Manipulating the Camera

After completing this chapter, you will be able to:

* Implement operations that are commonly employed in 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 appealing 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 . A viewport is a rectangular area on the HTML canvas with the lower-left corner located at and a dimension of . The Camera object’s setViewAndCameraMatrix() 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.

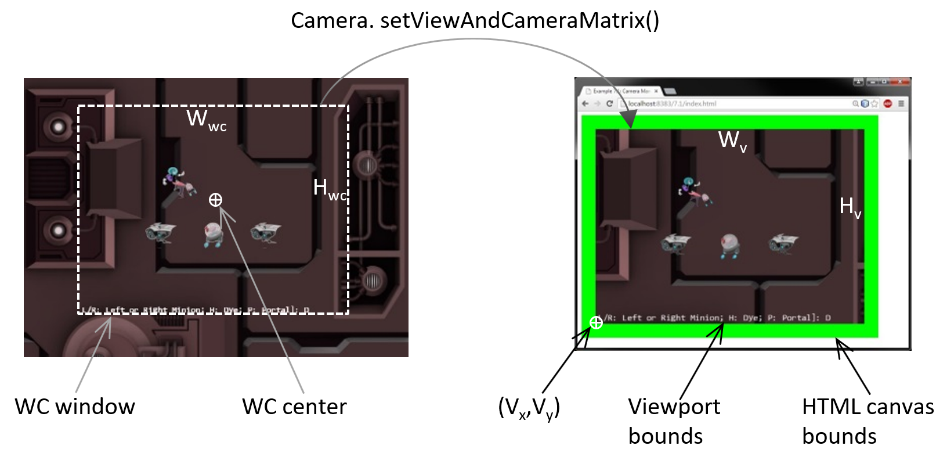


Figure 7-1. Review of WC parameters that define a Camera object

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 camera to show your audiences the environment, you may want to attach the 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 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 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.camera\_manipulations 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 center 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 a large TextureRenderable object texture mapped with bg.png.

### Organize the Source Code

To accommodate the increase in functionality and the complexity of the Camera class you will create a separate folder for storing the related source code files. Similar to the case of dividing the complicated source code of TextureRenderable into multiple files, the following steps will separate the Camera class into:

* camera\_main.js for implementing the basic functionality from previous projects
* camera\_manipulation.js for supporting the newly introduced manipulation operations
* Camera.js for serving as the class access point.

The implementation steps are as follows:

1. Create a new folder called cameras in src/engine. Move the camera.js file into this folder and rename it to camera\_main.js.
2. Create a new file in src/engine/cameras and name it camera\_manipulation.js. This file will be used to extend the Camera class functionality in supporting manipulations. Add in the following code to important and export the basic Camera class functionality. For now, this file does not contain any useful source code and thus does not serve any purpose. You will define the appropriate extension functions in the following subsection.

"use strict";

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

// new functionality to be defined here in the next subsection

export default Camera;

1. Create a new camera.js to serve as the Camera access point by adding the following code.

"use strict";

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

export default Camera;

With this structure of the source code files, camera\_main.js implements all the basic functionality and exports to camera\_manipulation.js that defines additional functionality for the Camera class. Finally, camera.js imports the extended functions from camera\_manipulation.js. The users of the Camera class can simply import from camera.js and will have access to all of the defined functionality. In this way, camera.js serves as the access point to the Camera class and hides the details of the implementation source code structure.

### Support Clamping to Camera WC Bounds

Edit camera\_main.js to import bounding box functionality and define a function to clamp the bounds associated with a Transform object to the camera WC bound.

import \* as glSys from "../core/gl.js";

import BoundingBox from "../bounding\_box.js";

import { eBoundCollideStatus } from "../bounding\_box.js";

// … identical to previous code …

clampAtBoundary(aXform, zone) {

let status = this.collideWCBound(aXform, zone);

if (status !== eBoundCollideStatus.eInside) {

let pos = aXform.getPosition();

if ((status & eBoundCollideStatus.eCollideTop) !== 0) {

pos[1] = (this.getWCCenter())[1] +

(zone \* this.getWCHeight() / 2) - (aXform.getHeight() / 2);

}

if ((status & eBoundCollideStatus.eCollideBottom) !== 0) {

pos[1] = (this.getWCCenter())[1] –

(zone \* this.getWCHeight() / 2) + (aXform.getHeight() / 2);

}

if ((status & eBoundCollideStatus.eCollideRight) !== 0) {

pos[0] = (this.getWCCenter())[0] +

(zone \* this.getWCWidth() / 2) - (aXform.getWidth() / 2);

}

if ((status & 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

Recall that you have created an empty camera\_manipulation.js source code file. You are now ready to edit this file and define additional functions on the Camera class to manipulate the camera.

**Note** JavaScript classes are implemented based on prototype chains. After class construction, instance methods can be accessed and defined via the prototype of the class or aClass.prototype.method. For more information on JavaScript classes and prototypes, please refer to https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance\_and\_the\_prototype\_chain

1. Edit camera\_manipulate.js. Ensure you are adding code between the initial import and final export of the Camera class functionality.
2. Import the bounding box collision status and define the panWidth() function to pan the camera based on the bounds of a Transform object. This 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. Like the clampAtBoundary() function, the camera will not be changed if the aXform bounds are completely outside the tested WC bounds area.

import { eBoundCollideStatus } from "../bounding\_box.js";

Camera.prototype.panWith = function (aXform, zone) {

let status = this.collideWCBound(aXform, zone);

if (status !== eBoundCollideStatus.eInside) {

let pos = aXform.getPosition();

let newC = this.getWCCenter();

if ((status & eBoundCollideStatus.eCollideTop) !== 0) {

newC[1] = pos[1]+(aXform.getHeight() / 2) - (zone \* this.getWCHeight() / 2);

}

if ((status & eBoundCollideStatus.eCollideBottom) !== 0) {

newC[1] = pos[1] - (aXform.getHeight() / 2) + (zone \* this.getWCHeight() / 2);

}

if ((status & eBoundCollideStatus.eCollideRight) !== 0) {

newC[0] = pos[0] + (aXform.getWidth() / 2) - (zone \* this.getWCWidth() / 2);

}

if ((status & eBoundCollideStatus.eCollideLeft) !== 0) {

newC[0] = pos[0] - (aXform.getWidth() / 2) + (zone \* this.getWCWidth() / 2);

}

}

}

1. Define camera panning functions panBy() and panTo() by appending to the Camera class prototype. These two functions change the camera WC center by adding a delta to it, or moving it to a new location.

Camera.prototype.panBy = function (dx, dy) {

this.mWCCenter[0] += dx;

this.mWCCenter[1] += dy;

}

Camera.prototype.panTo = function (cx, cy) {

this.setWCCenter(cx, cy);

}

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

Camera.prototype.zoomBy = function (zoom) {

if (zoom > 0) {

this.setWCWidth(this.getWCWidth() \* zoom);

}

}

Camera.prototype.zoomTowards = function (pos, zoom) {

let 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 in a process we intuitively know as zooming out. A 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.

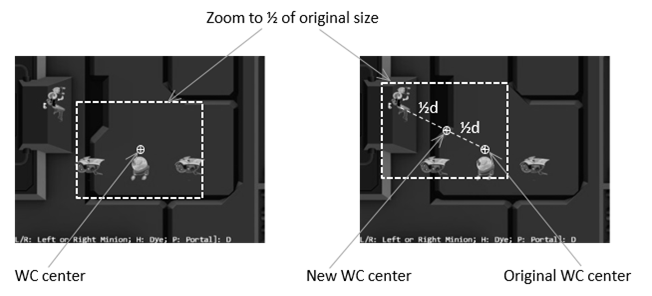


Figure 7-3. Zooming toward the WC Center and toward a target position

### Manipulating the Camera in MyGame

There are two important functionalities to be tested: panning and zooming. The only notable changes to the MyGame class are in the update() function. The init(), load(), unload(), and draw() functions are similar to previous projects and can be found in the project source code.

update() {

let zoomDelta = 0.05;

let msg = "L/R: Left or Right Minion; H: Dye; P: Portal]: ";

// … code to update each object not shown

// Brain chasing the hero

let h = [];

if (!this.mHero.pixelTouches(this.mBrain, h)) {

this.mBrain.rotateObjPointTo(this.mHero.getXform().getPosition(), 0.01);

engine.GameObject.prototype.update.call(this.mBrain);

}

// Pan camera to object

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

this.mFocusObj = this.mLMinion;

this.mChoice = 'L';

this.mCamera.panTo(this.mLMinion.getXform().getXPos(),

this.mLMinion.getXform().getYPos());

}

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

this.mFocusObj = this.mRMinion;

this.mChoice = 'R';

this.mCamera.panTo(this.mRMinion.getXform().getXPos(),

this.mRMinion.getXform().getYPos());

}

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

this.mFocusObj = this.mPortal;

this.mChoice = 'P';

}

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

this.mFocusObj = this.mHero;

this.mChoice = 'H';

}

// zoom

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

this.mCamera.zoomBy(1 - zoomDelta);

}

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

this.mCamera.zoomBy(1 + zoomDelta);

}

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

this.mCamera.zoomTowards(this.mFocusObj.getXform().getPosition(),

1 - zoomDelta);

}

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

this.mCamera.zoomTowards(this.mFocusObj.getXform().getPosition(),

1 + zoomDelta);

}

// interaction with the WC bound

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 listed code, the first four if statements select the in-focus object, where L and R keys also re-center the camera by calling the panTo() function with the appropriate WC positions. The second set of four if statements control the zoom, either 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. Zooming into a desired region of a world is a useful feature for game developers with many applications. Try modifying this demo to replicate ways that you have seen zoom utilized in other games.

# 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 is not only visually unpleasant, but can confuse the player as to what has happened.

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 a parameter with the old value is to be assigned a new one. In this case, instead of updating the value abruptly, 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 .

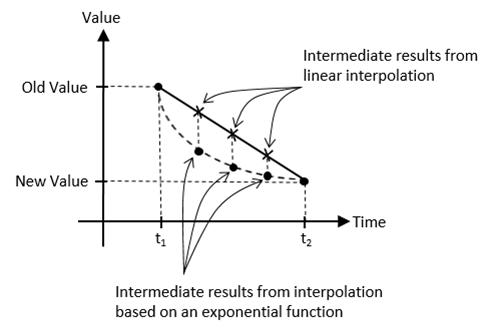


Figure 7-4. Interpolating values based on linear and exponential functions

Figure 7-4 shows that there are 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. In this way, with linear interpolation, a camera position would move from an old to new position with a constant speed similar to a moving (or panning) a camera at some constant-speed. In comparison, the interpolation based on the given exponential function would move the camera position rapidly at first, then slow down over time giving a sensation of moving and focusing the camera on a new target.

Human motions and movements typically follow the exponential interpolation function. For example, try turning your head from facing the front to facing the right, or moving your hand to pick up an object on your desk. Notice that in both cases, you began with a relatively quick motion and slowed down significantly when the destination is in close proximity. That is, you probably started by turning your head quickly and slowed down rapidly as your view approaches your right side, and it is likely your hand started moving quickly towards the object and slowed down significantly when the hand is almost reaching the object. In both of these examples, your displacements followed the exponential interpolation function as depicted in Figure 7-4—quick changes followed by rapid slow down as destination approaches. This is the function you will implement in the game engine because it mimics human movements and is likely to seem natural to human players.

**Note** Linear interpolation is often referred to as LERP or lerp. Often times the term is also being used to refer to non-linear interpolations. For example, many applications and libraries, including the gl-matrix library that you are using, refer to the exponential interpolation depicted in Figure 7-4 as *lerp*. This is the convention that will be followed in this game engine.

This section introduces the Lerp and LerpVec2 utility classes to support smooth and gradual camera movements resulting from camera manipulation operations.

## The Camera Interpolations Project

This project demonstrates the smoother visually appealing 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.camera\_interpolations 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 the concept of interpolation between given values
* To implement interpolation supporting gradual camera parameter changes
* To experience interpolated changes in camera parameters

As in previous projects, you can find external resource files in the assets folder.

### Interpolation as a Utility

Similar to the Transform class supporting transformation functionality and the BoundingBox class supporting collision detection, a Lerp class can be defined to support interpolation of values. To keep the source code organized, a new folder should be defined to store these utilities.

Create the new folder called src/engine/utils and move the transform.js and bounding\_box.js files into this folder.

#### The Lerp Class

Define the Lerp class to compute interpolation between two values.

1. Create a new file in the src/engine/utils folder, name it lerp.js, and define the constructor. This class is designed to interpolate values from mCurrentValue to mFinalValue in the duration of mCycles. During each update, intermediate results are computed based on the mRate increment on the difference between mCurrentValue and mFinalValue, as shown next.

class Lerp {

constructor(value, cycles, rate) {

this.mCurrentValue = value; // begin value of interpolation

this.mFinalValue = value; // final value of interpolation

this.mCycles = cycles;

this.mRate = rate;

// Number of cycles left for interpolation

this.mCyclesLeft = 0;

}

// … implementation to follow …

}

1. Define the function that computes the intermediate results.

// subclass should override this function for non-scalar values

\_interpolateValue() {

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. mCurrentValue will be set to the intermediate value each iteration following an exponential curve until it equals mFinalValue.

1. Define a function to configure the interpolation. The mRate variable defines how quickly the interpolated result approaches the final value. A mRate of 0.0 will result in no change at all, where 1.0 causes instantaneous change. The mCycle variable defines the length of the interpolation process.

config(stiffness, duration) {

this.mRate = stiffness;

this.mCycles = duration;

}

1. Define relevant getter and setter functions. Note that the setFinal() function both sets the final value and triggers a new round of interpolation computation.

get() { return this.mCurrentValue; }

setFinal(v) {

this.mFinalValue = v;

this.mCyclesLeft = this.mCycles; // will trigger interpolation

}

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

update() {

if (this.mCyclesLeft <= 0) { return; }

this.mCyclesLeft--;

if (this.mCyclesLeft === 0) {

this.mCurrentValue = this.mFinalValue;

} else {

this.\_interpolateValue();

}

}

1. Last, make sure to export the defined class.

export default Lerp;

#### The LerpVec2 Class

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

1. Create a new file in the src/engine/utils folder and name it lerp\_vec2.js, and define its constructor.

class LerpVec2 extends Lerp {

constructor(value, cycle, rate) {

super(value, cycle, rate);

}

// … implementation to follow …

}

1. Override the \_interpolateValue() function to compute intermediate results for vec2.

\_interpolateValue() {

vec2.lerp(this.mCurrentValue, this.mCurrentValue, this.mFinalValue, this.mRate);

}

The vec2.lerp() function defined in the gl-matrix.js file computes the vec2 components for x and y. This is identical to the interpolateValue() function in the Lerp class.

### Represent Interpolated Intermediate Results with CameraState

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

1. Create a new file in the src/engine/cameras folder, name it camera\_state.js, import the defined Lerp functionality, and define the constructor.

import Lerp from "../utils/lerp.js";

import LerpVec2 from "../utils/lerp\_vec2.js";

class CameraState {

constructor(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 LerpVec2(center, this.kCycles, this.kRate);

this.mWidth = new Lerp(width, this.kCycles, this.kRate);

}

// … implementation to follow …

}

export default CameraState;

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 corresponding Lerp classes and are capable of interpolating and computing intermediate results to achieve gradual changes.

1. Define the getter and setter functions.

getCenter() { return this.mCenter.get(); }

getWidth() { return this.mWidth.get(); }

setCenter(c) { this.mCenter.setFinal(c); }

setWidth(w) { this.mWidth.setFinal(w); }

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

update() {

this.mCenter.update();

this.mWidth.update();

}

1. Define a function to configure the interpolation.

config(stiffness, duration) {

this.mCenter.config(stiffness, duration);

this.mWidth.config(stiffness, duration);

}

The stiffness variable is the mRate of Lerp. It defines how quickly the interpolated intermediate results should converge to the final value. As discussed in the Lerp class definition, this is a number between 0 to 1, where 0 means the convergence will never happen and a 1 means instantaneous convergence. The duration variable is the mCycle of Lerp. It defines the number of update cycles until convergence should take place. This must be a positive integer value.

### Integrate Interpolation into Camera Manipulation Operations

The Camera class in camera\_main.js must be modified to represent the WC center and width using the newly defined CameraState.

1. Edit the camera\_main.js file and import the newly defined CameraState class.

import CameraState from "./camera\_state.js";

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

constructor(wcCenter, wcWidth, viewportArray) {

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

// … identical to previous code …  
}

1. 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.update();

}

// For LERP function configuration

Camera.prototype.configLerp = function (stiffness, duration) {

this.mCameraState.config(stiffness, duration);

}

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

Camera.prototype.panBy = function (dx, dy) {

let newC = vec2.clone(this.getWCCenter());

newC[0] += dx;

newC[1] += dy;

this.mCameraState.setCenter(newC);

}

1. Update panWith() and zoomTowards() functions to receive and set WC center to the newly defined CameraState object.

Camera.prototype.panWith = function (aXform, zone) {

let status = this.collideWCBound(aXform, zone);

if (status !== eBoundCollideStatus.eInside) {

let pos = aXform.getPosition();

let newC = vec2.clone(this.getWCCenter());

if ((status & eBoundCollideStatus.eCollideTop) !== 0)

// … identical to previous code …

this.mCameraState.setCenter(newC);

}

}

Camera.prototype.zoomTowards = function (pos, zoom) {

// … identical to previous code …

this.zoomBy(zoom);

this.mCameraState.setCenter(newC);

}

### Export the Lerp Functionality to the Client

The last step in integrating any new functionality into the engine involves modifying the engine access file, index.js. Edit index.js and add the following import and export statements to grant the client access to Lerp.

// … identical to previous code …

import Transform from "./utils/transform.js";

import BoundingBox from "./utils/bounding\_box.js";

import { eBoundCollideStatus } from "./utils/bounding\_box.js";

import Lerp from "./utils/lerp.js";

import LerpVec2 from "./utils/lerp\_vec2.js";

// … identical to previous code …

export default {

// … identical to previous code …

input,

// general utils

Lerp, LerpVec2,

// Util classes

Camera, Scene, Transform, BoundingBox,

// … identical to previous code …

}

### Testing Interpolation 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.

update() {

let zoomDelta = 0.05;

let msg = "L/R: Left or Right Minion; H: Dye; P: Portal]: ";

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

//… identical to previous code …

}

The call to update the camera for computing interpolated intermediate results is the only change in the my\_game.js file. You can now run the project and experiment with the smooth and gradual changes resulting from camera manipulation operations. Notice that the interpolated results mean the rendered image never abruptly changes and there is a sense of continuity in space before and after the user’s camera manipulation. You can try changing the stiffness and duration variables to better appreciate the different rates of interpolation convergence.

# Camera Shake and Object Oscillation Effects

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 re-center 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.

Note that straight mathematic formulation is precise with perfect predictability. Such formulation can be suitable for describing regular, normal, or expected behaviors. For example, the bouncing of a ball, or the oscillation of a pendulum. A shaking effect should involve slight chaotic and unpredictable randomness. For example, the stabilization of the coffee carrying hand after an unexpected collision, or, as in the previous example, the stabilization of the video camera after being startled. Following this reasoning, in this section you will define a general damped oscillation function, and then inject pseudo-randomness to simulate the slight chaos to achieve the shake effect.

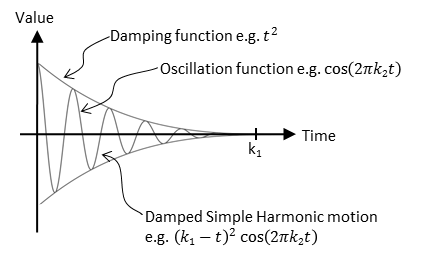


Figure 7-6. The displacements of a damped simple harmonic motion

## The Camera Shake and Object Oscillate Project

This project demonstrates how to implement damped simple harmonic motion to simulate object oscillation, and the injection of pseudo-randomness to create camera shake effect. You can see an example of this project running in Figure 7-7. This project is identical to the previous project except for an additional command to create object oscillation and camera shake effects. The source code to this project is defined in the chapter7/7.3.camera\_shake\_and\_object\_oscillate folder.



Figure 7-7. Running the Camera Shake and object oscillate project

The following is the new control of this project:

* Q key: Initiates the positional oscillation of the Dye character and the camera shake effects.

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 insight into modeling displacements with simple mathematical functions
* To oscillate the position of an object
* To experience the camera shake effect
* To implement oscillations as damped simple harmonic motion, and to introduce pseudo-randomness to create the camera shake effect

As in previous projects, you can find external resource files in the assets folder.

### Abstract the Shake Behavior

The ability to shake the camera is a common yet dynamic behavior in many games. However, it is important to recognize that the shake behavior can be applied to more than just the camera. That is, the shaking effect can be abstracted as the perturbation (shaking) of numerical value(s) such as a size, a point, or a position. In the case of camera shake, it just so happened that the numerical values being shaken represent the x- and y-positions of the camera. For this reason, the shake and associated supports should be written as general utility functions of the game engine so they can be applied by the game developer to any numerical values. The following are the new utilities that will be defined.

* Oscillate: the base class that implements simple harmonic oscillation of a value over time
* Shake: an extension of the Oscillation class that introduces randomness to the magnitudes of the oscillations to simulate slight chaos of the shake effect on a value
* ShakeVec2: an extension of the Shake class that expands the Shake behavior to two *values* such as a position

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

Because all of the described behaviors depend on simple oscillation, this should be implemented first.

1. Create a new file in the src/engine/utils folder and name it oscillate.js. Define a 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);

}

Refer to Figure 7-8, mNumCyclesLeft is number of cycles left in the oscillation, or k-t, and the frac variable, , is damping factor based on the ratio until the end of the oscillation. This function returns a value between -1 and 1 and should be scaled accordingly.

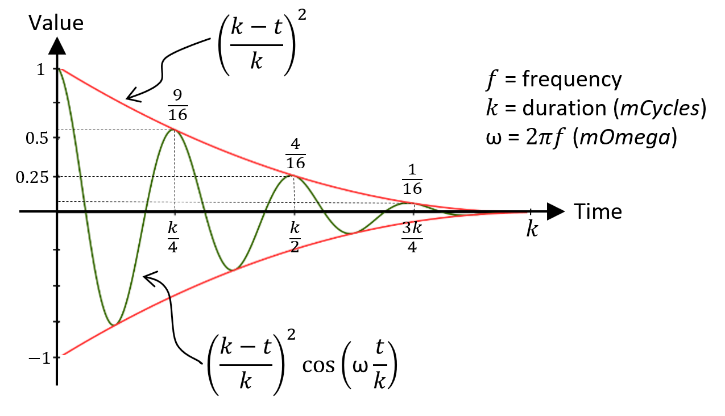


Figure 7-8. The damped simple harmonic motion that specifies value ocillation

1. Define a protected function to retrieve the value of the next damped harmonic motion. This function may seem trivial and unnecessary. However, as you will observe in the next subsection, this function allows a shake subclass to over-write and inject randomness.

// local/protected methods

\_nextValue() {

return (this.\_nextDampedHarmonic());

}

1. Define functions to check for the end of the oscillation, and for restarting the oscillation.

done() { return (this.mNumCyclesLeft <= 0); }  
reStart() { this.mNumCyclesLeft = this.mCycles; }

1. Lastly, define a public function to trigger the calculation of oscillation. Notice that the computed oscillation result must be scaled by the desired magnitude, mMag.

getNext() {

this.mNumCyclesLeft--;

let v = 0;

if (!this.done()) {

v = this.\_nextValue();

}

return (v \* this.mMag);

}

#### Create the Shake Class to Randomize an Oscillation

You can now extend the oscillation behavior to convey a sense of shaking by introducing pseudo-randomness into the effect.

1. Create a new file, shake.js, in the src/engine/utils folder. Define the Shake class to extend Oscillate 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. Overwrite the \_nextValue() to randomize the sign of the oscillation results as follows. Recall that the \_nextValue() function is called from the public getNext() function to retrieve the oscillating value. While the results from the damped simple harmonic oscillation continuously and predictably decrease in magnitude, the associated signs of the values are randomized causing sudden and unexpected discontinuities conveying a sense of chaos from the results of a shake.

\_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 vec2, or a Position

You can now generalize the shake effect to support the shaking of two values simultaneously. This is a useful utility because positions in 2D games are two-value entities and positions are convenient targets for shake effects. Just as in the case of shaking of the camera position in camera shake simulation.

The ShakeVec2 class extends the Shake class to support the shaking of a vec2 object, shaking the values in both x and y-dimensions. The x-dimension shaking is support via an instance of the Shake object, while the y-dimension is supported via the Shake class functionality that is defined in the super class.

1. Create a new file, shake\_vec2.js, in the src/engine/utils folder. Define the class ShakeVec2 class to extend the Shake class. Similar to the constructor parameters of the Shake super classes, the deltas and freqs parameters are 2-dimensional, or vec2, versions of magnitude and frequency for shaking in the x- and y-dimensions. In the constructor, the xShake instance variable keeps track of shaking effect in the x-dimension. Note the y-component parameters, array indices of 1, in the super() constructor invocation. The Shake super class keeps track of the shaking effect in the y-dimension.

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. Extend the restart() and getNext() functions to support the second dimension.

reStart() {

super.reStart();

this.xShake.reStart();

}

getNext() {

let x = this.xShake.getNext();

let y = super.getNext();

return [x, y];

}

### Define the CameraShake Class to Abstract the Camera Shaking Effect

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

1. Create a new file, camera\_shake.js, in the src/engine/cameras folder and define the constructor to receive the camera state, the state parameter, and shake configurations: deltas, freqs, and shakeDuration. The parameter state is of datatype CameraState, consisting of its center position and width.

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

class CameraShake {

// state is the CameraState to be shaken.

constructor(state, deltas, freqs, shakeDuration) {

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

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

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

}

// … implementation to follow …

}

export default CameraShake;

1. Define the function that triggers the displacement computation for accomplishing the shaking effect. Notice that the shake results are offsets from the original position. The given code adds this offset to the original camera center position.

update() {

let delta = this.mShake.getNext();

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

}

1. Define utility functions: inquire if shaking is done, re-start the shaking, and getter/setters.

done() { return this.mShake.done(); }

reShake() {this.mShake.reStart();}

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 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. In camera\_main.js modify the Camera constructor to initialize a CameraShake object.

constructor(wcCenter, wcWidth, viewportArray) {

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

this.mCameraShake = null;

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

// … identical to previous code …

}

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

setViewAndCameraMatrix() {

// … identical to previous code …

// Step B: Compute the Camera Matrix

let center = [];

if (this.mCameraShake !== null) {

center = this.mCameraShake.getCenter();

} else {

center = this.getWCCenter();

}

// … identical to previous code …

}

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

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

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

}

// Restart 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();

}

### Export the Oscillate and Shake Functionality to the Client

As the last step in the integration, modify the engine access file, index.js, to export the newly defined utilities.

// … identical to previous code

// general utilities

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

// … identical to previous code

import Oscillate from "./utils/oscillate.js";

import Shake from "./utils/shake.js";

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

// … identical to previous code

export default {  
 // … identical to previous code

// general utils

Lerp, LerpVec2, Oscillate, Shake, ShakeVec2,

// … identical to previous code

}

### Testing the Camera Shake and Oscillation Effects in MyGame

The my\_game.js file only needs to be modified slightly in the init() and update() functions to support triggering the oscillation and camera shake effects with the Q key.

1. Define a new instance variable for creating oscillation or bouncing effect on the Dye character.

init() {

// … identical to previous code …

// create a Oscillate object to simulate motion

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

}

1. Modify the update() function to trigger the bouncing and camera shake effects with the Q key. In the following code note the advantage of well-designed abstraction. For example, the camera shake effect is opaque where the only information a programmer needs to specify is the actual shake behavior, i.e., the shake magnitude, frequency, and duration. In contrast, the oscillating or bouncing effect of the Dye character position is accomplished by explicitly inquiring and using the mBounce results.

update() {

// … identical to previous code …

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 the pseudo-random damped simple harmonic motion that simulates the camera shake effect. You can also observe the oscillation of the Dye character’s x-position. 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 when creating the mBounce object, or when calling the mCamera.shake() function to experiment with different oscillation and shake configurations. Recall that in both cases the first two parameters control the initial displacements and the frequency (number of cosine periods), and the third parameter is the duration of how long the effects 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 providing 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 abstraction can be extended to support multiple Camera objects. Each view in the game can 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 positional oscillation of the Dye character and the camera shake effects.
* 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 specific purposes

As in previous projects, you can find external resource files 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. Edit camera\_main.js and 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 the 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 code …

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

gl.scissor(this.mScissorBound[0], // x position of bottom-left corner

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

this.mScissorBound[2], // width of the area to be drawn

this.mScissorBound[3]);// height of the area to be drawn

// … identical to previous code …

}

### 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 init(), draw(), and update() functions are modified to handle the multiple Camera objects and are highlighted.

1. Modify the init() function to define three Camera objects. 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.

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 code …

}

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

draw() {

// Step A: clear the canvas

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

// Step B: Draw with all three cameras

this.\_drawCamera(this.mCamera);

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

this.\_drawCamera(this.mHeroCam);

this.\_drawCamera(this.mBrainCam);

}

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 code …

// 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 displayed 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.draw() function the mBrainCam is drawn last. The last drawn object 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.configLerp() 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 a 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. Remember that 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.

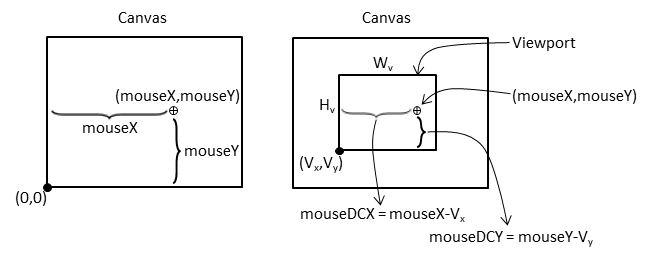


Figure 7-10. Mouse position on canvas and viewport

The left drawing in 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 right drawing in 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:



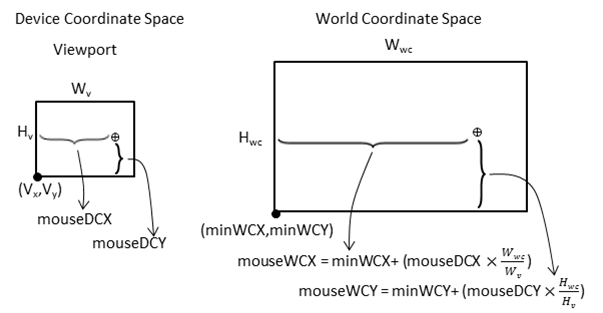


Figure 7-11. Mouse position in viewport DC space and WC space

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.

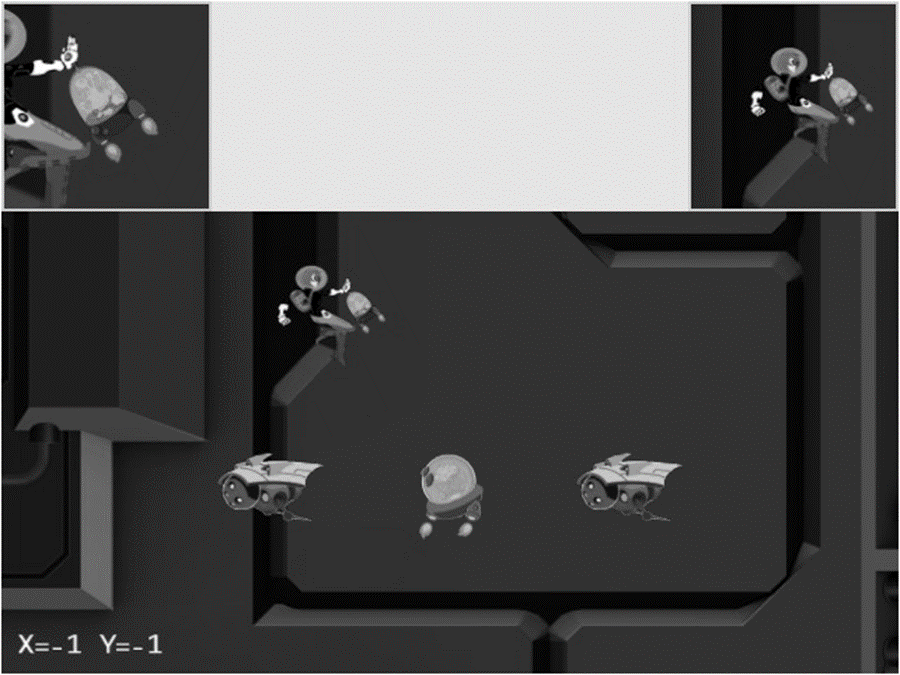


Figure 7-12. Running the Mouse Input project

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/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 positional oscillation of the Dye character and the camera shake effects.
* 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 that mouse clicks are specific to individual viewports
* To implement transformation between coordinate spaces
* To support and experience working with mouse input

As in previous projects, you can find external resource files in the assets folder.

### Modify index.js to Pass Canvas ID to Input Component

To receive mouse input information, the input component needs to have access to the HTML canvas. Edit index.js and modify the init() function to pass the htmlCamvasID to the input component during initialization.

// … identical to previous code …

// general engine utilities

function init(htmlCanvasID) {

glSys.init(htmlCanvasID);

vertexBuffer.init();

input.init(htmlCanvasID);

audio.init();

shaderResources.init();

defaultResources.init();

}

// … identical to previous code …

### Implement Mouse Support in input.js

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

1. Edit input.js and define the 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 code …

// 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 code …

// update mouse input state

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

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

mButtonPreviousState[i] = mIsButtonPressed[i];

}

}

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

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

function isButtonClicked(button) { return mIsButtonClicked[button]; }

function getMousePosX() { return mMousePosX; }

function getMousePosY() { return mMousePosY; }

1. Lastly, remember to export the newly defined functionality.

export {

keys, **eMouseButton**,

init, cleanUp, update,

// keyboard

isKeyClicked, isKeyPressed,

**// mouse**

**isButtonClicked, isButtonPressed, getMousePosX, getMousePosY**

}

### Modify the Camera to Support Viewport to WC Space Transform

The Camera class encapsulates the WC window and viewport and thus should be responsible for transforming mouse positions. Recall that to maintain readability, the Camera class source code files are separated according to functionality. The basic functions of the class are defined in camera\_main.js. The camera\_manipulate.js file imports from camera\_main.js and defines additional manipulation functions. Lastly, the camera.js file imports from camera\_manipulate.js to include all the defined functions and exports the Camera class for external access.

This chaining of imports from subsequent source code files to define additional functions will continue for the Camera class, with camera\_input.js defining input functionality.

1. Create a new file in the src/engine/cameras folder and name it camera\_input.js. This file will expand the Camera class by defining the mouse input support functions. Import the following files:
2. camera\_manipulation.js for all the defined functions for the Camera class
3. eViewport constants for accessing the viewport array
4. input module to access the mouse-related functions.

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

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

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

// … implementation to follow …

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]));

}

Lastly, update the Camera class access file to properly export the newly defined input functionality. This is accomplished by editing the camera.js file and replacing the import from camera\_manipulate.js with camera\_input.js.

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

export default Camera;

### 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 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 code …

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. These two mouse click controls will cause execution 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 following the mouse position changes. 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

You’ve learned the basics of object interaction and it’s a good time to start thinking about creating your first simple game mechanic and experimenting with 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 we might expect in an industry like video games where the creators typically spend quite a bit of time as content consumers before transitioning into content makers. Game studios often 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 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 entirely new systems and mechanics from the ground-up.

The above might lead us to ask, “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, board, physical, video, and other game types. Taking it a step further, a good game is an interactive experience with rules people enjoy learning and applying to achieve an outcome they feel invested in. There’s quite a bit to unpack in this brief definition, of course, but as a general rule, players will enjoy a game more when the rules are discoverable, 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 being 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 we’ll explore 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. This exercise will help you develop insight into creating 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 puzzle.

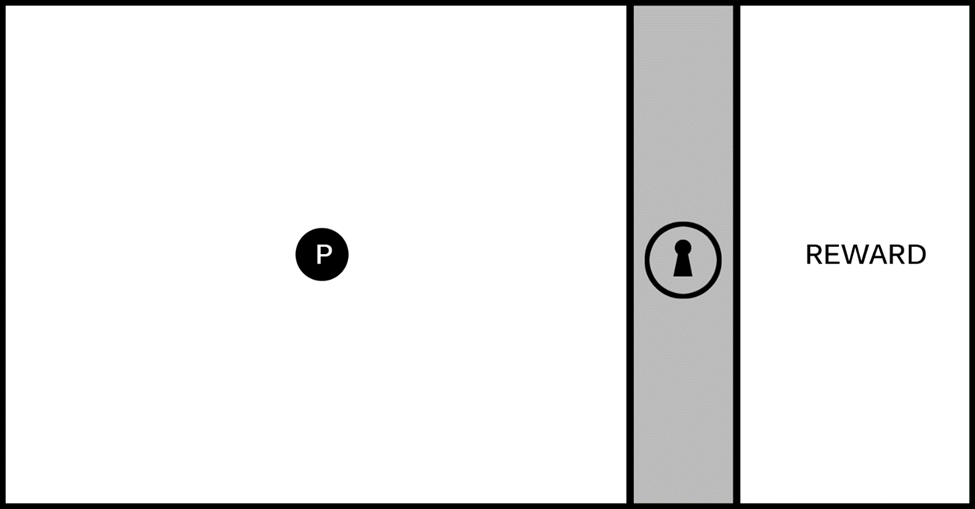


Figure 7-13. The image represents a single game screen divided into three areas. a playable area on the left with a hero character (the circle marked with a P), an impassable barrier marked with a lock icon, and a reward area on the right.

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 might involve jumping or shooting, puzzle solving, narrative situations, or 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.

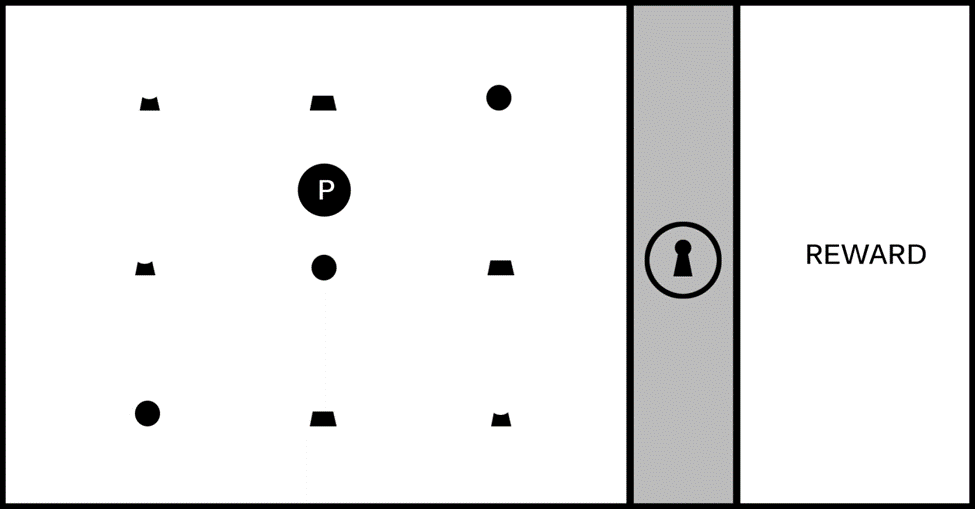


Figure 7-14. The game screen is populated with an assortment of individual objects.

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.

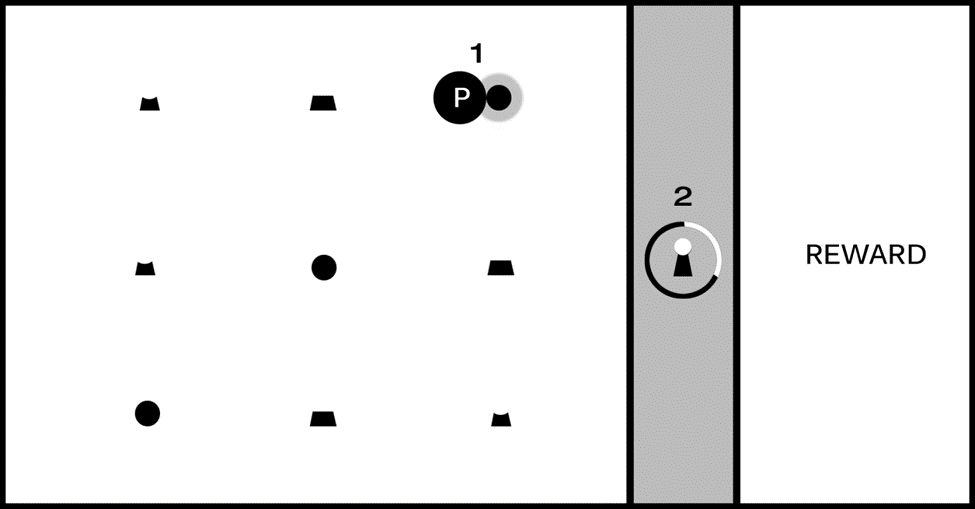


Figure 7-15. As the player moves the hero character around the game screen, the shapes “activate” with a highlight (#1); activating certain shapes causes a section of the lock and one-third of the surrounding ring to glow (#2).

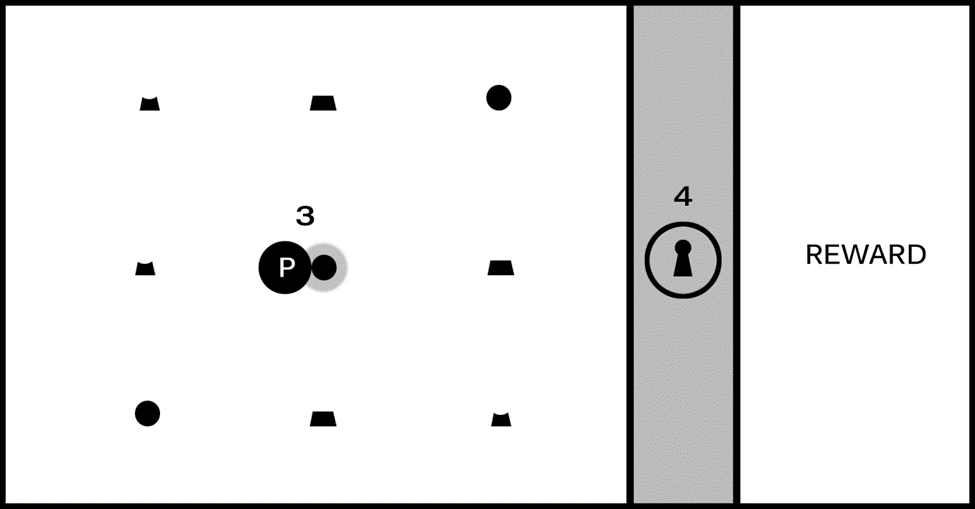


Figure 7-16. Activating some shapes (#3) will not cause the lock and ring to glow (#4).

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.

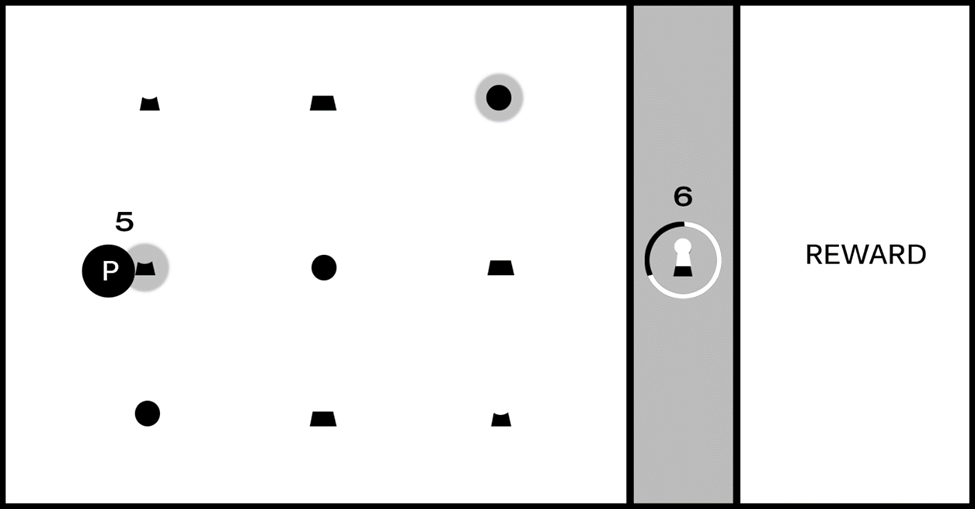


Figure 7-17. After the first object was activated (the circle in the upper-right) and caused the top section of the lock and first third of the ring to glow, as shown in Figure 7-15, the second object in the correct sequence (#5) caused the middle section of the lock and second third of the ring to glow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

You (and players) should now have all required clues to learn the rules of this mechanic and solve the puzzle. There are three shapes the player can interact with and 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).

Congratulations, 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 a game mechanic without narrative context: the game screen is intentionally devoid of game setting, visual style, or genre alignment at this stage of design because we don’t want to burden our exploration yet with any preconceived expectations. It can benefit you as a designer to spend time exploring game mechanics in their purest form before adding higher-level game elements like narrative and genre,, and you’ll likely be surprised at the unexpected directions these simple mechanics will take you as you build them out.

Simple mechanics like the one in this example can be described as “complete a multistage task in the correct sequence to achieve a goal” and 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 your 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 a method for advancing game action. Imagine in the previous example that after a player receives a reward for unlocking the barrier they move the hero object to the right side of the screen and advance 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 our “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 “adjectives” that add interest to the basic nouns and verbs we’ve been creating from the letters in the game design alphabet.

A game designer’s primary challenge is typically 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 generally considered poor design to frustrate players by creating scenarios that are logically inconsistent and make players feel that they succeeded in a challenge only by random 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.