Mobile Games Development 1 Coursework: Code Explanation

*I confirm that the code contained in this file (other than that provided or authorised) is all my own work and has not been submitted elsewhere in fulfilment of this or any other award*.

Fotios Spinos

Contents

[1.0 Introduction 3](#_Toc26986475)

[2.0 HTML 3](#_Toc26986476)

[3.0 CSS 3](#_Toc26986477)

[4.0 JavaScript 3](#_Toc26986478)

[5.0 GitHub Link 3](#_Toc26986479)

[6.0 Engine Related Scripts 3](#_Toc26986480)

[6.1 Input 3](#_Toc26986481)

[6.1.1 Class Variables 4](#_Toc26986482)

[6.1.2 Methods 4](#_Toc26986483)

[6.1.3 Event Handlers 5](#_Toc26986484)

[5.3 Axis 5](#_Toc26986485)

[5.3.1 Class Variables 5](#_Toc26986486)

[5.2.2 Methods 6](#_Toc26986487)

[5.4 Component 6](#_Toc26986488)

[5.6 CollisionChecker 7](#_Toc26986489)

[5.7 Collider 8](#_Toc26986490)

[5.8 CircleCollider 9](#_Toc26986491)

[Class Variables 9](#_Toc26986492)

[5.9 SquareCollider 9](#_Toc26986493)

[5.10 SquareShape 10](#_Toc26986494)

[5.11 CircleShape 10](#_Toc26986495)

[5.12 Vector2D 11](#_Toc26986496)

[5.13 EngineImage 12](#_Toc26986497)

[5.14 Canvas 13](#_Toc26986498)

[5.15 GameObject 14](#_Toc26986499)

[5.16 Scene 16](#_Toc26986500)

[5.17 SceneManager 17](#_Toc26986501)

[static instance 17](#_Toc26986502)

[5.18 SpriteAnimation 18](#_Toc26986503)

[Relevant Global Variables 18](#_Toc26986504)

[5.19 AnimationController 21](#_Toc26986505)

[5.20 EngineAudio 22](#_Toc26986506)

[5.21 AudioManager 22](#_Toc26986507)

[5.22 Window 23](#_Toc26986508)

[Class Variables 23](#_Toc26986509)

[5.23 EngineText 24](#_Toc26986510)

[5.24 Engine 24](#_Toc26986511)

# Introduction

The goal of this project is to develop a game using JavaScript. Since this is the author’s first project using the language, it was decided that existing libraries will not be used. Instead the implementation of a component-based game engine would give a better understanding on how game technology is build using the language. With that being said, it is desirable to build a relatively large scoped and stable tool which covers the basic game development needs.

# HTML

The HTML5 consists of the head and body tags. One of the project goals was to keep the main functionality of the project in JavaScript(js) components. This allows the creation of HTML tags in real time. The head of the document makes use of the title tag to provide a title for the window. The Body on the other hand is used to link to and connect the js scripts. To link these scripts the document makes use of the script tag. By assigning the src element the software can refer to external scripts. This improves the software’s readability as the scripts are not written on the same document. This approach might be appropriate for some narrow-scoped projects but since one of the goals of the project is to build a game engine, it would be more appropriate to use classes and separate script functionality as much as possible.

# CSS

While CSS provides many formatting capabilities for HTML elements, the software will not be using a CSS files to bind formatting attributes to tag elements. It is preferable to provide these formatting capabilities to the engine components to avoid undesired behaviors arising from tags sharing the same type or class name.

# JavaScript

The core functionality of engine components and game related scripts is written in JavaScript. A detailed explanation of each script will take place in the following sections of the document.

# GitHub Link

<https://github.com/FotisSpinos/Component-Based-Game-Engine-In-Javascript.git>

# Engine Related Scripts

# Input

A class exposing a static instance of itself (Singleton). By using JavaScript event handlers, the class can store input information recorded in real-time by the user.

# Class Variables

**keys**

An array of Booleans indexed with the given JavaScript keycodes. The keycodes are numbers and elements in the array can be accessed likewise:

Input.instance.keys[70]

By referring to the char table provided by Cambia Research (CambiaResearch, 2017) we can identify that the number 70 matches the ‘F’ key on the keyboard. When a key in the collection returns true, the software assumes that the button is currently pressed.

**cursorDown**

A Boolean used to identify if the left cursor button is currently pressed.

**onMouseDownPos**

Stores a Vector2D representing the window position of the cursor when the left button is pressed.

**onMouseUpPos**

Stores a Vector2D representing the window position of the cursor when the left button is released.

**instance**

Holds a static instance of the Input object.

**axis**

A collection of axis objects.

# Methods

**constructor**

The constructor is responsible for variable initialization. In addition, the function assigns events to their corresponding method references. This is possible by calling the addEventListener method. Finally, the constructor makes sure that the static instance of the object is not reassigned if the variable has not been already assigned which is the case for all singleton classes.

**getInstance**

Returns a static instance of the Input class. While the instance variable is public, the use of getInstance ensures that the method’s return value is not equal to null as the method creates an and assigns the variable if it was unsigned.

**resetCursorInputs**

Resets all cursor related variables: onMouseDownPos, onMouseUpPos and cursorDown. Variables storing mouse positions are set to (-1, -1) to ensure that UI elements cannot use these values since window coordinates start from (0, 0).

**updateAxis**

Updates all the assigned axis

**keyDown**

Called when a key is pressed. The event parameter holds the keycode of the pressed key. The corresponding element of the “key” array is set to true.

**keyUp**

Called when a key is released. The functionality is similar to keyDown. The corresponding element of the “key” array is set to false.

**mouseDown**

Records cursor position when the left cursor button is pressed.

**mouseUp**

Records cursor position when the left cursor button is released.

# Event Handlers

All class variables are defined dynamically using the appropriate event handlers. The class uses the following events: ‘keydown’, ‘keyup’, ‘mousedown’ and ‘mouseup’.

# Axis

Allows the developer to create virtual axis. Recognized game engines like Unreal Engine 4 and Unity3D provide much more functionality but considering the scope of this project this implementation of virtual axis provides a simple and appropriate use in an attempt to produce smooth motion. The value of a virtual axis varies between a set range. Depending on user input, the axis value tends to come closer to the max or min boundary. To ensure that the value varies between these ranges, the class uses two factors varying between 0 and 1. Each factor increases when a defined key is pressed and decreases on release.

# Class Variables

**id**

The virtual axis’s name.

**rate**

A number representing the rate in which the axis increases or decreases.

**delay**

A number representing the remaining amount of time before the axis’s value is updated.

**positiveKeyID**

The keycode that increases the axis value on press.

**negativeKeyID**

The keycode that decreases the axis value on press.

**increaseFactor**

A value between 0 and 1 increased by a rate when the positive key is pressed.

**decreaseFactor**

A value between 0 and -1 increased by a rate when the negative key is pressed.

# 5.2.2 Methods

**Constructor()**

Assigns object variables and passes a reference of itself to the axes of the input class.

**axisValue()**

A getter returning the difference between the positive and the negative factor. The value returned represents the value of the virtual axis.

**culculateFactor(factor, key)**

Returns a factor value using a reference of the factor and keycode increasing the factor. The function consists of a collection of checks returning the appropriate factor value. The function increases the axis value when the key passed as a parameter is pressed. When the key is not pressed the factor reduces the axis value by the predefined rate. The function makes sure that the factor ranges between 0 and 1.

**update**

Updates the two factors when the difference between the current time and the previous update is greater than the specified rate.

# Component

The component provides a set of abstract methods. Classes inheriting from the component can choose to override the provided methods and provide their own functionality to the game object.

* + 1. Class Variables

**gameObject**

A reference to the attached gameObject. Components should be able to access the attached or other gameObjects and gameObject components found in a scene to change the game state.

* + 1. Methods

**constructor()**

initializes the attached game object.

**start()**

Executes when a component is attached to a gameObject. Should be used to initialize the component as the gameObject reference is being set.

**update()**

Executes on each frame update. Is mainly used to describe the behavior of the component altering the game state.

**Render()**

Executes on each frame update. Provides logic to render the component on the canvas attached to the game object.

**onCollisionEnter(collisionObj)**

Executes when a collision occurs. The parameter collisionObj provides collision information like the collided object. The developer can use this function to provide collision-based behaviors between objects.

**onSceneLoad()**

Executes when a scene loads. Is mostly used to initialize scenes.

**onSceneExit()**

Executes when a new scene loads. Is mostly used to clean up scenes.

* 1. Transform

A component holding the game object’s position and scale.

* + 1. Class Variables

**position**

A Vector2D storing the position of the attached gameObject.

**scale**

A vector storing the scale of the attached gameObject.

* + 1. Methods

**constructor(pos, scale)**

Defines the two class variables using the parameters passed.

**Print()**

Prints the object’s state, mostly used for debugging purposes.

# CollisionChecker

A class providing static functions used to identify collisions in the game environment.

* + 1. Methods

**static circleToCircle(circle1, circle2)**

returns true if a collision between two circle shaped colliders has occurred. Since every point of a circle has the same distance to its central point, a collision between two circles can be identified by calculating the distance between the two center points and comparing it with the addition of the two circle radiuses. A collision has occurred if the distance magnitude is less than the addition of radius of each circle.

**static rectToRect(rect1, rect2)**

Returns true if a collision occurred between two rectangles. The collision algorithm used to identify a collision between two rectangles can be found at the MDN web docs website. According to this source: “two rectangular colliders intersect if there is no gap between any of the 4 sides of the rectangles”(MDN contributors, 2019).

**static defineCollisionType(collisionObj)**

This method is used to identify the type of a collider. To check the type of a collider, the function uses the constructor attribute of the collider object. A switch statement checks through each collider type. If the collision object matches the supported collision types, the function returns the matching type.

**static checkCollision** **(collider1, collider2)**

checks if a collision occurs between two components inheriting from the collision class. The function starts of by identifying the collider types. Depending on the two types found, the function makes the appropriate function call to identify if a collision has occurred.

# Collider

A collider is a component providing its own set of abstract methods. Classes inheriting from this class should override these functions to introduce their own functionality into the engine.

* + 1. Class Variables

**offset**

The offset to the gameObject’s position.

**active**

A Boolean defining if the collider is active. This allows the developer to activate or deactivate colliders during runtime. Collision checks on the collider object occur when the variable is set to true.

**pos**

Represents the position of the collider.

* + 1. Methods

**updatePos()**

Updates the collider’s position. This process is essential as the offset and therefore the collider’s position can change during runtime. This function ensures that the object has up to date attributes when the game updates.

* + 1. Abstract functions

The functionality for the following abstract functions is identical to the Component functions described previously in the document.

**start()**

**render()**

**update()**

# CircleCollider

A class representing circle shaped colliders.

# Class Variables

**offset**

Inherited from the Collider class

**radius**

A number representing the radius of the circle

**pos**

A 2D vector representing the position of the circle

* + 1. Methods

**constructor(offset, radious)**

sets class attributes

**onSceneLoad**

Makes sure that the object’s position has been updated when a scene loads

**update()**

Updates the object’s position and checks for collisions with other canvas objects.

# SquareCollider

A class representing square shaped colliders. The provided by this class is very similar to Circle Collider

* + 1. Class Variables

**Size**

A 2D vector representing the size of the square’s width and height

**pos**

* + 1. Methods

**constructor(offset, size)**

**onSceneLoad()**

**update()**

# SquareShape

A component used to render a rectangle on the game canvas.

* + 1. Class Variables

**Color**

Holds the color of the square

* + 1. Methods

**Constructor()**

**render()**

Renders the square using the gameObject’s canvas context and transform attributes. The function consists of setting the context color and calling the fillRect function to draw the square with the defined color.

# CircleShape

A component used to render a circle on the game canvas.

* + 1. Class Variables

**startAng**

A number representing the start angle of the circle. This variable is set by default to 0

**endAng**

A number representing the end angle of the circle. This variable is set by default to 360 so that a full circle can be drawn.

**Radious**

A number representing the radius of the circle.

**Color**

Stores the color of the circle.

* + 1. Methods

**constructor()**

Sets relevant object attributes. Since the purpose of the component is to draw a circle the start angle should be set to 0 and endAngle to 360 degrees.

**render()**

The render function consists of three function calls. To draw a circle the program first needs to begin a path by calling beginPath(). To define the circle path the function uses the arc function with the object’s attributes as parameters. Finally, the method calls the stroke function to draw the path.

# Vector2D

A vector has a direction and a scale. Vector calculations can represent speed, positioning and other game related information. Javascript does not support operation overloading which makes the use of functions necessary for vector calculations. Methods representing vector operations return a vector instance so that the vector’s functionality can be reused by the produced vector.

The goal of the Vector2D class is to allow code with the following capabilities:

let vec1 = new Vector2D(1,1);

let vec2 = new Vector2D(3, 3);

let testVector = vec1.AddVec(vec2).scaleVec(2);

testVector.print();

This code translates to ((1,1) + (3,3)) \* 2

The expected outcome is (8, 8)

* + 1. Class Variables

**x**

A number representing the x element of a 2D vector

**y**

A number representing the y element of a 2D vector

* + 1. Methods

constructor(x, y)

Sets the x and y attributes of the vector

**scaleVec(scalar)**

returns a new vector instance produced by the multiplication of the vector object with a scalar passed as parameter.

**multVec(vec)**

Returns a new vector instance produced by the multiplication of the vector object with the vector parameter.

**minVec(vec)**

Returns a new vector instance produced by the subtraction of the vector object with the vector parameter.

**addVec(vec)**

Returns a new vector instance produced by the subtraction of the vector object with the vector parameter.

**static get zero()**

returns a (0, 0) vector.

**magnitude()**

measures and returns the scale of the current vector using the Pythagoreans theorem.

**normalize()**

Returns a new vector instance by subtracting the vector object with its magnitude.

# EngineImage

A component used to render and store image data.

* + 1. Class Variables

**imgSource**

A string representing the source of the image. The source can be a url or a local path to an image.

**Id**

A string representing the name of the image

**Img**

The HTML image element

* + 1. Methods

**constructor(imgSource, id)**

Sets object attributes

**start()**

Creates an HTML image elements

**render()**

Renders an image by calling the canvases context method drawImage.

# Canvas

A class used to manage canvas objects and provide canvas functionality.

* + 1. Class Variables

**C**

Holds a canvas reference.

**canDraw**

A Boolean allowing canvas rendering when set to true.

**canUpdate**

A Boolean allowing canvas updating when set to true.

**drawObjs**

An array of all the objects rendered on the canvas

**Id**

A string representing the canvas name

* + 1. Methods

**setSize(size)**

Sets the size canvas

**createCanvas(id, size)**

Creates a div canvas container, a canvas element and defines appropriate styling. To achieve this the program makes use of the web API. To create a canvas the function calls the create element function accessed from the document object. The style of the canvas is set in this method by setting the style attributes. The width and height are set to 100% meaning that the canvases size will fully cover the container. Using the style of the container, the program sets the div element at the bottom right of the screen. Once the canvas width and height are being corrected from the set aspect ratio, the container will be positioned at the center of the screen.

**constructor(id, size, drawObjs)**

Makes appropriate function calls and sets object attributes to create a canvas element.

**addDrawObj(drawObjs)**

Adds a GameObject to the drawObjs array.

**removeDrawObj(drawObj)**

Removes a GameObject from the drawObjs array.

**clearCanvas()**

Clears canvas to ensure that the updated draw objects are visible.

**update()**

Updates every canvas GameObject. Makes use of a for loop to iterate through each element of the drawObjs.

**render()**

Renders every canvas GameObjects

**onSceneLoad()**

Calls the on-scene load functionality of each gameObject.

**onSceneExit()**

Calls the on-scene exit functionality of each gameObject.

# GameObject

A container used to store and handle component objects.

* + 1. Class Variables

**Components**

An array of components handled by the game object.

**transform**

The transform component of the GameObject. All game objects should have a transformcomponent by default since every game object should have a position and a scale.

**id**

A string storing the game object id.

**tag**

A string storing the game object tag. Tags can be used to categorize a set of game objects.

**canvas**

A reference to a canvas

* + 1. Methods

**constructor(name, pos, scale)**

Initializes and sets attributes using parameters. The name parameter is a string and is assigned to the gameObject’s id. The position and scale parameters are vector2D variables representing the position and scale of the game object respectively.

**set setCanvas(canvasObj)**

Sets the canvas holding the game object

**addComponent(addedComp)**

Adds a component to the components array.

**removeComponent(type)**

Adds a component from the components array.

**getComponent(type)**

Returns a parameter by type. The program iterates through every element in the components array using a for loop. The function stores the type of the component using the object’s prototype. A while loop iterates through the hierarch of the component until the component is found or the local type variable is equal to null which indicates that the object was not found. On each iteration the function moves through the hierarchy using the prototype variable as indicated above. This allows the developer to receive components derived from the type passed as a parameter. For example, a game object with a boxCollider can be accessed by passing a Collider to this function:

gameObject.getComponent(Collider)

The function will return a reference to the box collider component as the class inherits from the collider class.

**start()**

**onCollisionEnter(collision)**

Calls the onCollisionEnter function of every component attached to this game object.

**updateComponents()**

Calls the update function of every component attached to this game object.

**renderComponents()**

Calls the render function of every component attached to this game object.

**onSceneLoad()**

Calls the onSceneLoad function of every component attached to this game object.

**onSceneExit()**

Calls the onSceneExit function of every component attached to this game object.

**static find(id)**

A static function used to return a game object with a matching id. The id is passed as a parameter. The function iterates through every game object in every canvas of the current running scene using a nested for loop to identify the first game object matching the specified id.

**static findObjsWithTag(tag)**

A static function used to return an array of game objects with a matching tag. The tag is passed as a parameter. The method uses a nested for loop to iterate through every game object in every canvas of the running scene to identify game object with a matching tag.

# Scene

The architecture of the engine is based on the following principle. A scene contains a collection of Canvases and a Canvas contains a collection of GameObjects. The class provides functionality to manage canvases contained in the scene.

* + 1. Class Variables

**Name**

A string containing the scene name.

**Canvases**

A collection of canvases managed by the scene.

**Index**

A number containing the index of the scene.

**static sceneNumber**

Represents the current number of scenes.

* + 1. Methods

**constructor(id, canvases)**

Sets object attributes.

**removeCanvas** **(id)**

Removes a canvas from the canvases array. The functionality is similar to other remove functions mentioned earlier in the document.

**addCanvas** **(canvasObj)**

The function adds the canvasObj which is a canvas object to the canvases array.

**getCanvas** **(id)**

Returns the first found canvas with a matching id.

**clearCanvaces()**

Clears all scene canvases.

**update()**

Updates all scene canvases.

**render()**

Renders all scene canvases.

**onload()**

Calls the onload function of all scene canvases.

**onExit()**

Calls the onExit function of all scene canvases.

**collisionCheck()**

Calls the collision check function of all scene canvases.

# SceneManager

A singleton class managing all loaded scenes.

* + 1. Class Variables

# static instance

A static instance of to the SceneManager object.

**static sceneNumber**

The current number of scenes

**scenes**

A collection of scene objects

**runningScene**

An instance to the running scene.

* + 1. Methods

**getInstance()**

Returns the static instance of SceneManager.

**addScene(scene)**

Adds a scene object to the scenes array.

**loadScene(sceneID)**

loads a scene matching the id passed as a parameter.

# SpriteAnimation

A sprite animation consists of an engine image. The illusion of motion is achieved by displaying parts of the image in a believable manner. The images used for this purpose are PNG sprite sheets. Often, sprite sheets do not follow the same format. Animations are often displayed horizontally or vertically but certain instabilities produce the need to build a tool capable of reading several of formats.

It might also be desirable to combine sprite animations and define special rules for looping them. For instance, the player attack animation reads the image horizontally until its end and continues from the opposite direction towards the start. This splits the sprite animation into two parts. At the first half of the animation the mage points its staff forward. At the second half the mage returns the staff towards himself. This as the animation is produced by displaying the same parts of the sprite sheet. The second time the animation is played the same process is being carried out.

Another way of looping an animation it to read parts of the sprite sheet until its end and return back to the initial position. This approach repeats the animation and is mostly used in background or other character related motion sprites.

# Relevant Global Variables

* + 1. Class Variables

**spritePos**

Represents the start position in a sprite sheet.

**spriteScale**

The Vector2D size of the rendered subpart on the engine image. For example a sprite animation with a sprite position of (10, 10) and a size of (5, 5) produce a square with the following edges: (10, 10), (15, 10), (10, 15), (15, 15). The software makes use of the canvas API to render that square on the image.

**blankSpace**

The vector2D size between sprites on the image

**animState**

The state of the animation represented by a string. The two possible values are “played” or “paused”.

**posOffset**

A Vector2D offset applied to the position when the sprite renders.

**Name**

A string representing the name of the sprite.

**ySteps**

A number added to the y axis of the position vector. It is used to move the rendered square vertically in the image.

**xSteps**

An array of numbers where each element represents the number of times the rendered area will be shifted horizontally

**xStepsIndex**

An index to the xSteps array

**currentStepsX**

The current number of horizontal transformations

**currentStepsY**

The current number of vertical transformations

**updateTimer**

A timer representing the amount of time remaining before the next sprite update.

**playForwardAnim**

A Boolean suggesting that the animation is played from start to end when set to true.

**animType**

A string defining the type of the animation. Can be: 'reversable' or 'repeatable'.

**spriteScaleStore**

A copy of the initial sprite scale value.

**offsetStore**

A copy of the initial offset value.

**updateTimerStore**

A copy of the update timer value.

**spritePosStore**

A copy of the initial sprite position value.

**spriteImg;**

The sprite sheet image.

* + 1. Methods

**Constructor()**

**initImg(scale)**

Creates an image object using the engine image data.

**reset()**

Resets the animation progress. Once this method is called the animation should play from the set start defined by the constructor parameters.

**update()**

The update function is responsible for progressing the animation. The method checks the animation state and update timer to define if a progression should occur. Image sections rendered undergo a horizontal transformation until the current x step reaches the value defined in the xStep array. For example, when an xSteps array = [2, 3] suggest that 2 horizontal transformation will take place, followed by a vertical transformation, and finally 3 horizontal transformations before the animation loops. By transformation we suggest that square on the image being rendered is moved horizontally. A transformation is defined as follows:

Where axis is the x or y component of the used vectors to represent a horizontal or vertical transformation.

Once the horizontal and vertical steps have been completed, the function reverses or resets the animation according to the data passed to the object.

**render(objPos, objScale, ctx)**

Receives a vector representing the position, a vector representing the game object scale and the context of the canvas. To render a square on the image the function uses an overload of the draw image function used previously in the program. This overload adds 4 parameters representing the position and size of the square rendered. The following code was used to render the sprite animation:

ctx.drawImage(this.spriteImg,

this.spritePos.x, this.spritePos.y,

this.spriteScale.x, this.spriteScale.y,

objPos.x, objPos.y,

objScale.x, objScale.y);

**pause()**

Pauses the animation.

**play()**

plays the animation.

**isPlaying()**

returns true if the animation can be updated

**destinationX(index)**

The x axis destination of the image section being rendered once all the steps have been completed on the given index of the array.

**destinationY()**

The x axis destination of the image section being rendered once all the y steps have been completed.

**reverse()**

Reverses the flow of the animation. To achieve that the function changes the sign of blank space and sprite scale. It also changes the initial position to the destination position.

# AnimationController

A component ensuring that one animation is played at a time along with providing functionality for playing, pausing and retrieving sprite animations.

* + 1. Class Variables

**spriteAnimations**

An array of sprite animation handled by the animation controller

**activeAnimation**

A reference to the playing animation

* + 1. Methods

**Start()**

**addSpriteAnimation(spriteAnimation)**

Adds a sprite animation reference to the spriteAnimations array.

**pauseAnimation(animName)**

Removes the first sprite animation with the matching animation name passed as a parameter

**removeSpriteAnimation(animName)**

Removes the first sprite animation with the matching animation name passed as a parameter

**playAnimation(animName)**

**render**

Most render functions provided by the canvas API require a position and a size. As seen in previous section of the document, the matching data can be retrieved from the gameObject’s transform component. Since a sprite animation is not a component, the object cannot access the game object playing the animation. With that being said the animation controller has a reference to the game object and passes all the appropriate data to the sprite animation.

**update**

Updates the active animation.

# EngineAudio

The EngineAudio stores audio related data.

* + 1. Class Variables

**Audio**

The Audio object provides the functionality of the audio API.

**audioName**

A string representing the name of the audio.

**Looped**

A Boolean suggesting that the audio should be looped when equal to true.

* + 1. Methods

**constructor(audioName, audioSrc, looped)**

Initializes audio data using the given parameters

# AudioManager

A singleton class managing audio objects. It is responsible for all game sounds providing an abstraction to the Audio API. In a resent update Google Chrome introduced a new policy suggesting that the user has to interact with the window before audio can be played. If the application attempts to play an audio clip before the user interacts with the game, then an error will occur. To solve this problem the audio manager is disabled until the player clicks on the canvas. Before that the object does not allow the developer to play audio clips.

* + 1. Class Variables

**Instance**

A static instance of the audio manager.

**enabled**

When set to true audio clips can be played.

**engineAudios**

An array of audio objects managed by the audio manager.

* + 1. Methods

**Constructor**

Initializes class variables and assigns a click event to set the enabled variable to true when the user clicks on the canvas.

**addAudio(engineAudio)**

**playAudio(audioName)**

**stopAudio(audioName)**

# Window

A singleton class using a defined aspect ratio to resize canvas elements within a div container when the window resizes. The algorithm used to implement aspect ratio and resize the game area appropriately can be found in the case study: “Case Study: Auto-Resizing HTML5 Games” conducted by Gopherwood Studios (Detweiler D. 2011).

# Class Variables

* + 1. Methods

**constructor()**

Assigns an event listener calling resizeCanvaces() when the window is being resized.

**resizeCanvaces()**

The method responsible for resizing the container holding the canvas objects. The function uses document.querySelectorAll to receive an instance to the container holding the canvases of the current running scene. To apply the current aspect ratio the function needs to consider the current non updated ratio. To record window dimensions the function uses window.innerWidth and window.innerHeight provided from the web API.

Since both ratios are known, the software compares them to identify which one is larger. When the newWidthToHeight variable (containing the actual window ratio) is larger, the new width of the container is defined by:

newWidth = actual window width \* actual window height;

Considering that the aspect ratio is defined by a fixed width and height:

Aspect ratio: width / height

The function recalculates the width so that the two ratios are equal. With that being said, the height of the container is being recalculated when the actual aspect ratio is less than the desired aspect ratio.

# EngineText

Component providing functionality to display text on the canvas.

* + 1. Class Variables

**Text**

A string representing the text displayed on the canvas

**Font**

A string representing the font of the text

**Color**

A string representing the text color

* + 1. Methods

**Constructor**

Initializes default class variables. The default font is set to '30px Comic Sans MS' and the color to ‘red’. Both values are valid and can be used for the upcoming function calls provided by the canvas API.

**Render**

Uses the canvas API to render the specified text using the fillText function. The font and color of the text are set by assigning the context’s font and fillStyle members using the class variables “font” and “color” respectively.

# Engine

Initializes all engine managers and provides a game loop making the necessary functions responsible for updating the state of the program, rendering the canvas, and handling input.

* + 1. Class Variables

**running**

when set to false the application terminates.

**inpt**

A reference to the Input instance.

**sceneManager**

A reference to the scene manager instance.

**audioManager**

A reference to the audio manager instance.

**Window**

A reference to the window instance

**previousScene**

A reference to the previously running scene handled by the engine.

**lastFrameTime**

Records the total time the application runs every time the game loop executes.

**deltaTime**

The amount of time required to update the game state

* + 1. Methods

**Constructor()**

Initializes all engine related objects.

**getInstance()**

Returns the engine instance.

**run(timestamp)**

This method makes all the appropriate function calls to update the game state.

To calculate the delta time, the function subtracts the current total time the application runs with the last recorded frame time. The subtraction provides the difference or in other words the time it took for the game state to update. To record the current time the function uses Time.now() which returns the time in milliseconds. By dividing this value by a thousand we can turn the value to seconds.

The function checks if this is the first time a scene loads or if a new scene is being loaded (A transition occurred between scenes). If one of these cases is true, the method calls the on-load function of the scene so that every related object applies its functionality when a scene is being loaded. Accordingly, the previous scene calls all the appropriate on exit functions from all related game objects and clears all canvases before the new scene is loaded. The game master also searches and records objects matching its class variables. This is appropriate as there is no need for the game master to keep searching matching game objects on the current scene under the assumption that the objects searched are not being created after the scene is being initialized.

The game loop proceeds by clearing the canvas, updating axis objects, rendering and updating scene canvases and resetting cursor inputs. The previous scene is also set to hold a reference to the current scene so that the function can identify if a new scene is being loaded.

The function iterates using requestAnimationFrame which ensures that a function is called when the browser is ready to render the window. Javascript is single threaded and so a repeated function calls would result in the application crashing as calls would be continually stacking on top of each other.

**update()**

Checks if the reference to the two class variables has been assigned. If they have not, the function conducts a search to the scene and assigns the variables if the objects are found.

**static getInstance()**

**get score()**

**static createEnemy(id, position, canvas)**

Creates an enemy game object

**static createPlayer(id, position, canvas)**

Creates a player game object

1. Game Related Scripts

The upcoming sections provide an explanation of the scripts responsible for the game logic.

* 1. GameMaster

The game master stores a player and score text reference. These two variables are used on the scene initialization process or through multiple scenes. It is possible to search on the appropriate scenes to receive a reference but admittedly this can be difficult since the scene manager provides a reference to the running scene. The software would have to rely on hardcoding the scene index the objects belong to and search through all the draw objects until the two references are found. This solution is not expandable and degrades the readability of the script. The script also provides functionality for building certain game objects. It is worth mentioning that the script could be expanded further to support other object types to simplify other scripts and possibly be used to load scenes from JSON files.

* + 1. Class Variables

**scoreValueTextComp**

Holds a reference to the score value text component. The rederence is obtained with the following code:

//Obtains a game object reference

let scoreValueText = GameObject.find("scoreValueText");

//Retrieves the component of type passed as a parameter

this.scoreValueTextComp = scoreValueText.getComponent(EngineText);

**player**

A reference to the player GameObject. The GameObject is obtained with the following function call: GameObject.find("player");

* 1. PlayerScript

Contains all player character functionality.

* + 1. Class Variables

**displacement**

A number representing the player disclaimant on the x axis.

**charImageAnim**

The engine image containing the PNG containing player animations.

**charImageAnimFlipped**

A flipped version of the image contained in the charImageAnim.

**deathSprite**

The death sprite animation.

**walkForwardSprite**

The sprite played when the player character moves forward.

**attackSprite**

The sprite played when the player character attacks

**playerAC**

The animation controller handling all the player character animation.

**lastWalkAnim**

The sprite animation played before a player action interrupted the action.

**currentState**

The state of the player represented by a string

**attackDuration**

The time duration the attack animation is being played

**attackDurationStore**

Stores the value of the attackDuration variable.

**attackRate**

The amount of time required before the player can attack consecutively.

**attackRateStore**

A copy of the attackRate value.

**Collision**

An object holding collision information. The variable is being updated on each frame.

**Health**

A number representing the player’s health

* 1. Reverse sprite animations

Each one of these sprites contains a reverse version used when the player moves backwards.

* + 1. Methods

**start()**

**Initializes animation variables**

**createFireBall()**

Creates a fireball by creating a game object Infront of the player character and attaching a FireballScript component containing the fireball functionality. To define the fireball spawn direction, the function makes use of the direction variable.

**update()**

The function handles all user inputs and applies all related character reactions accordingly. The function checks if the player is attacking using the character state variable. If the player is not attacking, the displacement variable stores the horizontal axis value. Alternatively, the displacement is set to 0 meaning that the objects position will remain the same.

When the player is not attacking the function checks if a collision occurs. If a collision is undefined on the current frame, the displacement is applied to the player’s position. It is worth mentioning that the program checks if the player is not getting off the screen boundary by checking if the position plus the displacement of the object is greater than 10. Considering that the canvas container starts at 0, the value of 10 provides a small offset to the x -axis of the canvas. If the addition of the two values is less than 0 the motion will not be applied given the current direction value.

The same approach is used when enemies collide with the player. The if statement makes the following check: Is the player moving towards the direction a collision occurred? If the statement returns false, the transformation will be applied to the game object.

If the player is attacking the described code is not being executed. The player state is set to ‘ATTACK’ when: The player presses ‘E’, the attack rate is less than 0 and the current state is not already set to ‘ATTACK’. It should not be possible for the player to attack when an attack is being currently executed. When an attack occurs, the appropriate animation is being reset and all rate timers copy the value of their corresponding initial values.

**onCollisionEnter(collision)**

Stores a collision object if the collision’s game object is an enemy tagged game object.

* 1. SkeletonScript

Provides a basic finite state machine applied to enemy game objects. The enemy moves towards the player until the distance between them is less than the attack range variable. The mechanics of the skeleton are very similar to what has already been described to the player script.

* 1. FireballScript

Contains fireball functionality. The script follows the same logic as in the player and skeleton script. The fireball receives a direction from the player and moves horizontally until a collision occurs.

* 1. EndSceenTrigger

A collision trigger component. When the player colliders with the box collider attached to this game object, the game over scene is being loaded.

* 1. InteractionChecker

Checks if the player has clicked on the canvas. When this happens the introduction, scene loads.

* 1. Scene Initialization Scripts: StartSceneInit, MainSceneInit, EndSceneInit

Initialize the scene of dynamic objects. By checking if predefined objects are present to the scene the components reset or create new game objects accordingly. For example, skeletons can be destroyed by the player. If the player character dies and two skeletons remain the maindSceneInit makes sure that the existing skeletons have their initial spawned state (health value and position). The destroyed skeleton is being recreated all together.

* 1. HealthSliderScript
     1. Class Variables

**Player**

A reference to the player game object.

**maxHealth**

A number representing max health of the player.

**currentHealth**

A number representing the current health of the player.

* + 1. Methods

**constructor()**

**onSceneLoad()**

Stores a player and health related references.

**update()**

On each frame the component checks if the object’s health is up to date. If the variable is not up to date the game object’s scale is transformed by the ratio produced by dividing the max health with the current health. For example, a ratio equal to 1 suggests that the rectangle is full while a ratio of 0.5 suggests that the width of the object has been reduced by half.

* 1. MainGame.js

The main game script is responsible for initializing scene related data along with the engine itself. This script consists of functions and global variables representing scenes canvases and game objects.

* 1. Creating a scene

As mentioned, game objects contain a collection of components, a canvas stores a collection of game objects and a scene a collection of canvases.

To initialize a scene, we need a collection of canvases. The following code fragment initializes a canvas:

let openningCanvas = new Canvas('Openning Canvas', new Vector2D(1578, 969), []);

Since the scene includes only one canvas the program proceeds by creating the array:

let canvasArray = [openningCanvas];

Now the function can create the scene itself:

var openningScene = new Scene('Openning Scene', canvasArray);

To lead the scene, we can use the scene manager likewise:

SceneManager.instance.addScene(openningScene);

* 1. Creating GameObjects

By calling the gameObject’s constructor the application creates a game object with a transform component:

let backgroundGO = new GameObject('backgroundImg', new Vector2D(0, 0), openningCanvas.size);

To add components, we can use the addComponent function:

backgroundGO.addComponent(backgroundAC);

In this case the component is an animation controller managing a collection of sprite animations.

* 1. Adding game objects to canvases

To add a game object to a canvas we can use the addDrawObject function:

openingCanvas.addDrawObj(backgroundGO).

* 1. Sources

*Detweiler, D. (2011).* Case Study: Auto-Resizing HTML5 Games - HTML5 Rocks. Available from: <https://www.html5rocks.com/en/tutorials/casestudies/gopherwoord-studios-resizing-html5-games/>

*HTML Canvas Text.* (2019). Available from: <https://www.w3schools.com/graphics/canvas_text.asp>

*HTML Tutorial*. (2019). Available from: <https://www.w3schools.com/html/>. Last Accessed: 11th Dec. 2019

Javascript Char Codes (Key Codes). (2019). Available from: <https://www.cambiaresearch.com/articles/15/javascript-char-codes-key-codes>. Last Accessed: 1th Dec. 2019

* 1. References

I do not own any of the audio or images used for the project. These sources were found online.

*fireball gif transparent* [online photograph]. Available from: <https://myrealdomain.com/download.html> Last Accessed: 11th Dec. 2019

*The Spriters resource* [online photograph]. Available from: <https://www.spriters-resource.com/pc_computer/heroesofmightandmagic2/sheet/29404/> Last Accessed: 11th Dec. 2019

*Guilty Gear XX #Reload – Arcade Game – Animated Background Sprite Stages* [online photograph]. [viewed 02 Dec 2019]. Available from: <https://www.emugifs.net/guilty-gear-xx-reload-arcade-game-animated-background-sprite-stages/>

Matthey J., (2010) *JM\_FX\_Fireball 01.wav* [Audio]*.* Last Accessed: 11th Dec. 2019. Available from: <https://freesound.org/people/Julien%20Matthey/sounds/105016/>

*Full Moon Pixel Art GIF* [online photograph]. *Available from:* <https://giphy.com/gifs/animated-pixel-art-pixels-Qz5OFEwPwRprq> Last Accessed: 09th Dec. 2019

*GIF to Animated PNG converter [online]* .[ezgif.com](https://ezgif.com/) Animated Gifs Made Easy. Available from: <https://ezgif.com/gif-to-apng>

*2D Environment Platformer Asset Pack: Castle of Despair [online photograph].* Available from: <https://brullov-studios.itch.io/2d-platformer-asset-pack-castle-of-despair> Last Accessed: 5th Dec. 2019

(2019). *click1.wav [Audio].* Available from: <https://freesound.org/people/annabloom/sounds/219069/> Last Accessed: 3rd Dec. 2019

(2019). *Castlevania: Symphony of the Night OST: Metamorphosis I* [Video]. Available from: <https://www.youtube.com/watch?v=EjVoUFrdYFQ> Last Accessed: 3rd Dec. 2019

GhostGuardian\_Attack\_01.wav [Audio]. Available from:<https://freesound.org/people/STAudio/sounds/490515/> Last Accessed: 3rd Dec. 2019

*Necropolis - Heroes of Might and Magic IV (4) OST*. (2011). [Video]. Youtube. Available from: <https://www.youtube.com/watch?v=Mt2paelUYwo> Last Accessed: 4th Dec. 2019

*Castlevania: Symphony of the Night OST: Abandoned Pit*. (2010). [Video]. Youtube. Available from: <https://www.youtube.com/watch?v=YhIf_zV6yK0> Last Accessed: 6th Dec. 2019

*Castlevania SOTN: Lost Painting*. (2007). [Video]. Youtube. Available from: <https://www.youtube.com/watch?v=IYwS6481oQk> Last Accessed: 3rd Dec. 2019

Forum for RPG Lounge and RPG Lovers. [online]. Available from: <https://rpglounge.rpg-board.net/t513-browserquest-tips-tricks> Last Accessed: 9th Dec. 2019