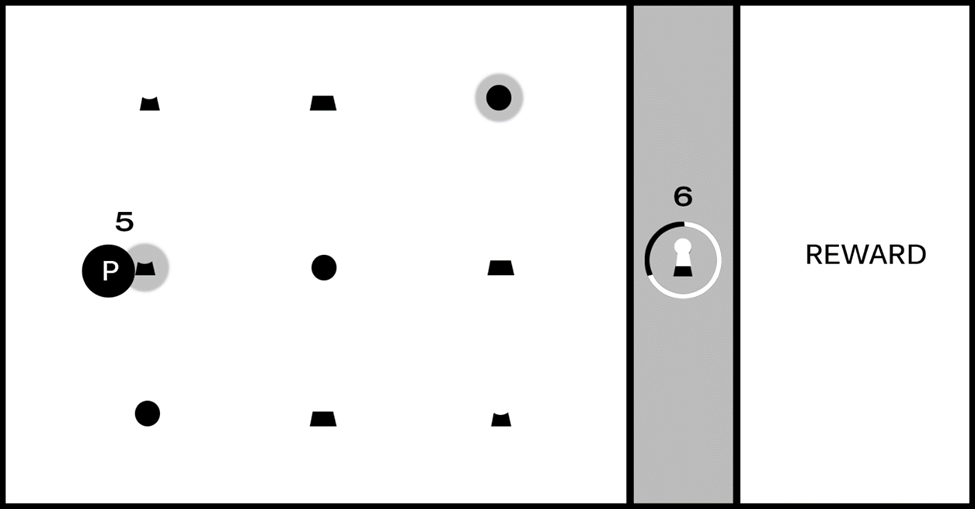


Figure 7-17. After the first object was activated (the circle in the upper-right) and caused the top section of the lock and first third of the ring to glow, as shown in Figure 7-15, the second object in the correct sequence (#5) caused the middle section of the lock and second third of the ring to glow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