Lab 04: *JavaScript* Frontend

Platform Jump

A Physics-based Gravity Action Game - (JavaScript + Canvas API)

By Ted Holmberg *(eholmber@uno.edu)*

Table of Content: *Platformer Game in JavaScript & Canvas API*

By Ted Holmberg *(eholmber@uno.edu)*

Lab Introduction

Prerequisites

Object-Oriented Programming . Software Requirements: Chrome browser, any code editor/IDE.

Motivation

Learn to use modern JavaScript to design a browser app with an Object-Oriented Architecture.

Goal

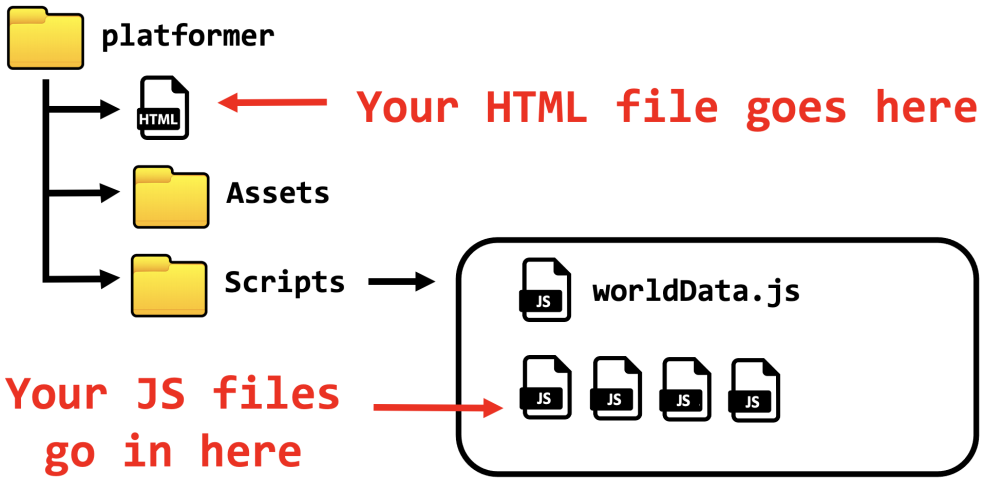
Build a Platformer Game that runs in Browser using JavaScript & Canvas API.

Learning Objectives

- JavaScript Basics
- Objects & Classes in JavaScript
- Inheritance & Polymorphism in JavaScript
- Applications of Data Structures in JavaScript
- Canvas API for drawable viewport & user inputs
- Designing & Implementing a Browser App

Project Architecture:

Start this project by downloading the starter files from github. See the project structure below.



Download Starter files:
https://github.com/scalemailted/platformer-js/archive/master.zip

Concepts

JavaScript (Programming Language)

Web browsers include a builtin JavaScript interpreter capable of both general-purpose computations and domain-specific processing (i.e. Browser-biased). JavaScript supports paradigms: Imperative, Procedural, Object-Oriented, & Functional.

- **Imperative:** JavaScript code may be defined & executed statement-by-statement.
- **Procedural:** JavScript code may organized & defined within functions
- **Components:** JavaScript code may be organized & defined as Objects & Classes
- **Functional:** JavaScript code may be cascaded into chained function calls, i.e. stateless.
- **Using JavaScript:** Link JavaScript files to the HTML document using the script element.
    - **Head:** Link the JS file to head if no dependencies in the body
    - **Body:** Link the JS file to body if it had dependencies to HTML elements in the body

---

Canvas API

Browsers provide a javascript-enabled viewport called HTML canvas element. This element supports real-time 2d or 3d graphics. User input may also be captured from the canvas element. More info: https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API

- **view:** Canvas provides real-time rendering of computer graphics & GUI elements.
- **controls:** Browsers also provide builtin API for capturing "user events" from input devices

---

Example Browser Game

Several examples of browser games exist online. Below is one builtin to all Chrome Browsers!

Examples: **about:dino** ***(Chrome Browser)***

By Ted Holmberg *(eholmber@uno.edu)*

Goal 0: OOP Design in JS

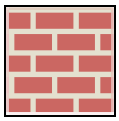
Approach' → Plan phase

Game Objective:
In this lab, we'll build a platformer game. The player must jump on blocks to navigate to the exit. The player must avoid floor and ceiling hazards to prevent a game over. If the player gets to the exit then another stage will load, until the player exits the last stage and wins. The player is affected by gravity and friction. This game is designed using inheritance and polymorphism to store common data for each type of related object.

'Apply' → Do phase

Step 1: Determine the Visual Objects modeled for the Application

Game Objects (from Player Perspective):

	Player character The player can move left, right and jump up. The player is affected by physics i.e. gravity and friction.
	Brick Block The player collides with blocks and cannot move through them. These types of blocks build floor, walls, ceiling, and platforms
	Ceiling Hazard, (type of block) Player dies if they collide with the spike
	Floor Hazard, (type of block) Player dies if they collide with the spike
	Exit, (type of block) Player advances to next stage if they collide with this block

By Ted Holmberg *(eholmber@uno.edu)*

Step 2: Determine the Data Objects modeled for the Application

Overview of Classes

`Game`	*Manages Game data and game methods*	**`GameObject`**	*Manages all common data for game objs*
`World`	*Manages All level setup data*	**`Player`**	*Extends GameObject to define Player*
`Scene`	*Manages Scene data and Scene methods*	**`Controller`**	*Manages Player inputs to move player*
`Block`	*Extends GameObject to define a Block*	**`Physics`**	*Manages physics: gravity, friction, velocity*
`Exit`	*Extends Block to define Exit*	**`Animation`**	*Manages which image to display*
`Ceiling Hazard`	*Extends Block to define Ceiling Hazard*	**`Pose`**	*Contains all possible animated poses*
`Floor Hazard`	*Extends Block to define Floor Hazard*	**`StdDraw`**	*Library Class for drawing graphics*

Config File *(worldData.js)*

Contains data for all the levels in the game.

```
3
14 12
############
#VV ! VV   #
#   #      #
#      AA  #
#     ###  #
##     ##  #
#AA   # #  #
###   V  # #
#VV    ##  #
#    #  AAA#
#AA  V  ####
####       #
#@         #
############
14 12
############
#VV A VV   #
#   #      #
#   V  A   #
#     ###  #
##      #  #
#AA   # #@ #
###   V  # #
#VV    ##  #
#    #  AAA#
#AA  V  ####
####       #
#!         #
############
14 12
############
#          #
#@         #
#          #
#          #
#          #
#          #
#          #
#          #
#          #
#          #
#          #
#AAAAAAAAA!#
############
```

```
Number of Levels, THREE
Number of Rows: 14, Number of Columns: 12
Text representation of level 1, each character represents a tile type
    -   #   WALL
    -   @   PLAYER
    -   !   EXIT
    -   V   CEILING HAZARD
    -   A   FLOOR HAZARD



Number of Rows: 14, Number of Columns: 12
Text representation of level 2, each character represents a tile type
    -   #   WALL
    -   @   PLAYER
    -   !   EXIT
    -   V   CEILING HAZARD
    -   A   FLOOR HAZARD



Number of Rows: 14, Number of Columns: 12
Text representation of level 3, each character represents a tile type
    -   #   WALL
    -   @   PLAYER
    -   !   EXIT
    -   V   CEILING HAZARD
    -   A   FLOOR HAZARD
```

Goal 1: HTML

'Approach' → Plan phase

Canvas is an html element which provides JS code to draw graphics into the browser's viewport.

'Apply' → Do phase

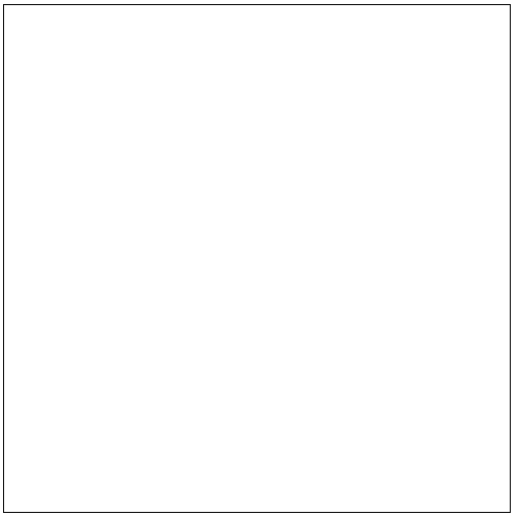
Step 1: Create a Platformer.html file

Platformer.html

```
<!DOCTYPE html>
<html lang="en">
   <head>
       <meta charset="utf-8">
       <title>Platformer</title>
   </head>

   <body>
       <canvas id='viewport' width="500" height="500" style='border: 1px solid black'></canvas>
   </body>
</html>
```

'Assess' → Test phase



Test Instructions:

Open Platformer.html in Browser

Expected Output (Viewport):

The canvas should render as 500x500 drawable space. An black outline should show the boundaries of canvas

(See screenshot on the left)

Goal 2: *class* View

'Approach' → Plan phase

This JS class is responsible for calling the browser's Canvas API to render images into the HTML canvas.

'Apply' → Do phase

Define a view class that represents the game's renderer. Then use it to draw objects to the viewport

Step 1: class: `View` - contains instance variables to the Canvas API browser calls

View.js → instance variables

```
class View {
    /* View Properties */
    #canvas;
    #context;
}

const view = new View();
```

Step 2: method: `View.constructor` - that instantiates an instance of View

View.js → constructor

```
constructor(){
    this.#canvas = document.getElementById("viewport");
    this.#context = this.#canvas.getContext("2d");
}
```

Step 3: method: `View.picture` - draws an image to the HTML canvas

View.js → picture

```
picture( img, x, y, width, height){
    this.#context.drawImage(img, x, y, width, height);
}
```

Step 4: update HTML - add the View.js file as a script element to the body

Platformer.html → <body>

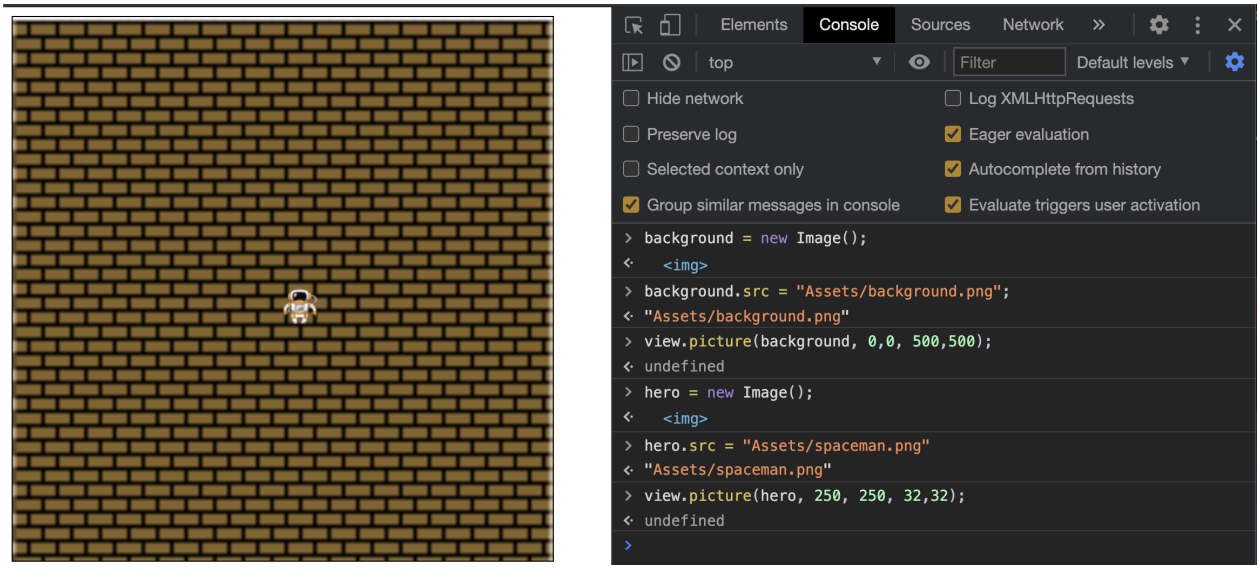
```
<body>
    <canvas id='viewport' width="500" height="500" style='border: 1px solid black'></canvas>
    <script src='Scripts/View.js'>      </script>
</body>
```


By Ted Holmberg *(eholmber@uno.edu)*

'Assess' → Test phase

Testing:

Open Platformer.html in the browser. In the console, type the code shown below & verify the viewport shows images.



Goal 3: *class* Game

'Approach' → Plan phase

Create a Game class that manages all of the game data and game rules.

'Apply' → Do phase

Create a Game class & stub out the necessary methods and attributes to get a game loop to start executing.

Step 1: class: `Game` - contains an instance variable for the game over condition

Game.js → instance variables

```
class Game {
   /* Properties */
    #isOver;
}

const game = new Game();
```

Step 2: method: `Game.constructor` - that instantiates an instance of Game

Game.js → constructor

```
/* Create a new Platform Game */
constructor() {
    this.#isOver = false;
}
```

Step 3: method: `Game.update` - responsible for updating the game state during the game-loop

Game.js → update()

```
update() {
    console.log("Game Update");   //Test - delete this line after finishing this goal.
}
```

Step 4: method: `Game.render` - responsible for drawing the game during the game-loop

Game.js → render()

```
render() {
    console.log("Game Render");   //Test - delete this line after finishing this goal.
}
```

Step 5: method: `Game.main` - the game loop, this is a static method that recursively calls itself.

Game.js → main()

```
/* The main game loop (static method) */
static main() {
    if (game.isOver === false){
        game.update();
        game.render();
        window.requestAnimationFrame(Game.main);
    }
    else{
       console.log("Game Over!");
    }
}
```

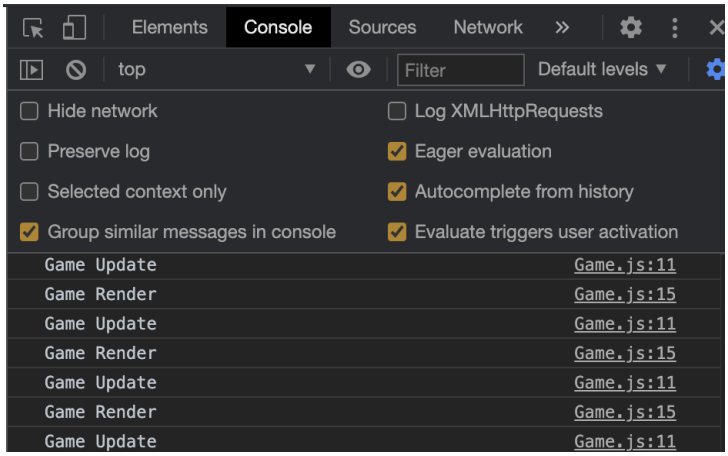
Step 6: update HTML - add the Game.js as a script element to the body, & invoke Game.main

Platformer.html → <body>

```
<body>
    <canvas id='viewport' width="500" height="500" style='border: 1px solid black'></canvas>
    <script src='Scripts/View.js'>      </script>
    <script src='Scripts/Game.js'> </script>
    <script>
        window.addEventListener('load', Game.main );
    </script>
</body>
```

'Assess' → Test phase

Testing:



Test Instructions:

Open `Platformer.html` in Browser
Use DevTools to access the JS console

Expected Output (console):

Infinite console logs should print from the game's update & render methods.

(See illustration on the left)

Clean Up:

After testing, remove the `console.logs`

Goal 4: *class* World

'Approach' → Plan phase

Create a World class that parses & manages all of the game world data.

'Apply' → Do phase

Parse the configuration data, save it into a multidimensional array, & display into the console

Step 1: class: `World` - contains instance variable to store the game levels

World.js → instance variables

```
class World{
   /* World Properties */
   #levels;
}
```

Step 2: method: `World.constructor` - instantiates an instance of World

World.js → constructor

```
constructor(){
    //parse all map data and save it for later
    this.#levels = world.map( level => (level.split('\n')).map(row => row.split("")) );
}
```

Step 3: props: `Game` - add instance variable to store a World instance

Game.js → instance variables

```
/*Game Properties */
#world;
#isOver;
```

Step 4: method: `Game.constructor` - set the instance variable for world

Game.js → constructor

```
/*Create a new Platform Game*/
constructor() {
    this.#isOver = false;
    this.#world = new World();
    console.log('world', this.#world);   //Test - delete this line after finishing this goal.
}
```


By Ted Holmberg *(eholmber@uno.edu)*

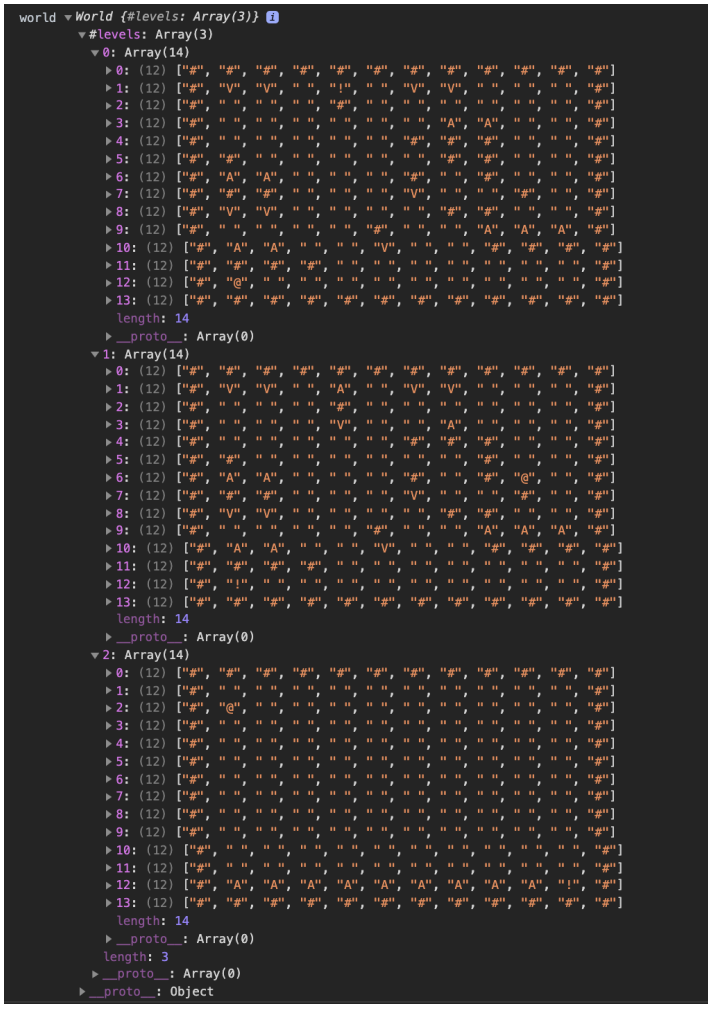
Step 5: update HTML - add worldData.js & World.js as script elements to the body

Platformer.html → <body>

```
<body>
    <canvas id='viewport' width="500" height="500" style='border: 1px solid black'></canvas>
    <script src='Scripts/worldData.js'>  </script>
    <script src='Scripts/World.js'>      </script>
    <script src='Scripts/View.js'>      </script>
    <script src='Scripts/Game.js'>       </script>

    <script>
        window.addEventListener('load', Game.main );
    </script>
</body>
```

'Assess' → Test phase



Test Instructions:

Open Platformer.html in Browser
Use DevTools to access JS console

Expected Output (console):

The console logs the game's world instance which contains the 'per tile' parsings of all three level maps.

Unpack the array data to see its contents (See illustration on the left)

Clean Up:

After testing, remove the `console.logs`

Goal 5: *class* Scene

'Approach' → Plan phase

Define Scene class that manages the data for each playable game scene

'Apply' → Do phase

Step 1: class: `Scene` - manages over a playable game level

Scene.js → instance variables

```
class Scene {
    /* Scene Properties */
}
```

Step 2: method: `Scene.constructor` - instantiates an instance of Scene

Scene.js → constructor

```
constructor( map ) {
    this.setScene( map );
}
```

Step 3: method: `Scene.setScene` - helper method to parse each tile in its 2d grid model

Scene.js → setScene

```
setScene (worldData){
    const cols = worldData[0].length;          //Count of tiles height of world
    const rows = worldData.length;             //Count of tiles across of world

    for (let y=0; y < rows; y++){
        for (let x=0; x < cols; x++){
            const tile = worldData[y][x];
            console.log(tile);                 //Test - delete this line after finishing this goal.
        }
    }
}
```

Step 4: method: `World.getLevel` - returns the level from the levels Array given a level number

World.js → getLevel()

```
getLevel(level){
    return this.#levels[level];
}
```


By Ted Holmberg *(eholmber@uno.edu)*

Step 5: prop: `Game` - add instance variables for current level number & the scene object.

Game.js → instance variables

```
/* Game Properties */
#world;
#isOver;
#level;
#scene;
```

Step 6: method: `Game.constructor` - set each of the new instance variables.

Game.js → constructor

```
/*Create a new Platform Game*/
constructor() {
     this.#isOver = false;
     this.#world = new World();
     this.#level = 0;
     const levelData = this.#world.getLevel( this.#level );
     this.#scene = new Scene(levelData);
}
```

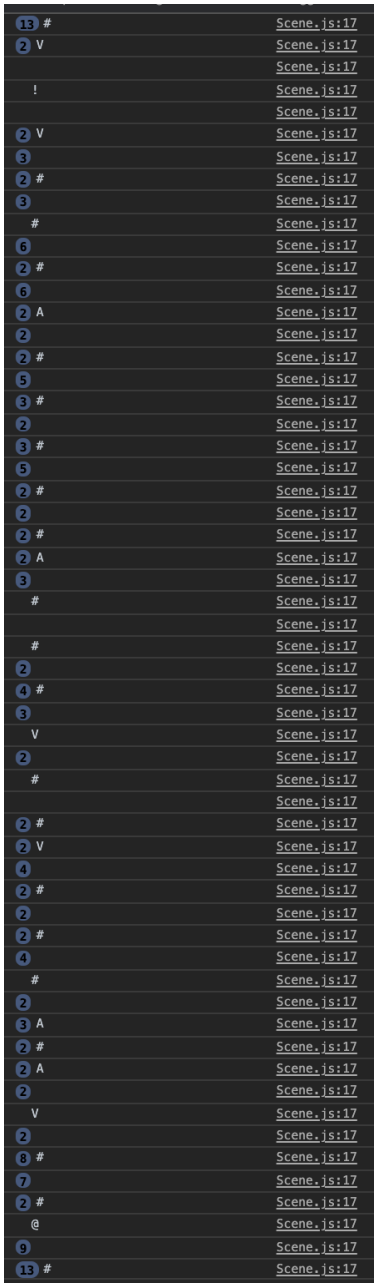
Step 7: update HTML - add Scene.js as script elements to the body

Platformer.html → <body>

```
<body>
    <canvas id='viewport' width="500" height="500" style='border: 1px solid black'></canvas>
    <script src='Scripts/worldData.js'>  </script>
    <script src='Scripts/World.js'>      </script>
    <script src='Scripts/Scene.js'>      </script>
    <script src='Scripts/View.js'>       </script>
    <script src='Scripts/Game.js'>       </script>

    <script>
         window.addEventListener('load', Game.main );
    </script>
</body>
```

'Assess' → Test phase



Test Instructions:

Open `Platformer.html` in Browser
Use DevTools to access the JS console

Expected Output (console):

The console logs each tile from the first level map
For duplicate strings, the count is given
(See illustration on the left)

Clean Up:

After testing, remove the `console.logs`

Goal 6: *class* GameObject

'Approach' → Plan phase

Define a class GameObject managing the data that all game objects share.

'Apply' → Do phase

Define a Game object that represents any type of object in the game. Then use it to build background object in scene

Step 1: class: `GameObject` - manages over data all game object's use to draw images into the canvas

GameObject.js → instance variables

```
class GameObject {
    /* Game Object Properties */
    #x;
    #y;
    #width;
    #height;
    #image;
}
```

Step 2: method: `GameObject.constructor` - instantiates an instance of GameObject

GameObject.js → constructor

```
constructor(x, y, width, height, src) {
        this.#x = x;
        this.#y = y;
        this.#width = width;
        this.#height = height;
        this.#image = new Image();
        this.#image.src= src;
}
```

Step 3: method: `GameObject.draw` - uses the view instance to draw its image into the canvas

GameObject.js → draw() method

```
draw() {
    view.picture(this.#image, this.#x, this.#y, this.#width, this.#height);
}
```

Step 4: props: `Scene` - add an instance variable to represent a scene's background

Scene.js → instance variables

```
/* Scene Properties */
#background;
```


By Ted Holmberg *(eholmber@uno.edu)*

Step 5: method: `Scene.setBackground` - sets background with a new GameObject based on cols/rows

Scene.js → setBackground

```
setBackground( rows, cols, img="Assets/background.png" , tileSize=32 ){
    const width = cols * tileSize;
    const height = rows * tileSize;
    this.#background = new GameObject( 0, 0, width, height, img);
}
```

Step 6: method: `Scene.setScene` - invoke the setBackground from the setScene method

Scene.js → setScene

```
setScene (worldData){
    const cols = worldData[0].length;          //Count of tiles height of world
    const rows = worldData.length;             //Count of tiles across of world
    this.setBackground(rows, cols);

    for (let y=0; y < rows; y++){
        for (let x=0; x < cols; x++){
            const tile = worldData[y][x];
        }
    }
}
```

Step 7: method: `Scene.draw` - manages the drawing of all drawable objects in the game scene

Scene.js → draw() method

```
draw() {
    this.#background.draw();
}
```

Step 8: method: `Game.render` - manages the drawing of all renderable data in the game

Game.java → render()

```
render() {
    this.#scene.draw();
}
```

Step 9: update HTML - add GameObject.js as a script element to the body

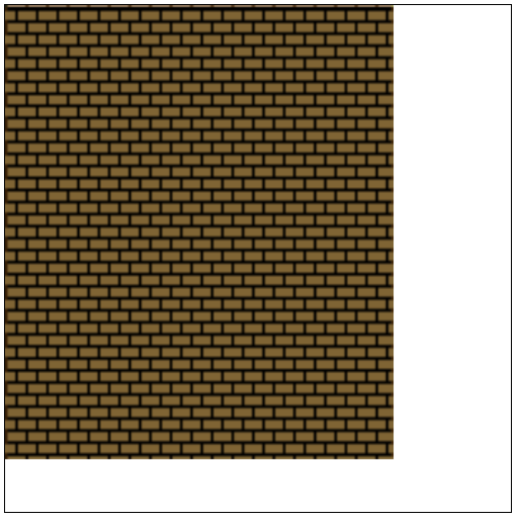
Platformer.html → <body>

```
<body>
    <canvas id='viewport' width="500" height="500" style='border: 1px solid black'></canvas>
    <script src='Scripts/worldData.js'>  </script>
    <script src='Scripts/World.js'>      </script>
    <script src='Scripts/GameObject.js'> </script>
    <script src='Scripts/Scene.js'>      </script>
    <script src='Scripts/View.js'>       </script>
    <script src='Scripts/Game.js'>       </script>

    <script>
         window.addEventListener('load', Game.main );
    </script>
</body>
```


By Ted Holmberg *(eholmber@uno.edu)*

'Assess' → Test phase



Test Instructions:

Open `Platformer.html` in Browser

Expected Output:

GUI window display with:

- ***Background image***

Goal 7: *class* Block

'Approach' → Plan phase

Model the basic type of brick block in the game by extending from the GameObject class

'Apply' → Do phase

Create Block class by extending GameObject and draws blocks in scene

Step 1: class: `Block` - models a brick block in the game. Every block is the same size to fit the grid model

Block.js → instance variables

```
class Block extends GameObject {
    /* Block Properties */
    static SIZE = 32;
}
```

Step 2: method: `Block.constructor` - instantiates an instance of Block, invokes super constructor

Block.js → constructor

```
constructor(x, y, image="Assets/tile-brick.png") {
    super( x * Block.SIZE, y * Block.SIZE, Block.SIZE, Block.SIZE, image);
}
```

Step 3: props: `Scene` - add instance variable to Scene to store all the blocks

Scene.js → instance variables

```
/* Scene Properties */
#background;
#blocks;
```

Step 4: method: `Scene.constructor` - set blocks property to an empty array.

Scene.js → constructor

```
constructor(map){
    this.#blocks = [];
    this.setScene( map );
}
```

Step 5: method: `Scene.setScene` - send each tile to a helper method to parse into a game object

Scene.js → setScene

```
setScene (worldData){
    const cols = worldData[0].length;          //Count of tiles height of world
    const rows = worldData.length;             //Count of tiles across of world
    this.setBackground(rows, cols);

    for (let y=0; y < rows; y++){
        for (let x=0; x < cols; x++){
            const tile = worldData[y][x];
            this.setTile(x, y, tile);
        }
    }
}
```

Step 6: method: `Scene.setTile` - factory pattern: on case of "#" then create a block object at that coord

Scene.js → setTile

```
setTile (x, y, tile){
    switch(tile){
        case "#": this.#blocks.push( new Block(x,y) );  break;
     }
}
```

Step 7: method: `Scene.draw` - for each block in blocks array invoke its draw method

Scene.js → draw() method

```
draw(){
   this.#background.draw();
   this.#blocks.forEach( (block) => block.draw() );
 }
```

Step 8: update HTML - add Block.js as a script element to the body

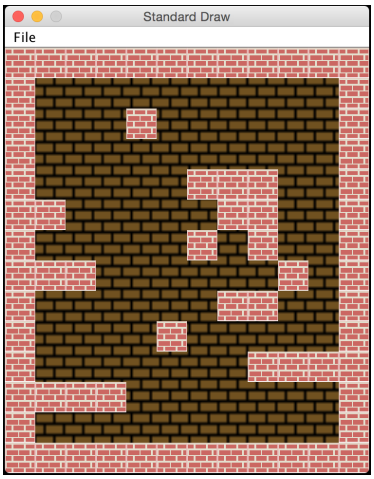
Platformer.html → <body>

```
<body>
    <canvas id='viewport' width="500" height="500" style='border: 1px solid black'></canvas>
    <script src='Scripts/worldData.js'>  </script>
    <script src='Scripts/World.js'>      </script>
    <script src='Scripts/GameObject.js'> </script>
    <script src='Scripts/Block.js'>      </script>
    <script src='Scripts/Scene.js'>      </script>
    <script src='Scripts/View.js'>       </script>
    <script src='Scripts/Game.js'>       </script>

    <script>
         window.addEventListener('load', Game.main );
    </script>
</body>
```


By Ted Holmberg *(eholmber@uno.edu)*

'Assess' → Test phase



Test Instructions:

Open `Platformer.html` in Browser

Expected Output:

GUI window display with:

- *Background image*
- ***Brick blocks***

Goal 8: *class* Player

'Approach' → Plan phase

Model the Player in the game by extending from the GameObject class

'Apply' → Do phase

Create Player class by extending GameObject class and draw player in scene

Step 1: class: `Player` - models the player in the game.

Player.js

```
class Player extends GameObject {
    /* Player Properties */
}
```

Step 2: method: `Player.constructor` - used to instantiate Player, invokes super constructor

Player.js → constructor

```
constructor(x, y) {
    super( x*Block.SIZE, y*Block.SIZE, Block.SIZE, Block.SIZE, "Assets/link-down.png");
}
```

Step 3: props: `Scene` - add a instance variable to store the player object

Scene.js → instance variables

```
/* Scene Properties */
#background;
#blocks;
#player;
```

Step 4: method: `Scene.setTile` - in case of "@", then instantiate a player object at that coord.

Scene.js → setTile

```
setTile (x, y, tile){
    switch(tile){
        case "#": this.#blocks.push( new Block(x,y) );  break;
        case "@": this.#player = new Player(x,y);       break;
    }
}
```


By Ted Holmberg *(eholmber@uno.edu)*

Step 5: method: `Scene.draw` - invoke player to draw itself

Scene.js → draw() method

```
draw(){
   this.#background.draw();
   this.#blocks.forEach( (block) => block.draw() );
   this.#player.draw();
 }
```

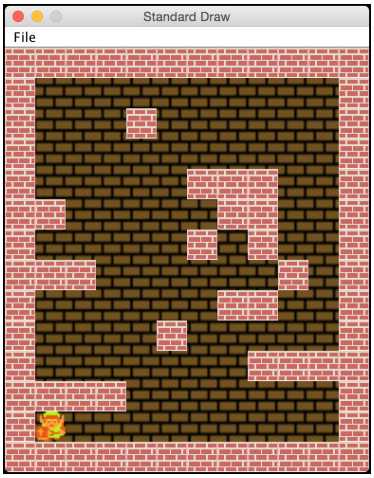
Step 6: update HTML - add Player.js as a script element to the body

Platformer.html → <body>

```
<body>
    <canvas id='viewport' width="500" height="500" style='border: 1px solid black'></canvas>
    <script src='Scripts/worldData.js'>  </script>
    <script src='Scripts/World.js'>      </script>
    <script src='Scripts/GameObject.js'> </script>
    <script src='Scripts/Block.js'>      </script>
    <script src='Scripts/Player.js'>     </script>
    <script src='Scripts/Scene.js'>      </script>
    <script src='Scripts/View.js'>       </script>
    <script src='Scripts/Game.js'>       </script>

    <script>
         window.addEventListener('load', Game.main );
    </script>
</body>
```

'Assess' → Test phase



Test Instructions:

Open `Platformer.html` in Browser

Expected Output:

GUI window display with:

- *Background image*
- *Brick blocks*
- ***Player character***

Goal 9: *class* Physics

'Approach' → Plan phase

Design the physics system of a platform game. Gameplay relies on pushing the character with velocities in X-axis & Y-axis. Different forces influence the players velocity such as gravity, constantly pulling the player downward. Velocity affects the players position in the world, it represents the momentum they have for movement.

'Apply' → Do phase

Create Physics class and add to player using composition

Step 1: class: `Physics` - models the physical interactions between objects of the game world.

Physics.js → instance variables

```
class Physics {
    /* Physics Properties */
    #speed;
    #gravity;
    #terminal;
    #velocityX;
    #velocityY;
}
```

Step 2: method: `Physics.constructor` - used to instantiate Physics, parameter of speed passed in.

Physics.js → constructor

```
constructor(speed) {
    this.#speed = speed;
    this.#gravity = 0.3;
    this.#terminal = 8;
    this.#velocityX = 0.0;
    this.#velocityY = 0.0;
}
```

Step 3: method: `Physics.applyGravity` - if downward velocity isn't terminal then increase it

Physics.js →applyGravity

```
applyGravity() {
   if (this.#velocityY < this.#terminal) {
        this.#velocityY += this.#gravity;
   }
}
```

Step 4: method: `Physics.update` - Each iteration of game loop, the physics should update its state.

Physics.js →update

```
update() {
    this.applyGravity();
}
```


By Ted Holmberg *(eholmber@uno.edu)*

Step 5: method: `Physics getters` - provides access to the velocity values from outside the class.

Physics.js →getter methods

```
getVelocityX()  {
    return this.#velocityX;
}

getVelocityY()  {
    return this.#velocityY;
}
```

Step 6: method: `GameObject getters` - provides access to the x,y coord values from outside the class

GameObject.js →getter methods

```
getX()  {
    return this.#x;
}

getY()  {
    return this.#y;
}
```

Step 7: method: `GameObject.move` - move the game object to the given x,y coords

GameObject.js → move

```
move(x, y) {
    this.#x = x;
    this.#y = y;
}
```

Step 8: props: `Player` - add physics instance into the player class

Player.js → instance variables

```
/* Player Properties */
#physics;
```

Step 9: method: `Player.constructor` - instantiate Physics instance in constructor

Player.js → constructor

```
constructor(x, y) {
    super( x*Block.SIZE, y*Block.SIZE, Block.SIZE, Block.SIZE, "Assets/link-down.png");
    this.#physics = new Physics(4);
}
```

Step 10: method: `Player.move` - override the move method in Player to use the Physics' velocities

Player.js → move

```
move() {
    const dx = this.getX() + this.#physics.getVelocityX();
    const dy = this.getY() + this.#physics.getVelocityY();
    super.move(dx,dy);
}
```

Step 11: method: `Player.update` - Each pass of game-loop the player should update its state

Player.js → update

```
update() {
    this.#physics.update();
    this.move();
}
```

Step 12: method: `Scene.update` - Each pass of game-loop the Scene should update all its game objects

Scene.js → update

```
update() {
    this.#player.update();
}
```

Step 13: method: `Game.update` - Each pass of game-loop the Game should invoke the Scene's update

Game.js → update

```
update() {
    this.#scene.update();
}
```

Step 14: update HTML - add Physics.js as a script element to the body

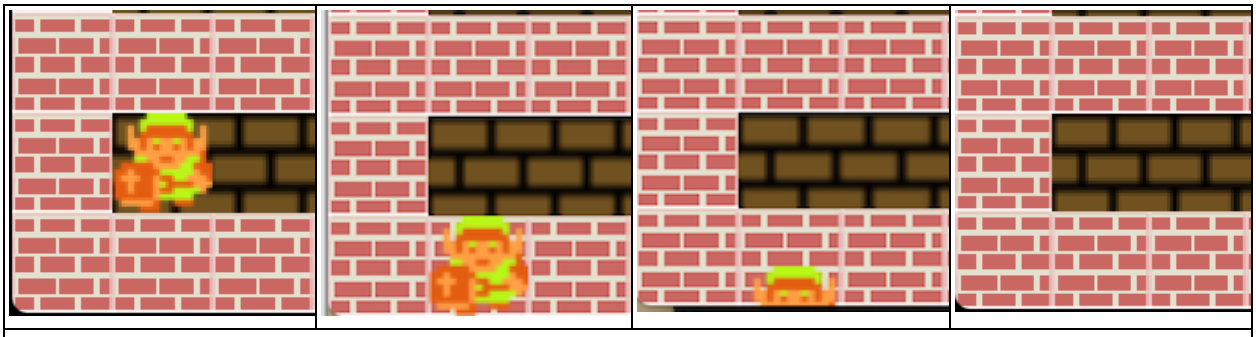
Platformer.html → <body>

```
<body>
    <canvas id='viewport' width="500" height="500" style='border: 1px solid black'></canvas>
    <script src='Scripts/worldData.js'>  </script>
    <script src='Scripts/World.js'>      </script>
    <script src='Scripts/GameObject.js'> </script>
    <script src='Scripts/Block.js'>      </script>
    <script src='Scripts/Physics.js'>     </script>
    <script src='Scripts/Player.js'>     </script>
    <script src='Scripts/Scene.js'>      </script>
    <script src='Scripts/View.js'>       </script>
    <script src='Scripts/Game.js'>       </script>

    <script>
         window.addEventListener('load', Game.main );
    </script>
</body>
```


By Ted Holmberg *(eholmber@uno.edu)*

'Assess' → Test phase



Test Instructions:

Open `Platformer.html` in Browser

Expected Output:

Character falls down, through the floor and off the screen.

Goal 10: *class* Controller (jump)

'Approach' → Plan phase

Design the controller of the game. We should separate the logic that handles user input from the logic that moves the character. Thus we may have different control schemes for the game without refactoring the code base. This is called a model-view-controller design pattern

'Apply' → Do phase

Create a controller class that listens for player input and responds by issuing commands to the player instance. The controller invokes methods in the player to move it, and the player will relay those calls to its internal physics system which defines the rules for movement.

Step 1: method: `Physics.jump` - a jump action defined as an upward thrust to velocity

Physics.js → jump

```
jump() {
    this.#velocityY = -this.#speed*2;
}
```

Step 2: props: `Player` - a boolean value to track whether the player is currently jumping or not.

Player.js → instance variables

```
/* Player Properties */
#physics;
#isJumping;
```

Step 3: method: `Player.constructor` - initialize "player is jumping" as false.

Player.js → constructor

```
constructor(x, y) {
    super( x*Block.SIZE, y*Block.SIZE, Block.SIZE, Block.SIZE, "Assets/link-down.png");
    this.#physics = new Physics(4);
    this.#isJumping = false;
}
```

Step 4: method: `Player.jump` - if player isn't already jumping then apply physics for a jump

Player.js → jump

```
jump() {
    if (this.#isJumping === false) {
        this.#physics.jump();
        this.#isJumping = true;
    }
}
```


By Ted Holmberg *(eholmber@uno.edu)*

Step 5: class: `Controller` - Models the user input. Move the player based on the inputs.

Controller.js → instance variables

```
class Controller {
    /* Controller Properties */
    #player;
    #inputs;
}
```

Step 6: method: `Controller.constructor` - Player as param. Inputs as new Set. Add Event Listeners

Controller.js → constructor

```
constructor(player) {
    this.#player = player;
    this.#inputs = new Set();
    document.addEventListener("keydown", this.buttonDown );
    document.addEventListener("keyup", this.buttonUp );
}
```

Step 7: method: `Controller.buttonDown` - Callback function. Adds 'up' into set if triggered

Controller.js → buttonDown

```
buttonDown = function(event){
    const inputs = this.#inputs;
    switch(event.keyCode){
        case 38: inputs.add('up');    break;
    }
}.bind(this)
```

Step 8: method: `Controller.buttonUp` - Callback function. Deletes 'up' from set if triggered

Controller.js → buttonUp

```
buttonUp = function(event){
    const inputs = this.#inputs;
    switch(event.keyCode){
        case 38: inputs.delete('up');    break;
    }
}.bind(this)
```

Step 9: method: `Controller.update` - Each pass of game-loop the Game invokes Controller's update

Controller.js → update

```
update(){
    const inputs = this.#inputs;
    if ( inputs.has('up') ){
        this.#player.jump()
    }
}
```

Step 10: method: `Scene.getPlayer` - used to access the Scene's player from outside the class

Scene.js → getPlayer

```
getPlayer() {
    return this.#player;
}
```

Step 11: props: `Game` - add instance variable to game to manage a controller instance

Game.js → instance variables

```
/* Game Properties */
#world;
#isOver;
#level;
#scene;
#controller;
```

Step 12: method: `Game.constructor` - get player from Scene and register it to controller instance

Game.js → constructor

```
/*Create a new Platform Game*/
constructor() {
     this.#isOver = false;
     this.#world = new World();
     this.#level = 0;
     const levelData = this.#world.getLevel( this.#level );
     this.#scene = new Scene(levelData);
     const player = this.#scene.getPlayer();
     this.#controller = new Controller( player );
}
```

Step 13: method: `Game.update` - Game invokes all updates on its properties

Game.java → update

```
update() {
    this.#controller.update();
    this.#scene.update();
}
```

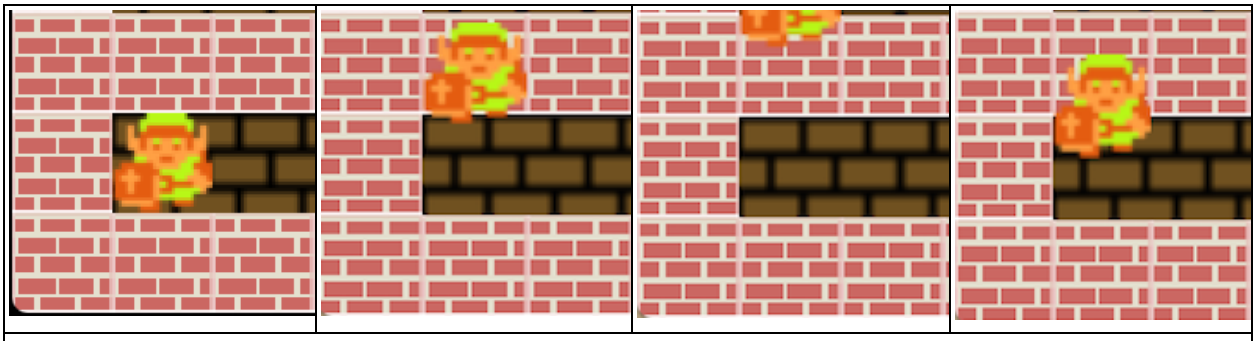
Step 14: update HTML - add Controller.js as a script element to the body

Platformer.html → <body>

```
<body>
    <canvas id='viewport' width="500" height="500" style='border: 1px solid black'></canvas>
    <script src='Scripts/worldData.js'>  </script>
    <script src='Scripts/World.js'>      </script>
    <script src='Scripts/GameObject.js'> </script>
    <script src='Scripts/Block.js'>      </script>
    <script src='Scripts/Physics.js'>    </script>
    <script src='Scripts/Player.js'>     </script>
    <script src='Scripts/Scene.js'>      </script>
    <script src='Scripts/Controller.js'> </script>
    <script src='Scripts/View.js'>       </script>
    <script src='Scripts/Game.js'>       </script>

    <script>
         window.addEventListener('load', Game.main );
    </script>
</body>
```

'Assess' → Test phase



Test Instructions:

Open `Platformer.html` in Browser

Expected Output:

When player presses 'up arrow' then character jumps up, then falls back down, through the floor and off the screen.

Goal 11: *class* Controller (left/right)

'Approach' → Plan phase

Design horizontal movement of player. Requires updating the velocity X of character. We can add a positive number to velocity to go right or a negative number to go left. The more you press in a direction, the more the velocity should increase, moving the character faster in that direction.

'Apply' → Do phase

Add methods that define leftward and rightward movement of character in physics class, then make calls to them in player so that the controller can invoke them.

Step 1: method: `Physics.moveLeft` - an action defined as a leftward thrust in velocity, limited by speed

Physics.js → moveLeft

```
moveLeft() {
    if (this.#velocityX > -this.#speed ) {
        this.#velocityX--;
    }
}
```

Step 2: method: `Physics.moveRight` - an action defined as a leftward thrust in velocity, limited by speed

Physics.js → moveRight

```
moveRight() {
    if (this.#velocityX < this.#speed  ) {
        this.#velocityX++;
    }
}
```

Step 3: method: `Player.moveLeft` - Adapter pattern: provide access to physics moveLeft out of class

Player.js → moveLeft

```
moveLeft() {
    this.#physics.moveLeft();
}
```

Step 4: method: `Player.moveRight` - Adapter pattern: provide access to physics moveRight out of class

Player.js → moveRight

```
moveRight() {
    this.#physics.moveRight();
}
```

Step 5: method: `Controller.buttonDown` - Add 'left'/'right' to inputs based on keycode from button event

Controller.js → buttonDown

```
buttonDown = function(event){
    const inputs = this.#inputs;
    switch(event.keyCode){
        case 38: inputs.add('up');    break;
        case 37: inputs.add('left');  break;
        case 39: inputs.add('right'); break;
    }
}.bind(this)
```

Step 6: method: `Controller.buttonUp` - Delete 'left'/'right' to inputs based on keycode from button event

Controller.js → buttonUp

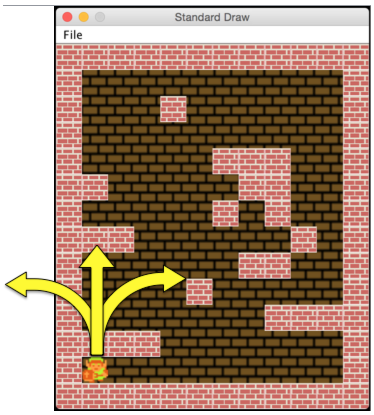
```
buttonUp = function(event){
    const inputs = this.#inputs;
    switch(event.keyCode){
        case 38: inputs.delete('up');    break;
        case 37: inputs.delete('left');  break;
        case 39: inputs.delete('right'); break;
    }
}.bind(this)
```

Step 7: method: `Controller.update` - invoke player to move left or right if inputs has that command.

Controller.js → update

```
update(){
    const inputs = this.#inputs;
    if (inputs.has('left')){
        this.#player.moveLeft();
    }
    if (inputs.has('right')){
        this.#player.moveRight();
    }
    if ( inputs.has('up') ){
        this.#player.jump()
    }
}
```

'Assess' → Test phase



Test Instructions:

Open `Platformer.html` in Browser

Expected Output:

When the player presses up/spacebar then character jumps up,
When the player presses left/right they arc in that direction.
Then falls back down, through the floor and off the screen.

Goal 12: *Floor Collisions*

'Approach' → Plan phase

Design the collisions for the game. There are two concepts: Are two game objects touching? Yes or no. Then if they are, which sides are they touching? Top, Left, Right, or Bottom. There will be four discrete points of collision, it's possible to have multiple sides colliding between two objects, such as when corners touch, that counts as both a vertical and horizontal touch.

'Apply' → Do phase

Physics class should detect collisions between player and blocks before moving player down

Step 1: method: `GameObject getters` - provide access to a game object's width/height values

GameObject.js → getter methods

```
getWidth() {
    return this.#width;
}

getHeight() {
    return this.#height;
}
```

Step 2: method: `Block.isTouchingX` - determines if this block & other object overlap on X by some ratio

Block.js → isTouchingX

```
isTouchingX(gameObject, ratio) {
    const overlap = this.getWidth() * ratio;
    return ( Math.abs( this.getX() - gameObject.getX() ) < overlap  );
}
```

Step 3: method: `Block.isTouchingY` - determines if this block & other object overlap on Y by some ratio

Block.js → isTouchingY

```
isTouchingY(gameObject, ratio) {
    const overlap = this.getHeight() * ratio;
    return ( Math.abs( this.getY() - gameObject.getY() ) < overlap  );
}
```

Step 4: method: `Block.isTouching` - determines if this block & other object overlap in 2d space

Block.js → isTouching

```
isTouching(gameObject) {
    return this.isTouchingY(gameObject, 1.0) && this.isTouchingX(gameObject, 0.75);
}
```

Step 5: method: `Physics.checkCollisions` - checks collisions between all blocks & players

By Ted Holmberg *(eholmber@uno.edu)*

Physics.js → checkCollisions

```
checkCollisions(blocks, player) {
    for (let block of blocks) {
        if (block.isTouching(player) ) {
            this.checkCollisionFloor(block, player);
        }
    }
}
```

Step 6: method: `Physics.checkCollisionFloor` - floor collision stops downward velocity & resets jump

Physics.js → checkCollisionFloor method

```
checkCollisionFloor(block, player){
    if (player.getY() < block.getY() && this.#velocityY > 0 ) {  //player higher than block & falling
        if ( block.isTouchingX(player, 0.5) ) {                  //no wall jump, i.e. player/block on same column.
            this.#velocityY = 0;
            player.isJumping(false);
        }
    }
}
```

Step 7: method: `Physics.update` - Each pass of game-loop should check for collisions between blocks

Physics.js → update

```
update(blocks, player) {
    this.applyGravity();
    this.checkCollisions(blocks, player);
}
```

Step 8: method: `Player.update` - refactor method to pass in references for blocks & itself

Player.js → update

```
update(blocks) {
    this.#physics.update(blocks, this);
    this.move();
}
```

Step 9: method: `Player.isJumping` - setter methods for isJumping to set from outside the class

Player.js → isJumping

```
isJumping(isJumping) {
    this.#isJumping = isJumping;
}
```

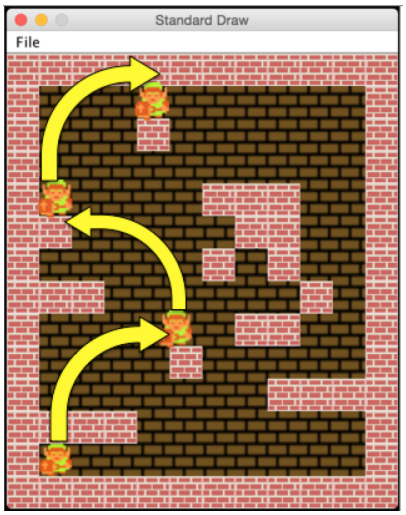
Step 10: method: `Scene.update` - pass blocks array into player's update method

Scene.js → update

```
update() {
    this.#player.update(this.#blocks);
}
```


By Ted Holmberg *(eholmber@uno.edu)*

'Assess' → Test phase



Test Instructions:

Open `Platformer.html` in Browser

Expected Output:

When player presses up/spacebar character jumps up,
When the player presses left/right they move in that direction.
Player lands on top of blocks.

NOTE: *No collisions on blocks' right, left or undersides.*
NOTE: *No friction is applied so player continues moving, never stopping*

Goal 13: Ceiling Collisions

'Approach' → Plan phase

A ceiling collision is when the player and block are touching, and the player is lower than the block (i.e. the player Y axis is larger than the block Y axis) and when the player's Y velocity is moving the player upward (i.e. the velocity Y is a negative number). In response, let's reduce the velocity and push the player in the opposite direction by multiplying the current velocity by a fractional negative number.

'Apply' → Do phase

Physics class should detect collisions between player and blocks before moving player up

Step 1: method: `Physics.checkCollisionCeiling` - Ceiling Collisions reflect velocity downward

Physics.js → checkCollisionCeiling

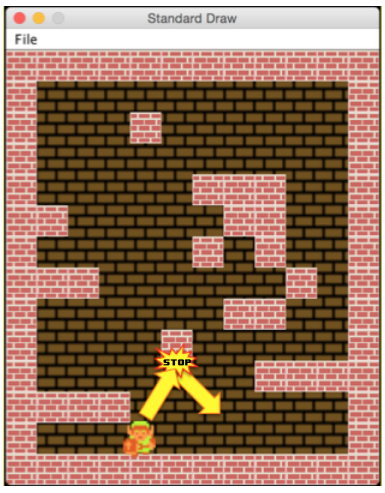
```
checkCollisionCeiling(block, player) {
    if (player.getY() > block.getY() && this.#velocityY < 0) {       //player lower than block & jumping
        this.#velocityY *= -0.5;
    }
}
```

Step 2: method: `Physics.checkCollision` - check for ceiling collision

Physics.js → checkCollisions

```
checkCollisions(blocks, player){
    for (let block of blocks){
        if (block.isTouching(player) ){
            this.checkCollisionFloor(block, player);
            this.checkCollisionCeiling(block, player);
        }
    }
}
```

'Assess' → Test phase



Test Instructions:

Open `Platformer.html` in Browser

Expected Output:

When player presses up/spacebar character jumps up,
When the player presses left/right they move in that direction.
Player lands on top of blocks.
Player bounces back down when jumping under block

NOTE: *No collisions on blocks' right, left.*
NOTE: *No friction is applied so player continues moving, never stopping*

Goal 14: Left/Right Collisions

'Approach' → Plan phase

Define collisions on left/right sides. Three checks are necessary, first are two blocks touching, if so, then if the player X is left of block X and velocity is positive (i.e. moving rightward) then a right collision is occurring, however we want to avoid corner touches (i.e. touching a block on another row than the one the player is on) so check the Y position of both blocks to make sure they are adjacent horizontally.

'Apply' → Do phase

Physics class should detect collisions between the player and blocks before moving the player left/right. Implement a horizontal check by decreasing the touch zone between two objects, at 50% on Y only objects on the same row will trigger a touch detection.

Step 1: method: `Physics.checkCollisionRight` - Right Collisions reflect velocity on X

Physics.js → checkCollisionRight

```
checkCollisionRight(block, player) {
    if (player.getX() < block.getX() && this.#velocityX > 0 ) {  //player left of block & moving right
        if ( block.isTouchingY(player, 0.5) ) {            //ensure player and block on same row
            this.#velocityX *= -1;
        }
    }
}
```

Step 2: method: `Physics.checkCollisionLeft` - Left Collisions reflect velocity on X

Physics.js → checkCollisionLeft method

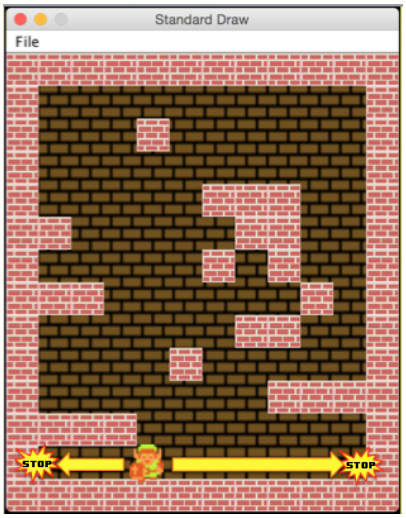
```
checkCollisionLeft(block, player) {
    if (player.getX() > block.getX() && this.#velocityX < 0 ) {   //player right of block & moving left
        if ( block.isTouchingY(player, 0.5) ) {             //ensure player and block on same row
            this.#velocityX *= -1;
        }
    }
}
```

Step 3: method: `Physics.checkCollisions` - check for right and left collisions

Physics.js → checkCollisions

```
checkCollisions(blocks, player){
    for (let block of blocks){
        if (block.isTouching(player) ){
            this.checkCollisionFloor(block, player);
            this.checkCollisionCeiling(block, player);
            this.checkCollisionRight(block, player);
            this.checkCollisionLeft(block, player);
        }
    }
}
```

'Assess' → Test phase



Test Instructions:

Open `Platformer.html` in Browser

Expected Output:

When player presses up/spacebar character jumps up,
When the player presses left/right they move in that direction.
Player lands on top of blocks.
Player bounces back down when jumping under block
Player bounces off the left/right sides of blocks

NOTE: *No friction is applied so player continues moving, never stopping*

Goal 15: *class* Physics (friction)

'Approach' → Plan phase

Define friction in Physics. Without friction, the player continues in horizontal motion indefinitely. Friction is the force that should be applied to the player's horizontal movement to slow them down to a stop. This can be accomplished by multiplying the velocity X by a value smaller than 1. Each time the friction is applied to the velocity X, it reduces it to a smaller value, meaning the player is moving slower than the previous update, until they finally stop.

'Apply' → Do phase

Physics class should apply friction to player when moving horizontally

Step 1: props: `Physics` - add instance variable for friction value

Physics.js → instance variables

```
class Physics {
    /* Physics Properties */
    #speed;
    #gravity;
    #terminal;
    #velocityX;
    #velocityY;
    #friction;
}
```

Step 2: method: `Physics.constructor` - initialize a friction value

Physics.js → constructor

```
constructor(speed) {
    this.#speed = speed;
    this.#gravity = 0.3;
    this.#terminal = 8;
    this.#velocityX = 0.0;
    this.#velocityY = 0.0;
    this.#friction = 0.8;
}
```

Step 3: method: `Physics.applyFriction` - friction reduces the velocity on X=x-axis

Physics.js →applyFriction

```
applyFriction() {
    this.#velocityX *= this.#friction;
}
```

Step 4: method: `Physics.update` - apply friction on each update of game-loop

Physics.js →update

```
update(blocks, player) {
    this.applyGravity();
    this.applyFriction();
    this.checkCollisions(blocks, player);
}
```


By Ted Holmberg *(eholmber@uno.edu)*

'Assess' → Test phase

Test Instructions:

Open `Platformer.html` in Browser

Expected Output:

Friction is applied to the player's horizontal movement, slowing the player to a resting position.

Goal 16: Floor Hazard & Ceiling Hazard

'Approach' → Plan phase

Design the hazards in the game, hazards are a special type of block that can kill the player. They are half sized blocks so their collision detection behavior will be different than a standard block.

'Apply' → Do phase

Inheritance to extend block to have Floor Hazard and Ceiling Hazard

Step 1: class: `FloorHazard` - extends Block, overrides isTouching methods for only bottom collisions

FloorHazard.js

```
class FloorHazard extends Block {
    constructor(x,y) {
        super(x,y, "Assets/tile-spikes-floor.png");
    }

    isTouching( player ) {
        return super.isTouchingX(player, 0.75) && this.isTouchingY(player);
    }

    isTouchingY( player ) {
       const topHalf = this.getY()+this.getHeight()/2 <= (player.getY() + player.getHeight() )
       const bottomHalf = player.getY() <= (this.getY() + this.getHeight() );
       return  topHalf && bottomHalf;
    }
}
```

Step 2: class: `CeilingHazard` - extends Block, overrides isTouching methods for only top collisions

CeilingHazard.js

```
class CeilingHazard extends Block {
    constructor(x,y) {
        super(x,y, "Assets/tile-spikes-ceiling.png");
    }

    isTouching( player  ) {
        return super.isTouchingX(player, 0.75) && this.isTouchingY(player);
    }

    isTouchingY( player ) {
        const topHalf = this.getY() <= (player.getY() + player.getHeight());
        const bottomHalf = player.getY() <= (this.getY() + this.getHeight()/2);
        return  topHalf && bottomHalf;
    }
}
```

Step 3: props: `Scene` - add instance variable for a monsters array

Scene.js → instance variables

```
/* Scene Properties */
#background;
#blocks;
#monsters;
#player;
```


By Ted Holmberg *(eholmber@uno.edu)*

Step 4: method: `Scene.constructor` - initialize monsters to an empty array

Scene.js → constructor

```
constructor(map){
    this.#blocks = [];
    this.#monsters = [];
    this.setScene( map );
}
```

Step 5: method: `Scene.setTile` - on tiles "A" or "V" create hazard instances.

Scene.js → setTile

```
setTile (x, y, tile){
    switch(tile){
         case "#": this.#blocks.push( new Block(x,y) );            break;
         case "@": this.#player = new Player(x,y);                 break;
         case "A": this.#monsters.push( new FloorHazard(x,y) );    break;
         case "V": this.#monsters.push( new CeilingHazard(x,y) );  break;
     }
}
```

Step 6: method: `Scene.draw` - for each hazard in monsters array, invoke its draw method

Scene.js → draw

```
 draw(){
   this.#background.draw();
   this.#blocks.forEach( (block) => block.draw() );
   this.#monsters.forEach( (monster) => monster.draw() );
   this.#player.draw();
 }
```

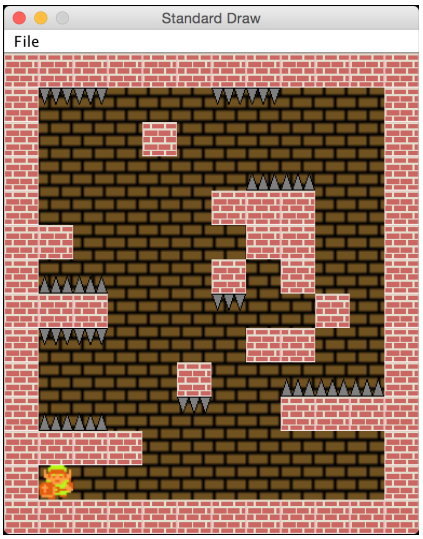
Step 7: update HTML - add FloorHazard.js & CeilingHazard.js as a script element to the body

Platformer.html → <body>

```
<body>
    <canvas id='viewport' width="500" height="500" style='border: 1px solid black'></canvas>
    <script src='Scripts/worldData.js'>  </script>
    <script src='Scripts/World.js'>      </script>
    <script src='Scripts/GameObject.js'> </script>
    <script src='Scripts/Block.js'>      </script>
    <script src='Scripts/FloorHazard.js'>  </script>
    <script src='Scripts/CeilingHazard.js'> </script>
    <script src='Scripts/Physics.js'>    </script>
    <script src='Scripts/Player.js'>     </script>
    <script src='Scripts/Scene.js'>      </script>
    <script src='Scripts/Controller.js'> </script>
    <script src='Scripts/View.js'>       </script>
    <script src='Scripts/Game.js'>       </script>

    <script>
         window.addEventListener('load', Game.main );
    </script>
</body>
```

'Assess' → Test phase



Test Instructions:

Open `Platformer.html` in Browser

Expected Output:

GUI window display with:

- *Background image*
- *Brick blocks*
- *Player character*
- ***Ceiling Hazards***
- ***Floor Hazards***

NOTE: *No collisions with hazards yet*

Goal 17: Game Over (Lose)

'Approach' → Plan phase

Define the Lose condition of the game. When the player touches any hazard tile, then the game should end.

'Apply' → Do phase

For each Game update, the game asks the scene if the player is dead or not, and sets the game over based on that. The scene iterates through each hazard & determines if player is touching, if so then player is dead, if not then player is alive.

Step 1: method: `Scene.hasCollisions` - returns if a player is touching a hazard

Scene.js → hasCollisions

```
hasCollisions(){
   return this.#monsters.some( monster  => monster.isTouching(this.#player) );
 }
```

Step 2: method: `Scene.getCollisions` - returns the hazard that is touching he player

Scene.js → getCollisions

```
getCollisions(){
   return this.#monsters.filter( monster => monster.isTouching(this.#player) )
 }
```

Step 3: method: `Game.update` - set game over value based on whether collisions occurred with hazards.

Game.js → update

```
update() {
    this.#controller.update();
    this.#scene.update();
    this.#isOver = this.#scene.hasCollisions();
}
```

'Assess' → Test phase

Test Instructions:

Open `Platformer.html` in Browser

Expected Output:

If player touches a hazard, the game should stop, i.e. Game Over.

Goal 18: *class* Exit

'Approach' → Plan phase

Exits are a special type of block that advances the player to the next stage. They are doorway blocks so their collision detection behavior will be different than a standard block.

'Apply' → Do phase

Create Exit class by extending Block class and then draw the exit into scene

Step 1: class: `Exit` - extends Block, overrides isTouching methods collision box (i.e. overlap ratio).

Exit.js

```
class Exit extends Block {
    constructor(x, y) {
        super(x,y,"Assets/tile-exit.png");
    }

    isTouching( player ) {
        return super.isTouchingX(player,0.25) && super.isTouchingY(player, 0.25);
    }
}
```

Step 2: props: `Scene` - add instance variable to hold an exit object in Scene

Scene.js → instance variables

```
/* Scene Properties */
#background;
#blocks;
#monsters;
#player;
#exit;
```

Step 3: method: `Scene.setTile` - on "!" then instantiate an exit instance at that coord

Scene.js → setTile

```
setTile (x, y, tile){
    switch(tile){
        case "#": this.#blocks.push( new Block(x,y) );            break;
        case "@": this.#player = new Player(x,y);                 break;
        case "A": this.#monsters.push( new FloorHazard(x,y) );    break;
        case "V": this.#monsters.push( new CeilingHazard(x,y) );  break;
        case "!": this.#exit = new Exit(x,y);                     break;
    }
}
```


By Ted Holmberg *(eholmber@uno.edu)*

Step 4: method: `Scene.draw` - have exit draw in the scene's draw method

Scene.js → draw

```
draw(){
  this.#background.draw();
  this.#blocks.forEach( (block) => block.draw() );
  this.#monsters.forEach( (monster) => monster.draw() );
  this.#exit.draw();
  this.#player.draw();
}
```

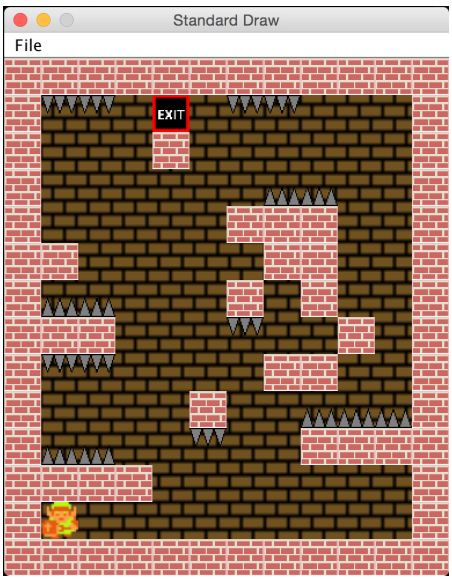
Step 5: update HTML - add Exit.js as a script element to the body

Platformer.html → <body>

```
<body>
    <canvas id='viewport' width="500" height="500" style='border: 1px solid black'></canvas>
    <script src='Scripts/worldData.js'>  </script>
    <script src='Scripts/World.js'>      </script>
    <script src='Scripts/GameObject.js'> </script>
    <script src='Scripts/Block.js'>      </script>
    <script src='Scripts/FloorHazard.js'>  </script>
    <script src='Scripts/CeilingHazard.js'> </script>
    <script src='Scripts/Exit.js'>         </script>
    <script src='Scripts/Physics.js'>    </script>
    <script src='Scripts/Player.js'>     </script>
    <script src='Scripts/Scene.js'>      </script>
    <script src='Scripts/Controller.js'> </script>
    <script src='Scripts/View.js'>       </script>
    <script src='Scripts/Game.js'>       </script>

    <script>
         window.addEventListener('load', Game.main );
    </script>
</body>
```

'Assess' → Test phase



Test Instructions:

Open `Platformer.html` in Browser

Expected Output:

GUI window display with:

- *Background image*
- *Brick blocks*
- *Player character*
- *Ceiling Hazards*
- *Floor Hazards*
- ***Exit***

NOTE: *No collisions with Exit yet*

Goal 19: GameOver (Win)

'Approach' → Plan phase

Define the Win condition. When a player touches the exit tile advance to the next scene. The player wins if they exit the last level.

'Apply' → Do phase

Check if the player is touching exit during each update. If so, increment level, and check to see if it's the last level. If it isn't then update the scene to that level by getting the level map from world and constructing a new scene in the game, then register the new player object to the controller. If it is the last level, then the game is over.

Step 1: method: `Scene.getExit` - provides access to scene's exit from other classes

Scene.js → getExit

```
getExit() {
    return this.#exit;
}
```

Step 2: method: `World.getLength` - provides access to world's levels array length from other classes

World.js → getLength

```
getLength(){
    return this.#levels.length;
}
```

Step 3: method: `Game.loadScene` - loads a new scene within the game & binds player to new controller

Game.js → loadScene

```
loadScene(){
    const map = this.#world.getLevel(this.#level);
    this.#scene = new Scene(map);
    this.#controller = new Controller( this.#scene.getPlayer() );
}
```

Step 4: method: `Game.update` - if player touches exit then the game loads next level until none are left

Game.js → update

```
update() {
    this.#controller.update();
    this.#scene.update();
    if ( this.#scene.getExit().isTouching( this.#scene.getPlayer() ) ) {
        this.#level++;
        if (this.#level < this.#world.getLength() ){
            this.loadScene();
        }
        else{
            this.#isOver = true;
        }
    }
    if ( this.#scene.hasCollisions() ){
        this.loadScene()
    }
}
```

'Assess' → Test phase

Test Instructions:

Open `Platformer.html` in Browser

Expected Output:

When the player touches exit, the game should load the next level. If last level, then game over.

Goal 20: Sprite Animation

'Approach' → Plan phase

Design an approach that can animate the player by switching the images in a certain order after a certain duration. Then organize all the animated poses in a way to easily access them.

'Apply' → Do phase

Animation class manages the collection of images for an animation, which is the current image, and how to switch to the next one. The Pose enum, defines all the character animations. Player has a current pose which is updated whenever the player moves left/right. Override the draw method in player to update the image before drawing.

Step 1: class: `Animation` - models an Animation sequence and determines which frame to return

Animaton.js

```
class Animation{
    /* Animation Properties */
    #startTime;
    #frameIndex;
    #sequence;

    constructor(images){
        this.#startTime = Date.now();
        this.#frameIndex = 0;
        this.#sequence = images.map( e => Object.assign(new Image(), {src:e}) );
    }

    getImage(){
        const index = this.#frameIndex;
        const now = Date.now();
        if(now - this.#startTime > 75){
            this.#frameIndex = (index + 1) % this.#sequence.length;
            this.#startTime = now;
        }
        return this.#sequence[index];
    }
}
```

Step 2: class: `Pose` - models the player's animated poses: left & right

Pose.js

```
class Pose{
    static RIGHT = new Pose("Assets/link-right.png", "Assets/link-right2.png");
    static LEFT = new Pose("Assets/link-left.png", "Assets/link-left2.png");

    constructor(...images){
        this.animation = new Animation(images);
    }

    getImage(){
        return this.animation.getImage();
    }

}
```

Step 3: method: `GameObject.setImage` - allows other classes to set game object's image

GameObject.js → setImage

```
setImage(img){
    this.#image = img;
}
```

Step 4: props: `Player` - add a instance variable for a currentPose to Player

Player.js → instance variables

```
/* Player Properties */
#physics;
#isJumping;
#currentPose;
```

Step 5: method: `Player.constructor` - initialize current pose & set image with it.

Player.js → constructor

```
constructor(x,y) {
    super( x*Block.SIZE, y*Block.SIZE, Block.SIZE, Block.SIZE, "Assets/link-down.png");
    this.#physics = new Physics(4);
    this.#isJumping = false;
    this.#currentPose = Pose.RIGHT;                                    //ANIMATION POSES
    super.setImage( this.#currentPose.getImage() );
}
```

Step 6: method: `Player.moveLeft` - set current pose to left pose

Player.java → moveLeft() method

```
moveLeft() {
    this.#physics.moveLeft();
    this.#currentPose = Pose.LEFT;
}
```

Step 7: method: `Player.moveRight` - set current pose to right pose

Player.java → moveRight() method

```
moveRight() {
    this.#physics.moveRight();
    this.#currentPose = Pose.RIGHT;
}
```

Step 7: method: `Player.draw` - override draw: set image with current pose animation before drawing

Player.java → draw() method

```
draw() {
    super.setImage( this.#currentPose.getImage() );
    super.draw();
}
```

Step 8: update HTML - add Animation.js & Pose.js as a script element to the body

Platformer.html → <body>

```
<body>
    <canvas id='viewport' width="500" height="500" style='border: 1px solid black'></canvas>
    <script src='Scripts/worldData.js'>  </script>
    <script src='Scripts/World.js'>      </script>
    <script src='Scripts/GameObject.js'> </script>
    <script src='Scripts/Animation.js'> </script>
    <script src='Scripts/Pose.js'> </script>
    <script src='Scripts/Block.js'>      </script>
    <script src='Scripts/FloorHazard.js'>  </script>
    <script src='Scripts/CeilingHazard.js'> </script>
    <script src='Scripts/Exit.js'>         </script>
    <script src='Scripts/Physics.js'>    </script>
    <script src='Scripts/Player.js'>     </script>
    <script src='Scripts/Scene.js'>      </script>
    <script src='Scripts/Controller.js'> </script>
    <script src='Scripts/View.js'>       </script>
    <script src='Scripts/Game.js'>       </script>

    <script>
         window.addEventListener('load', Game.main );
    </script>
</body>
```

'Assess' → Test phase

		Right animation, consists of two frames
		Left animation, consist of two frames

Conclusions

Final Comments

In this lab you learned to implement a Browser App using an Object-Oriented Principles using JavaScript.

Future Improvements

- Refactor this into an Object-Oriented Drawing app
- Reduce the number of steps to ten
- Maximize calls for displaying to canvas & capturing input events

Lab Submission

Compress your project folder into a zip file and submit on Moodle.

Homework 4:

Create your own front-end browser app with JavaScript & Canvas API / Phaser.

Homework Bonus:

Showcase bonus. You can receive up to 20 bonus points if your project is outstanding and novel.
I'll publish all showcase projects on UNO's web page as a demo for future students. You should cite such projects on your resume.