Manipulating the Camera

After completing this chapter, you will be able to:

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

# Introduction

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

Figure 7-1 presents a brief review of the Camera object abstraction that was introduced in Chapter 3. The Camera object allows the game programmer to define a World Coordinate (WC) window of the game world to be displayed into a viewport on the HTML canvas. The WC window is the bounds defined by a WC center and a dimension of . 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 setUpViewProjection() function encapsulates the details and enables the drawing of all game objects inside the WC window bounds to be displayed in the corresponding viewport.

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

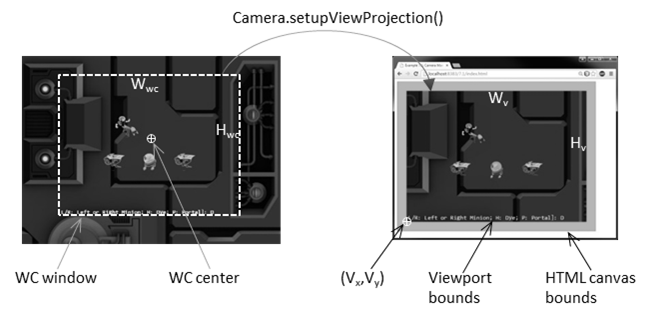


Figure 7-1. Review of WC parameters that define a Camera object

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

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

# Camera Manipulations

In a 2D world, you may want to clamp or restrict the movements of objects to be within the bounds of a camera, to pan or move the camera, or to zoom the camera into or away from specific areas. These high-level functional specifications can be realized by strategically changing the parameters of the Camera object: the WC center and the 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 focus on the Left or Right minion.
* N/M keys: Zoom into or away from the center of the camera.
* J/K keys: Zoom into or away while ensuring the constant relative position of the currently in-focus object. In other words, as the camera zooms, the positions of all objects will change except that of the in-focus object.

The goals of the project are as follows:

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

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

### Organize the Source Code

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

1. Create a new folder called cameras in src/engine. Move the camera.js file into this folder.
2. In order to divide the complexity of the Camera class across multiple files and avoid one giant file for the Camera, create a new file in src/engine/cameras called camera\_main.js and cut and paste the contents of camera.js into this file. The camera\_main.js file will serve as the Camera’s main class file which holds basic initialization while camera.js will serve as the class’s access point.

**Note** Since the Camera class is now defined in the camera\_main.js file it must contain the class’s constructor.

1. Now create a new file in src/engine/cameras and name it camera\_manipulation.js. This file will be used to extend the Camera class across multiple files and will contain the Camera functions needed to control/manipulate it.
2. Finally, edit camera.js to serve as the Camera access point by adding the following code. Remember all its previous functionality has been moved to camera\_manipulation.js.

"use strict";

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

export default Camera;

The details on how to extend the Camera across multiple files will be addressed upon implementation of camera\_manipulation.js later in this section. First you should add new clamping functionality to the Camera by modifying camera\_main.js.

### Support Clamping to Camera WC Bounds

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

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.

Also remember to update the import functions to account for the file location change and the use of the bounding\_box.js file as follows:

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

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

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

### Define Camera Manipulation Operations in camera\_manipulation.js File

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

1. Edit the camera\_manipulation.js file to define functions to pan, or move, the camera by an offset and to a new WC center.

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

this.mWCCenter[0] += dx;

this.mWCCenter[1] += dy;

}

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

this.setWCCenter(cx, cy);

}

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

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 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 and, thus, zoom out. The zoom value of less than 1 zooms in. Figure 7-3 shows the results of zoom=0.5 for zooming with respect to the center of WC and with respect to the position of the Hero object.

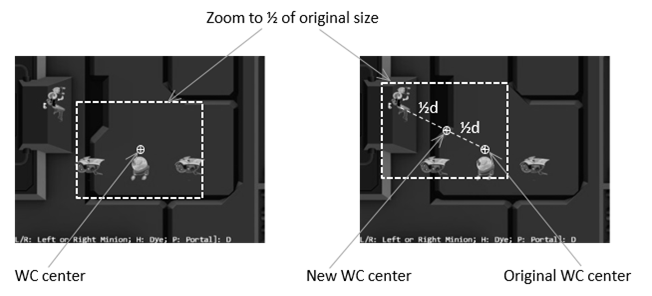


Figure 7-3. Zooming toward the WC Center and toward a target position

1. TEMP TEXT

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

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

export default Camera;

### Manipulating the Camera in MyGame

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

class MyGame extends engine.Scene {

constructor() {

super();

this.kMinionSprite = "assets/minion\_sprite.png";

this.kMinionPortal = "assets/minion\_portal.png";

this.kBg = "assets/bg.png";

// The camera to view the scene

this.mCamera = null;

this.mBg = null;

this.mMsg = null;

// the hero and the support objects

this.mHero = null;

this.mBrain = null;

this.mPortal = null;

this.mLMinion = null;

this.mRMinion = null;

this.mFocusObj = null;

this.mChoice = 'D';

}

load() {

engine.texture.load(this.kMinionSprite);

engine.texture.load(this.kMinionPortal);

engine.texture.load(this.kBg);

}

unload() {

engine.texture.unload(this.kMinionSprite);

engine.texture.unload(this.kMinionPortal);

engine.texture.unload(this.kBg);

}

init() {

// Step A: set up the cameras

this.mCamera = new engine.Camera(

vec2.fromValues(50, 37.5), // position of the camera

100, // width of camera

[0, 0, 640, 480] // viewport (orgX, orgY, width, height)

);

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

// sets the background to gray

// Large background image

let bgR = new engine.SpriteRenderable(this.kBg);

bgR.setElementPixelPositions(0, 1024, 0, 1024);

bgR.getXform().setSize(150, 150);

bgR.getXform().setPosition(50, 35);

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

// Objects in the scene

this.mBrain = new Brain(this.kMinionSprite);

this.mHero = new Hero(this.kMinionSprite);

this.mPortal = new TextureObject(this.kMinionPortal, 50, 30, 10, 10);

this.mLMinion = new Minion(this.kMinionSprite, 30, 30);

this.mRMinion = new Minion(this.kMinionSprite, 70, 30);

this.mFocusObj = this.mHero;

this.mMsg = new engine.FontRenderable("Status Message");

this.mMsg.setColor([1, 1, 1, 1]);

this.mMsg.getXform().setPosition(2, 4);

this.mMsg.setTextHeight(3);

}

// This is the draw function, make sure to setup proper drawing environment, and more

// importantly, make sure to \_NOT\_ change any state.

draw() {

// Step A: clear the canvas

engine.clearCanvas([0, 1, 0, 1.0]); // clear to bright green

// Step B: Draw with all three cameras

this.mCamera.setViewAndCameraMatrix();

// Step C: Draw everything

this.mBg.draw(this.mCamera);

this.mHero.draw(this.mCamera);

this.mBrain.draw(this.mCamera);

this.mPortal.draw(this.mCamera);

this.mLMinion.draw(this.mCamera);

this.mRMinion.draw(this.mCamera);

this.mMsg.draw(this.mCamera);

}

// The update function, updates the application state. Make sure to \_NOT\_ draw

// anything from this function!

update() {

let zoomDelta = 0.05;

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

this.mLMinion.update(); // for sprite animation

this.mRMinion.update();

this.mHero.update(); // for WASD movement

this.mPortal.update( // for arrow movement

engine.input.keys.Up,

engine.input.keys.Down,

engine.input.keys.Left,

engine.input.keys.Right

);

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

You can now run the project and move the Hero object with the WASD keys. Move the Hero object toward the WC bounds to observe the camera being pushed. Continue pushing the camera with the Hero object; notice that because of the clampAtBoundary() function call, the Portal object will in turn be pushed such that it never leaves the camera WC bounds. Now press the L/R key to observe the camera center switching to the center on the Left or Right minion. The N/M keys demonstrate straightforward zooming with respect to the center. To experience zooming with respect to a target, move the Hero object toward the top left of the canvas and then press the H key to select it as the zoom focus. Now, with your mouse pointer pointing at the head of the Hero object, you can press the K key to zoom out first and then the J key to zoom back in. Notice that as you zoom, all objects in the scene change positions except the areas around the Hero object. This is a convenient functionality to support zooming into a desired region of your game. You can experience moving the Hero object around while zooming into/away from it.

Kelvin:

Small file changes which may distract from 7.1 goal. Do we need these/can they be cleaner from ch6 to 7???

brain.js – speed changed

this.setSpeed(0.3);

texture\_object.js – removed rotation pararm

update(up, down, left, right) {

let xform = this.getXform();

if (engine.input.isKeyPressed(up)) {

xform.incYPosBy(this.kDelta);

}

if (engine.input.isKeyPressed(down)) {

xform.incYPosBy(-this.kDelta);

}

if (engine.input.isKeyPressed(left)) {

xform.incXPosBy(-this.kDelta);

}

if (engine.input.isKeyPressed(right)) {

xform.incXPosBy(this.kDelta);

}

}

# Interpolation

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

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

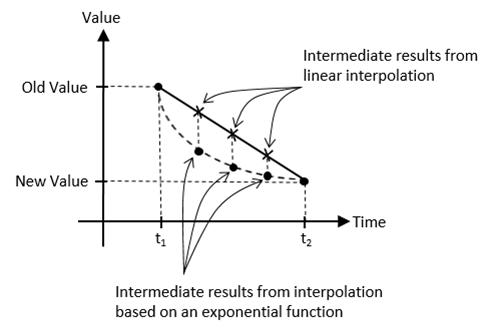


Figure 7-4. Interpolating values based on linear and exponential functions

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

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

## The Camera Interpolations Project

This project demonstrates the smoother and visually more pleasing interpolated results from camera manipulation operations. You can see an example of this project running in Figure 7-5. The source code to this project is defined in the chapter7/7.2.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 interpolated results between given values
* To implement interpolation supporting gradual camera parameter changes
* To experience interpolated changes in camera parameters

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

### Interpolation as a Utility

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

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

#### The Lerp Object

Define the Lerp object to compute interpolation between two values.

1. Create a new file in the src/engine/utils folder and name it lerp.js. Add the following code to construct the object:

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;

// if there is a new value to interpolate to, number of cycles left for interpolation

this.mCyclesLeft = 0;

}

// … implementation to follow …

}

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

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. However, since during each iteration the mCurrentValue is updated to the computed intermediate result, over the entire interpolation cycle mCurrentValue changes to the mFinalValue following an exponential function.

1. Define relevant getter and setter functions.

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 each update.

update() {

if (this.mCyclesLeft <= 0) {

return;

}

this.mCyclesLeft--;

if (this.mCyclesLeft === 0) {

this.mCurrentValue = this.mFinalValue;

} else {

this.\_interpolateValue();

}

}

1. TEMP TEXT

config(stiffness, duration) {

this.mRate = stiffness;

this.mCycles = duration;

}

1. TEMP TEXT

export default Lerp;

1. TEMP TEXT

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

export default {

// …

// general utils

Lerp, LerpVec2,

// …

}

#### The LerpVec2 Object

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

1. Create a new file in the src/engine/utils folder and name it lerp\_vec2.js. Add the following to construct the object as a child of Lerp:

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 for each of the x and y components of vec2 with identical calculations as the \_interpolateValue() function in the Lerp object.

1. TEMP TEXT

export default LerpVec2;

1. TEMP TEXT

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

export default {

// …

// general utils

Lerp, LerpVec2,

// …

}

### Represent Interpolated Intermediate Results with the CameraState Object

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

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

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 Lerp class and are capable of interpolating and computing intermediate results that achieve gradual changes.

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

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 the function to configure the interpolation.

config(stiffness, duration) {

this.mCenter.config(stiffness, duration);

this.mWidth.config(stiffness, duration);

}

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

### Integrate Interpolation into Camera Manipulation Operations

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

1. TEMP TEXT

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

const eViewport = Object.freeze({

eOrgX: 0,

eOrgY: 1,

eWidth: 2,

eHeight: 3

});

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);

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. TEMP TEXT

setWCCenter(xPos, yPos) {

let p = vec2.fromValues(xPos, yPos);

this.mCameraState.setCenter(p);

}

getWCCenter() { return this.mCameraState.getCenter(); }

setWCWidth(width) { this.mCameraState.setWidth(width); }

getWCWidth() { return this.mCameraState.getWidth(); }

getWCHeight() {

// viewportH/viewportW

let ratio = this.mViewport[eViewport.eHeight] / this.mViewport[eViewport.eWidth];

return this.mCameraState.getWidth() \* ratio;

}

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. TEMP TEXT

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) {

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);

}

this.mCameraState.setCenter(newC);

}

}

1. TEMP TEXT

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

let delta = [];

let newC = [];

vec2.sub(delta, pos, this.getWCCenter());

vec2.scale(delta, delta, zoom - 1);

vec2.sub(newC, this.getWCCenter(), delta);

this.zoomBy(zoom);

this.mCameraState.setCenter(newC);

}

### Testing Interpolations in MyGame

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

1. Modify the MyGame init() function.

init() {

// Step A: set up the cameras

this.mCamera = new engine.Camera(

vec2.fromValues(50, 37.5), // position of the camera

100, // width of camera

[0, 0, 640, 480] // viewport (orgX, orgY, width, height)

);

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

// sets the background to gray

// Large background image

let bgR = new engine.SpriteRenderable(this.kBg);

bgR.setElementPixelPositions(0, 1024, 0, 1024);

bgR.getXform().setSize(150, 150);

bgR.getXform().setPosition(50, 35);

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

// Objects in the scene

this.mBrain = new Brain(this.kMinionSprite);

this.mHero = new Hero(this.kMinionSprite);

this.mPortal = new TextureObject(this.kMinionPortal, 50, 30, 10, 10);

this.mLMinion = new Minion(this.kMinionSprite, 30, 30);

this.mRMinion = new Minion(this.kMinionSprite, 70, 30);

this.mFocusObj = this.mHero;

this.mMsg = new engine.FontRenderable("Status Message");

this.mMsg.setColor([1, 1, 1, 1]);

this.mMsg.getXform().setPosition(1, 2);

this.mMsg.setTextHeight(3);

}

1. Modify the MyGame draw() function.

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.mCamera.setViewAndCameraMatrix();

// Step C: Draw everything

this.mBg.draw(this.mCamera);

this.mHero.draw(this.mCamera);

this.mBrain.draw(this.mCamera);

this.mPortal.draw(this.mCamera);

this.mLMinion.draw(this.mCamera);

this.mRMinion.draw(this.mCamera);

this.mMsg.draw(this.mCamera);

}

1. Modify the MyGame update() function.

update() {

let zoomDelta = 0.05;

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

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

// …

}

The previous 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 uninterrupted interpolated results mean the rendered image never abruptly changes and the sense of continuation in space from before and after the user’s camera manipulation commands is preserved. You can try changing the stiffness and duration variables to better appreciate the different rates of interpolation convergence.