Sourav Basu

2D PLATFORM GAME

Developed using Unity game engine

Thesis
CENTRIA UNIVERSITY OF APPLIED SCIENCES
Degree Programme in Information Technology
December 2017

Centria
UNIVERSITY OF APPLIED SCIENCES

ABSTRACT

Centria University of Applied Sciences	**Date** December 2017	**Author** Sourav Basu
Degree programme Information Technology		
Name of thesis 2D PLATFORM GAME. Developed using Unity game engine		
Instructor Kauko Kolehmainen		**Pages** 54 + 44 appendices
Supervisor Kauko Kolehmainen		

The aims of this thesis were to create a game in 2D using the Unity game engine, and to learn about the new development tools introduced throughout each update and Unity versions. Unity is a cross-platform game engine, therefore releasing the same project on different platforms is quite easy. With smartphones and handheld devices increasing in popularity, 2D games have attracted a lot of attention. Moreover, with updates over the past years, Unity has seen many changes in 2D development with the addition of new tools and mechanics, making game development much faster.

The game developed for this thesis is a platformer game. This means that the character in the game has to try to avoid traps, enemies and falling into death and must try to reach the finish line at the end of each stage. There is a health system in the game for the player and enemies, so that they are attacked few times before getting killed. There is a point system, which the player is required to fulfill in order to end the stage. This game consists of three different stages, the third being the boss stage where a stronger enemy spawns. Killing the boss is the main objective of the game. Even though the game has only three stages, it is possible to create more stages in between, using saved assets in the projects assets folder, and this will be further explained along with other topics later into the thesis.

The game created as a result of this thesis is a good example of a modern-day 2D game. It is possible to use this as a step-by-step tutorial for creating a game.

Keywords:
2D, Unity engine.

CONTENTS

APPENDICES

Game Scripts

1 INTRODUCTION

The project is a platformer genre game, where the aim is to reach endpoint through obstacles and enemies in each stage. It was made using Unity 2D and it is a Windows Operating System game. Unity was chosen to develop the gamewas, that it is becoming more and more famous in producing some amazing games.A free version was used to make this game, and even though not all the features are available in this version, the tools provided were more than enough to develop a simple 2D game. Unity is an ultimate game-developing platform where it is possible to release the same project on multiple and various platforms, such as Linux, Mac, Android, iOS, Xbox, PlayStation and more.

Even though 3D games are more attractive and visually more realistic, 2D games are also very popular as well, especially with smartphones and tablets gaining popularity each year increasingly. With that in mind, Unity developers decided to release 2D features in 2013 with completely new 2D features and toolsets. With these additions, developing 2D games is much easier now, compared to earlier.

In this thesis, I will take a closer look at the new features of Unity 2D and explain how a game can be developed even with a basic knowledge about programming and Unity game engine. While working on the game, I was able to acquire a deep understanding of C# language. Although there were some Unity syntaxes that were new, but it was not hard to comprehend them. The purpose is to show the advantages of using Unity and to learn about the latest 2D features that have been updated over the past years, since 2D was introduced. The game that was developed was solely for study purposes and is used as an example in this thesis. This thesis can also be used as a tutorial to create a basic 2D game from start to finish.

Certain terms used in this thesis will be exactly the same as in Unity's documentation, for example GameObject and Rigidbody. Scripts for the game can be found in the Appendices.

2 OVERVIEW

Unity is a multiplatform game engine that supports 2D and 3D graphics.Games were commonly scripted in C# and UnityScript, similar to JavaScript, which makes them much easier to understand. Currently, Unity 2017.2 is the most updated version and at the time of Unity 5 and Unity 2017.1 release, Boo and UnityScript were removed from scripting languages, respectively. Unity version 5.5.1 was used to develop the game for thesis. (Alexsadr 2014.)

2.1 History

Unity is a cross-platform game engine that was developed by Unity Technologies. Unity Technologies was founded by David Helgason, Nicholas Francis and Joachim Ante in 2004. Many global companies helped Unity Technologies by funding the project, in hopes of developing an engine everyone could afford.The Unity engine was developed using C and C++ programming languages. For scripting and coding,C# and Java languages are used. (Corazza 2013.)

In 2005, at Apple's Worldwide Developers Conference, the first version of Unity was released. At the time of release, Unity 1 could only build projects for Mac systems. With successful responses from Unity 2 in 2007however, Unity Technologies became more famous and well-known among amateur and professional game developers. Unity 3 and Unity 4 were released in 2010 and 2012, respectively. These versions of Unity first included free licenses and many professional game-developing tools, so that amateur programmers could design and create various types of games and software. (Seraphina 2013.)

As of this day, Unity 2017.1 is the latest version on the market and in the Unity store, and it includes lots of changes and updates were made both in 2D and 3D graphics section.

2.2 Asset Store

In 2010Unity introduced the Asset Store, where developers and other users can download assets within the Unity editor. It is a library, where both free and commercial assets can be found. These assets were created by Unity technologies and by various members of the community.

The assets that are available in the assets store are:

- 3D Models
- Animations
- Audio
- Particle systems
- Scripting
- Shaders
- Textures
- Materials
- Unity Essentials and
- Services.

2.3 Licenses

Unity game engine mainly has three licensing options. Unity Personal, Unity Plus and Unity Pro. These differ from each other mostly in the features, bug reporting or requiring customer assistance. Unity Free version can be downloaded by any user and is the most limited version. It has all the necessary tools and features to develop a game. Unity also gives the user the freedom to upgrade to Pro in the middle of project without causing technical issues, as upgrading the license will only add new features to the user interface.

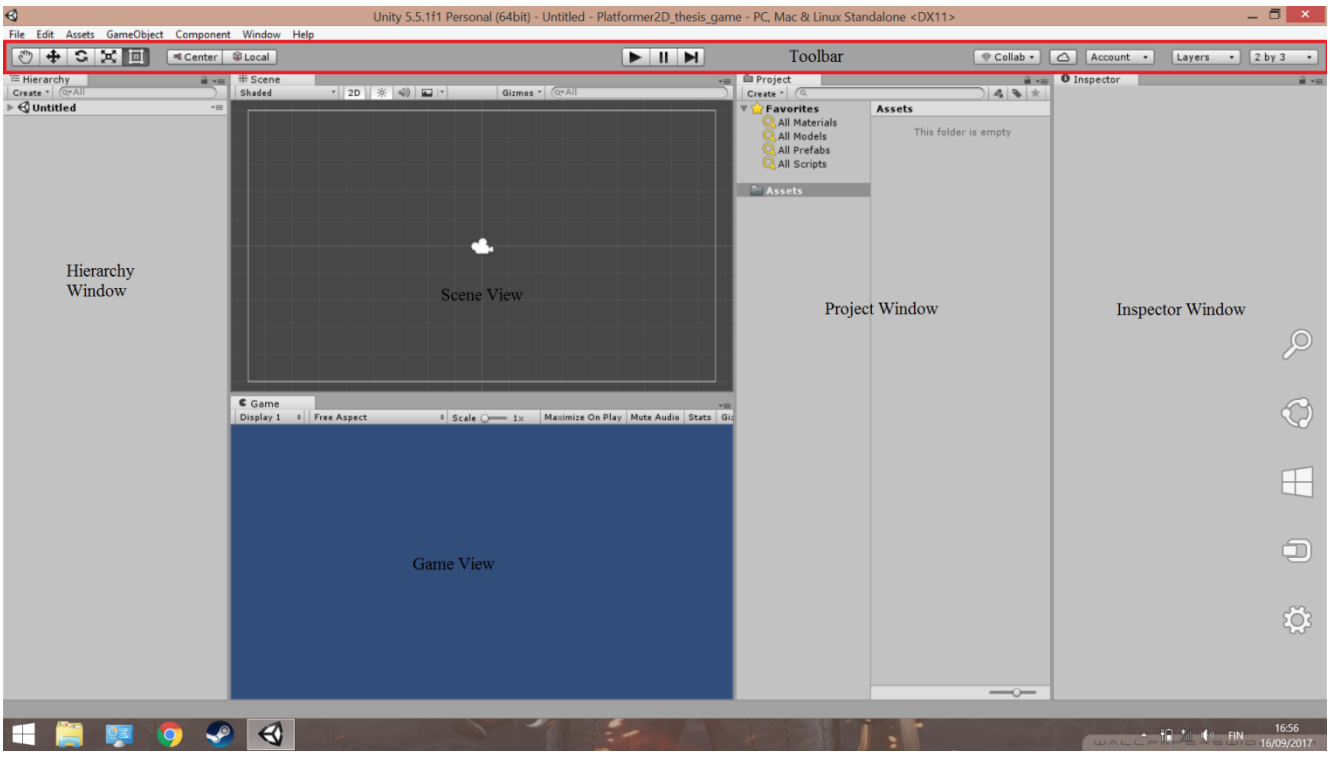
As per Unity's EULA Agreement, users using the free version should not have annual revenues more than 100,000 USD, or else their permission will be retracted. In that case users should either upgrade their Unity licenses to Plus or Pro, as Unity Plus allows annual revenues up to 200,000 USD and Unity Pro does not have limit. Unity Pro allows users to edit their splash screens of the game, which is not possible in the free version. It has also the option to improve performance and details of the games.(Downie 2016.)

2.4 Unity Interface

Unity has a built in editor interface, where it functions as the main window or workspace for a project. In the workspace or main editor window, there are several other tabbed windows, each having a specific purpose and function. (Unity Technologies 2017a.)

By default, the main editor window includes:

- Project Window
- Hierarchy Window
- Scene View
- Game View
- Inspector and
- Toolbar.



GRAPH 1 Unity Editor Layout, using 2 by 3 layout format

The Project Window shows all the folders and sub-folders that are related to the project. It shows a list of assets and scripts that are imported to the project folders. Assets can be used directly from the project window by dragging them into the scene view or using as a script for a particular GameObject. It

is also possible to search and select assets from Unity asset store from the Project Window. (Unity Technologies 2017b.)

The Hierarchy Window contains a list of GameObjects that are in-use in the current scene. When GameObjects are added or removed in the scene, they usually appear or disappear, respectively, in the Hierarchy Window. By Unity's understanding of parenting, it is possible to make a GameObject child of another GameObject by dragging them onto a parent in this window. In such process, this child GameObject inherits the attributes, such as the position and the rotation values of its parent GameObject. (Unity Technologies 2017e.)

The Toolbar includes some basic controls of the project. These include transform tools and gizmo toggles for the scene view, play/pause/step buttons for the game view, cloud buttons or Unity services, layer drop-down for displaying objects in the scene view and layout drop-down for the arrangement of views. (Unity Technologies 2017g.)

The Inspector Window displays detailed information about the selected GameObject, including all of its properties and the components attached to it. This window enables the user to modify the functionality of the GameObject, such as editing, adding or deleting new scripts or physics 2D materials, and making various changes to the scene.(Unity Technologies 2017f.)

The Scene View is the interactive window where most of the editing takes place and it is also one of the key features of Unity. Here users are able to select, move and modify characters, cameras, lights, GameObjects and different other assets, making editing very quick and easy.(Unity Technologies 2017c.)

The Game View Window shows a preview of the current scene, without building the project. By pressing the play button in the toolbar, Unity runs the active scene and the game is displayed in the game view. To capture any image and display in the game view, a camera component is required in the scene and hierarchy windows.(Unity Technologies 2017d.)

3 GAME DEVELOPMENT

At the start of any project, there should be a goal and proper planning to achieve success. A clear concept about the goals makes it easier to plan the steps ahead and make the necessary adjustments. There are also certain requirements that are needed to be fulfilled, in order to make a game more potent. To make a game playable and interesting, it must have proper game design, graphics and game play. With the help of a game engine, proper planning and a clear vision, it is easier to make a simple yet beautiful game.

3.1 Planning and requirements

While working on the project, several steps were planned beforehand so that the process would become easier. The first step was to design the game, which was a fairly easy task as playing various games have helped to acquire sufficient knowledge about platformer genre games. The idea of designing is to keep the player engaged with the game and introduce new challenges on every corner. The objectives for completing the stageswere an essential part of the game design. The basic platformer concept is to move from one point to another, but by including objectives on each level, the player is expected to fulfill these in order to complete the current level.

Acquiring textures for the game graphics was the most important and toughest part during the development. As it was required to search several websites and Unity's asset store for the desired material, which is free to use. There were several moments during the design phase where the process was altered due to the fact of unavailable asset. With the basic idea and knowledge about graphics design and Photoshop, such obstructions can be avoided. For the gameplay part, the input was kept fairly simple and intuitive, so that players are able to focus on the game rather than on the controls.

For developing the game, it is essential to write the code. The scripts are attached to the gameObjects, which functions in the game accordingly. MonoDevelop is the built-in Integrated Development Environment (IDE) that is provided by default when downloading the Unity game engine. Visual Studio can also be used for scripting, but it is required to set the default IDE from inside Unity.

3.2 Scripting

In order to add velocity and motion to the player character, first we need to add a script folder to the player GameObject and write some code in it. It is possible to create a script simply by right-clicking on the Project Window, then create a C# script. Another possibility is to select the player from the Hierarchy view and add a New Script component. On opening the script from Unity, it can be seen that the class created derives from the base class MonoBehaviour. MonoBehaviour is a class in UnityEngine and it is necessary to derive from it.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class practiceUnity : MonoBehaviour {

    // Use this for initialization
    void Start () {

    }

    // Update is called once per frame
    void Update () {

    }
}
```

GRAPH 2. Default Unity C# script

Graph 2 also illustrates two functions, Start and Update, which are called by the game at various intervals. For example, void Start() is called immediately when the game is run and void Update() is called once per frame.

Apart from these two functions Awake(), FixedUpdate() and LateUpdate() are also used in certain cases. The Awake() function is called before Start(), and is executed while the game is being loaded. Unlike the Update() function, the FixedUpdate() is called during every framerate frame, which means that the time interval between each FixedUpdate() called will always be the same, whereas it may vary in case of the Update() function. When working with physics, the FixedUpdate() function is used as it adjusts the physics of an object several times each frame. The LateUpdate() is called after all other Update functions are called. It is basically used for the camera to follow as it will be executed after the a certain object is moved in the Update functions. All methods and functions must be called in Up-

date() or FixedUpdate() function, as these are the main methods of MonoBehaviour. (Unity Technologies 2017j, Unity Technologies 2017k.)

There are several named methods and variables that I have used while writing the codes and such texts will be enclosed by apostrophes(") later in the thesis.

3.3 Unity 2D features

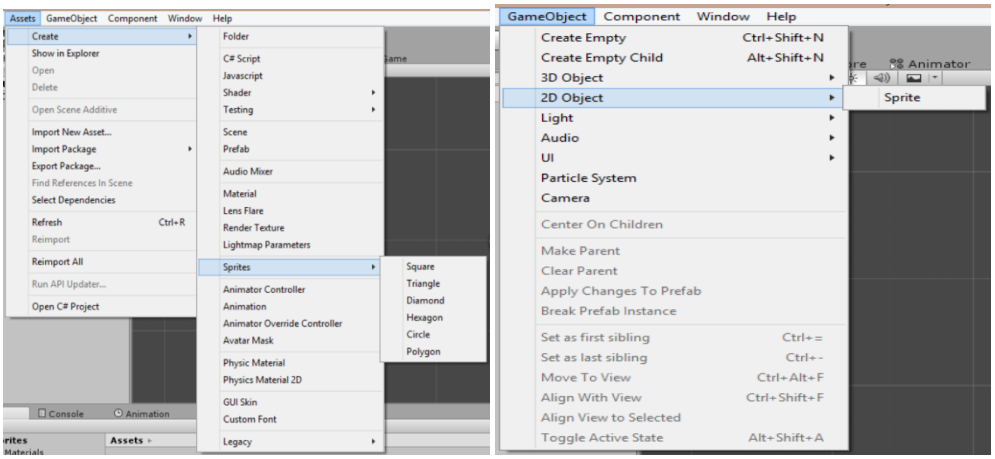
The Unity game engine has built in 2D toolsets, which are essential in developing 2D games. Without these toolsets, previously game developers had to use third party software to modify sprites. Graphics or textures in 2D usea sprite renderer whereas in 3D a mesh renderer is used.Present 2Dtoolsets are also used to customize and alter the physics mechanics of the sprites, making development of 2D games much easier and faster.

3.3.1 Sprites

This section will provide the necessary details about sprites, how sprites are created, edited, rendered and how to create animations of sprites. Sprites or graphical elements in 2D gamesplay an important role as they give the game their visual appearance. Sprites are two-dimensional bitmap or graphical objects used in 2D games. In Unity, sprites are defined by Texture2D, pivotal point and a rectangle. Each sprites has a texture2d element for visual appearance, a pivotal point where the image can be rotated and it has also a rectangle borderline. Because of the rectangular mesh, it is also possible to join vertex to vertex called vertex snapping, making aligning objects much easier. (Unity Technologies 2017r.)

Sprites are also a type of asset in Unity project and they are usually found in the Project Window. Sprites used for the game in this thesis were downloaded from the internet and were imported to Unity. There are many 2D sprites available on the internet and in the Unity store, and they are completely free for personal use. To import sprite assets into Unity, go to Assets> Import New Assets which will then open the File Explorer window in Windows OS or the Computer's Finder in Mac OSX. After selecting an image, Unity will import the file and it will be shown as an asset in the Project Window. To use

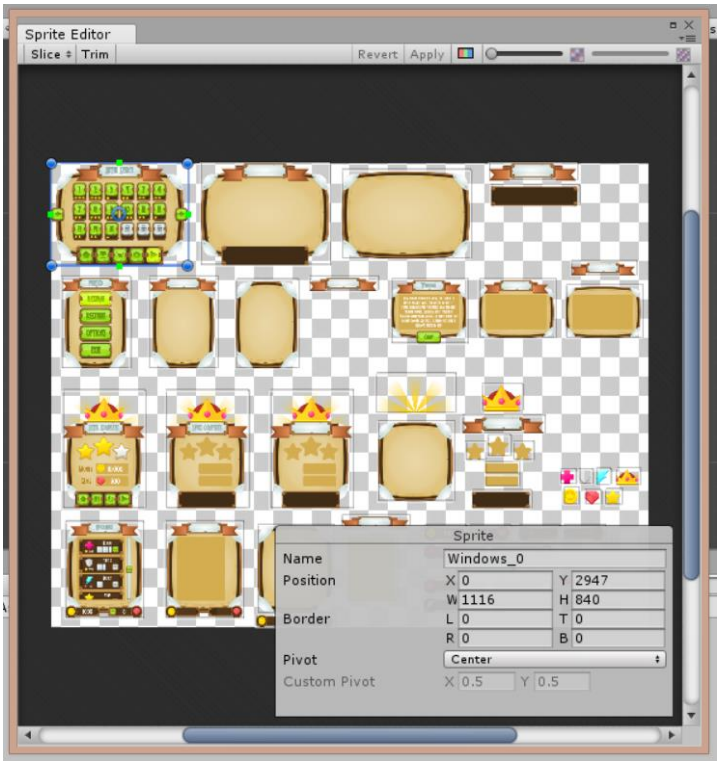
such images as sprites in 2D, it is required to change the texture type of the image to Sprite (2D and UI) from the Inspector Window.



GRAPH 3. Assets tab and GameObject tab

In Unity Editor, it is also possible to create spriteGameObjects. To create one, go to Assets>Create >Sprites and select the desired shape as an image, as shown in Graph 3. A sprite asset will appear on the Project Window, which can be used simply by dragging it into the Scene View or Hierarchy Window, or just by double-clicking. It is also possible to create a sprite by selecting GameObject>2D Object>Sprite, as shown in Graph 3. This way Unity will create a GameObject with a sprite renderer, a new sprite, in the Hierarchy window with no sprite of the image. To insert an image, select the new sprite and from the sprite renderer in the Inspector window, edit the sprite by clicking the small circle and input any desirable image or sprite. (Unity Technologies 2017i.)

The Sprite Editor is an editing tool that can be used from the Inspector Window, while a particular sprite texture in selected. Oftena single image may contain several different sprite textures that can be used in game development. In order to use them individually, it is necessary to separate them from its whole image texture. Unity has provided a sprite editor tool to obtain single elements from a composite texture.To extract individual sprites from a single image, first set the SpriteMode in the Inspector Window while selecting the image texture, to multiple and click the Sprite Editor.



GRAPH 4. Sprite Editor window

In the Sprite Editor window, along with the texture image, there are several controls in the top bar of the window. Such controls include zoom slider, pixilation slider, apply, revert, trim and slice. The most important control is the slice menu, which gives the option for dissecting the image and obtaining single elements.Graph 4 shows how elements are sliced individually from a composite image or sprite-sheet. (Unity Technologies 2017h.)

As seen in Graph 4, gray rectangle borders in the spritesheet depict different and individual sprite textures. It is possible to slice off graphic elements just by selecting automatic type from the Slice menu and clicking slice. In this process, Unity itself is detecting all the graphics textures and bordering them separately. If desired, it is also possible to slice manually by simply clicking and dragging around the image.

Sprite Renderer is a renderer ysed by Unity for rendering sprites in 2D, whereas in 3D a Mesh Renderer in used. The Sprite Renderer is a component attached to all the sprites that are present in the Scene view or Hierarchy window. In 3D mode, an object's appearance depends on the lighting and camera position, but in 2D, images are displayed according to their position, scale and rotation in 2D coordinates.

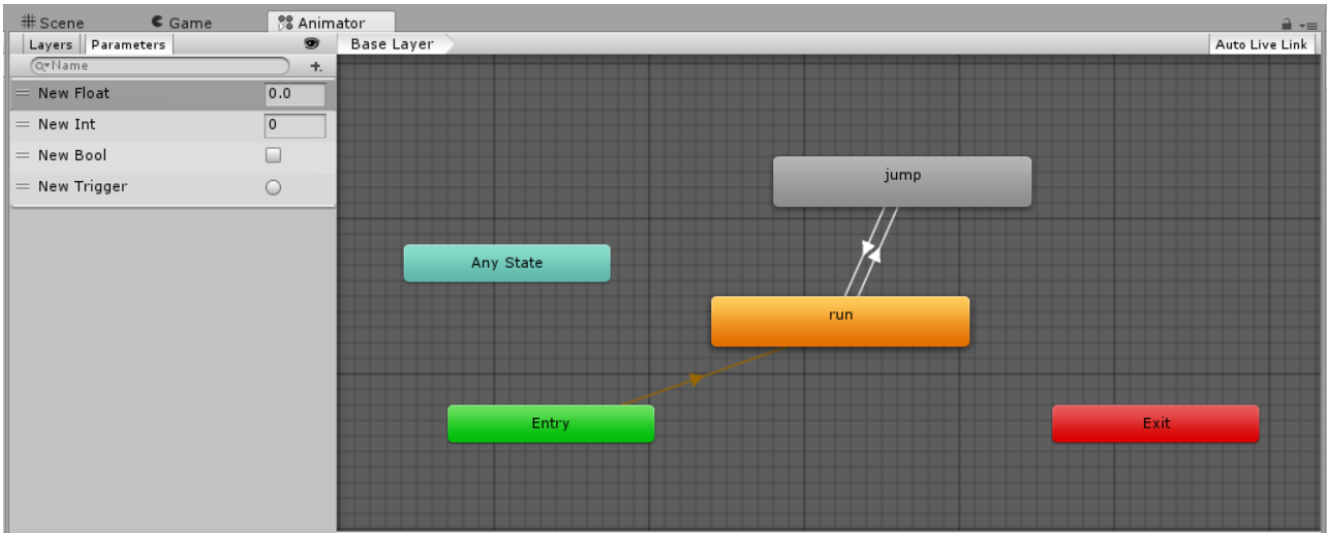
With Sprite Renderer attached, it is possible to alter the image properties by varying colour, material, sorting layer, layer order or even flipping the image in X and Y axes. In colour option, it is possible to change the colour or the level of transparency, whether the material interacts with lightning and layer options are used for setting priority to a sprite during rendering phase. (Unity Technologies 2017s.)

GRAPH 5. Walking Animation

Graph 5 shows eight separate images where a character has different stages of walking. When the images are looped in fast motion, it will appear like the character is walking.

While developing a game, scripting and graphical designare both of the same importance. In order to make a game look more realistic and lively, animation is added to the sprites. Creating animations with the Unity Engine is quite fast and easy. It is possible to create animation using either hand drawing or just by downloading spritesheets from the web. When a set of images, as we can see in Graph 5, are selected and dragged into the Scene View or Hierarchy Window, a GameObject is created. Unity then prompts a window called Create New Animation, in order to save the animation file. Along with the animation file, a controller component is also created, and will be attached with the GameObject.It is also possible to create animations from the Animation Window. To open the Animation Window, simply select Window from the tab, then Animation. To create animations and attach them to an object, select the GameObject from the Hierarchy Window, and drag an image into the time scale on the Animation Window. It is also possible to alter the speed of animation by changing the values beside the samples in the Animation Window.

To animate a character or sprite, a controller is required to access its Animator. To open the Animator window, select Window from the tab and then Animator, which will create the Animator Window. Each GameObject can have different animations and it is possible to control them from the Animator using various parameters.



GRAPH 6. Animator Window

Graph 6 shows the Animator Window where the orange coloured state contains a run animation and is defined as the default state. Which means, that whenever the animator is called, any animation in the run state will animate immediately.

The Animator window or State Machine window has various layer control, which will be explained later in the thesis, alongside with its parameters. There are four parameters float, integer, boolean and trigger. When a GameObject has two or more animations, parameters sets which animations to play and when. Parameters are set by clicking the white arrow connecting the run state and the jump state, as seen in Graph 6. These lines are called transitions. Suppose the transition from the run state to the jump state animation is defined as a Boolean parameter;if the value is true then the jump animation is played and if the value is false, it will continue with the run animation.Animations linked to any state will run immediately, whenever the set parameter is true and exit state forces the animator to quit.

3.3.2 Physics2D

Unity has 2D physics settings, which include various physics components. By using these components it is possible to add physics properties to a GameObject, such as adding force, gravity or collision.

Rigidbody2D is a component which, when attached, puts the Sprite under the control of Unity's physics engine. Furthermore, it is possible to change the object's mass, gravity, drag, body type, and also applying forces using scripts. Rigidbody2D has three body types, dynamic being the most common as

it allows objects to move similarly to movement in the real world. Kinematic body types are only able to collide with dynamic body types and these do not have gravity in them. Even though it is possible to move them using velocity in scripts, but they will not be affected by force or gravity. Static Rigidbody2D is not designed to move and behaves as an immovable object even when colliding with dynamic body types.(Unity Technologies 2017u.)

Colliders2D components are added so that objects with Rigidbody are able to collide with other objects whilst a collider is attached to them. Colliders are proximate shapes that determine the surface for collision. Box Collider2D and Circle Collider2D are the basic shapes of the collider. Polygon Collider2D gives a precise collider shape by modifying the line segments and edges freely. Edge Collider2D is the same as Polygon Collider2D, except it does not have to be an enclosed shape, such as a line or a curve. (Unity Technologies 2017v.)

Physics Material 2D components add friction and bounciness to the objects. Changing friction values changes the coefficient between the self-collider and the interacting collider, which provides more or less a slippery surface. Bounciness values set at 0 provideno bounce and any value more than 1 add bounce property without energy loss. If the value is between 0 and 1, then it adds potential energy property to the object, where it loses energy through each bounce until it comes to rest. (Unity Technologies 2017w.)

Joints2D components are used to attach one object to another object with a Rigidbody2D component or just a fixed position in Scene view. There are several Joints2D with different interactions. Distance Joint2D keeps two objects apart at a certain distance. Fixed Joint2D keeps the object in a fixed position and denies any movement if collided with other objects, when its break force and break torque are set to infinite. Friction Joint2D decreases the surface velocities between two objects to zero. Hinge Joint2D allows an object to rotate in place, fixed in a position in world space. It can be rotated passively or by applying motor force to it. Relative Joint2D allows two objects to maintain distance position and rotation based on each other's location. Slider Joint2D allows objects to slide along the line from their initial position to their declared position. Spring Joint2D allows objects to behave like a spring. Spring force will apply along the axis and will maintain to keep a distance between objects. Target Joint2D is a spring type joint, which allows an object acting under gravity to move to a specified location in the world space and fixes its position. Wheel Joint2D acts as a rolling wheel, on which other objects can move. It has a suspension type spring to keep the distance between two objects. (Unity Technologies 2017x.)

The Constant Force2D component allows us to add constant force to a Rigidbody. It lets the object to accelerate over time instead of just starting at maximum speed. It applies both linear and angular continuous forces to the GameObject.

Effectors2D components are co-related to the Colliders2D present in the object. Colliders2D must be added and the option "Used by Effector" should be checked in order for Effectors2D components to function. Except for Platform Effector2D and Surface Effector2D, all other effectors require both the options "Used by Effector" and "Is Trigger" to be checked.Platform Effector2D can be used to allow collisions only from one-way. Buoyancy Effectors2D is used to acquire a floating effect on the GameObjects. Area Effector2D applies magnitude force and angle to the object colliding into it. Point Effector2D adds attraction and repulsion force to the colliding object. Surface Effector2D provides additional speed when other objects collide with it. (Unity Technologies 2017y.)

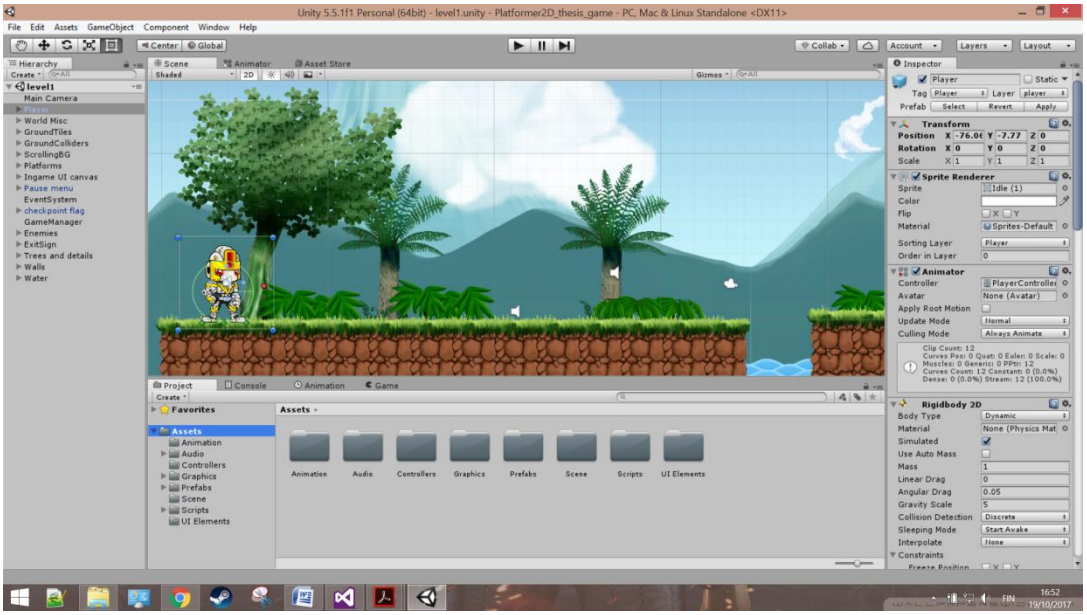
4 THE IMPLEMENTATION PROCESS OF THE GAME

The game built for this thesis can be used as an example on how to make 2D games. From this section onwards, I will be explaining the basics of the making of a game and how it works. Here I will explainthe necessary script functions that are required to create a basic 2D game. All the scripts used for the making of this game can be found in appendices.

4.1 Creating a New Project

To create a new project, we need to select New on the upper right corner of the window that opens, and then select a location to save your project giving it a project name. Below the location section, we need to check 2D as we will be working on two-dimensional world space. Assets package views a set of items that can be imported when creating a new project.

4.2 Basic Structure



GRAPH 7. Basic layout of the game

Graph 7 shows the basic layout structure of the game where the Project window contains all the necessary assets needed to develop the game. In the Animationfolder, animation clips of various characters

and sprites are kept. The Audiofolder contains sound files which were used. Controllers have all the animation controllers required to animate certain objects, since, as explained earlier, each individual sprite must have a controller for controlling animation clips. The Graphicsfolder contains all the textures used in the game, for example, trees, clouds, mountains. The Prefabsfolder contains assets that are saved as a single asset and these can be used multiple times. As we can see in Graph 7, the Hierarchy window contains some objects that are marked in blue, these arePrefab assets. Suppose, a change has been made to a Prefab instance, and there are several assets in the game world of the same prefab. Instead of changing them separately, prefab options lets us make changes in all of them, just by selecting the accept button in the Inspector window.The Scenefolder contains all the different scenes of the game, for example, the Start menu scene, the Level1 scene, and so on.The Scripts folder is where all the scripts used for the game were stored.

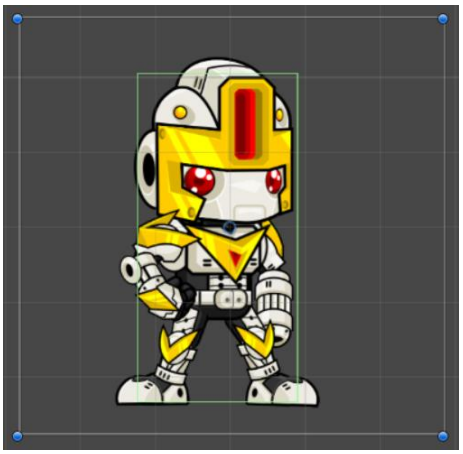
4.3 Player

In this section, I will explain about the characteristics of the player. The player is the most important character of the game, as it lets users to interact with the game world. Users can control the player by various inputs resulting different actions, for example, walking, jumping, shooting.

4.3.1 Player Physics

Before moving on to the movement section, it is important to add Rigidbody2D and Collider2D components to the player character. As it is discussed above, it is necessary to add Rigidbody2D so that physics can be applied to the player sprite and Collider2D gives sprites a physical property and for colliding against other colliders.

In Rigidbody2D setting, the value of mass is default to 1 and the gravity is set to 5. However, it is possible to change the value to acquire the desired physics behaviour. In 2D games, only the X and Y axes are used. In order to restrict any rotation through the Z axis, it is necessary to check the “Freeze Rotation Z” box, which can be found under constraints in the Rigidbody2D setting.



GRAPH 8. Box-Collider2D added to the player sprite

While choosing colliders for the sprites in 2D, the box collider is used the most, because of the two-dimensional world. It is also possible to choose a polygon collider for more precise measurement. After selecting the desired collider, edit the size of the collider so it borders with the sprite texture. In Graph 8 the green box around the sprite indicates the collider.

With the addition of Rigidbody2D, it is possible to move the player horizontally or vertically using velocity. Colliders will enable the player to collide with certain GameObjects in the game.

4.3.2 Movements

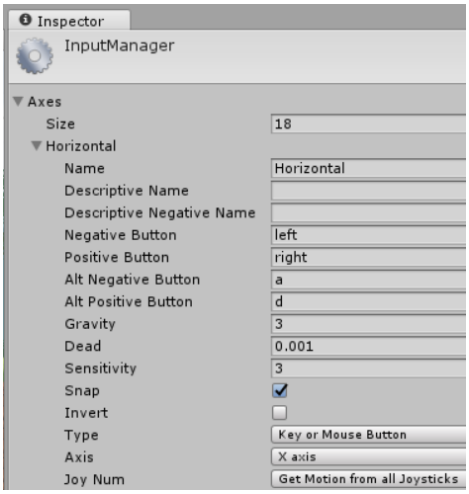
The scripting for the player movements were done in "PlayerController" script. In this script, the main class "PlayerController" was derived from base class "CharacterComponents". "CharacterComponents" has two derived classes, one being "PlayerController" and the other "EnemyController", which I will explain later in the Enemy movements section. Note that the "CharacterComponents" class is acting as a base class and its derived classes are attached to the GameObjects.

As this is a side-scrolling 2D game, the player is restricted only to move horizontally. It is also included for the player to gain vertical height through jumping. The movements in the script are done by accessing the velocity property in Rigidbody2D. Though it is also possible to add motion to the player by using the Addforce() method, but the aim was to give the player a linear speed while moving, rather than gradually gaining speed.

```
PlayerRigidbody.velocity = new Vector2(Input.GetAxis("Horizontal") * moveSpeed, PlayerRigidbody.velocity.y);
```

GRAPH 9. For moving in horizontal direction with a constant speed

Graph 9 shows a simple code, whichlets the player move horizontally. There are two variables in the line "PlayerRigidbody" and "moveSpeeed". PlayerRigidbody is a property of type RigidBody2D in the script, which gets and sets the rigidbody2D component attached to the player. Before RigidBody2D methods and properties can be called and used in the scripts, it is necessary to set it in Start() method by writing "PlayerRigidBody = GetComponent<RigidBody2D>();". Another variable "moveSpeed",which is a float, is called from itsbase class "CharacterComponent". "Vector2" is a type in the Unity Engine, which contains float values in X and Y components."GetAxis" is a static method in "Input" class, which returns a virtual axis named "Horizontal", where the value is either 1 or -1.



GRAPH 10. Horizontal axis in Input under Project settings

Graph 10 shows that there is a positive and a negative button, which describes that when the left arrow is pressed the value -1 will be registered and 1 when pressing the right arrow. Therefore, whenever the desired key is input, whether it is the left arrow or the right arrow or a and d, the velocity in the Rigid-Body2D translates the X position of the player by 1 or -1 and multiplies it by the "moveSpeed" value without changing the Y position. Furthermore, when the value is 1, the player should be facing right and while the value is -1, it should be facing left.

```
64      public virtual void changingDirection()
65      {
66          facingRight = !facingRight;
67
68          transform.localScale = new Vector3(transform.localScale.x * -1, transform.localScale.y, transform.localScale.z);
69      }
```

GRAPH 11. "ChangingDirection" function from "CharacterComponents" script

```
206     // Flipping the character
207     private void TurnPlayer()
208     {
209         if (Input.GetAxis("Horizontal") < 0 && facingRight || Input.GetAxis("Horizontal") > 0 && !facingRight)
210         {
211             changingDirection();
212         }
```

GRAPH 12. "TurnPlayer" function from "PlayerController" script

In graph 11, the "changingDirection()" method is written in the base class so that both derived classes can use the same code. Here a virtual keyword is used, so that the same function can be overridden in derived classes, if it is required. It is necessary that all the derived classes should inherit each and every base class methods.Without a virtual keyword, overriding cannot be done. In the graph, "facingRight" is acting as a Boolean variable, which only returns either true or false. Therefore, if the player character's X position is positive, then "facingRight" is true; otherwise, it is false.So when the condition in the if statement is true, changeDirection() will be executed and the player will face the appropriate direction. In graph 12 line 209, when the value of horizontal input is -1 and the player is facing right or if the value is 1 and facing left, "changingDirection" is called.

Before adding the jump function, the character should be able to differentiatethe ground from other GameObjects. To determine whether the player is on the ground or not, we need three variables:transform array of multiple ground points, float for giving those points a radius and LayerMask to define which GameObject is the ground.For the Transform array, create the necessary amount of empty GameObjects and make them children of the player. These points should be positioned at player’s feet where it is supposed to touch the ground. Layer Mask only takes a layer as its selected option. When a GameObject is selected, on the upper right corner in the Inspector View, there is an option named layer. It is possible to determine a certain layer for a GameObject or to add a new layer. In the game, I used "ground" as layer name for my ground colliders.

```
263     // Multiple points used to check if Player is in ground.
264     private bool IsGround()
265     {
266
267         if (PlayerRigidbody.velocity.y <= 0)
268         {
269             foreach (Transform point in groundPoints)
270             {
271                 Collider2D[] colliders = Physics2D.OverlapCircleAll(point.position, groundPointRadius, whatIsGround);
272
273                 for (int i = 0; i < colliders.Length; i++)
274                 {
275                     if (colliders[i].gameObject != gameObject)
276                     {
277                         return true;
278                     }
279                 }
280             }
281         }
282
283         return false;
284
285     }
```

GRAPH 13. "IsGround" function in "PlayerController" script

In Graph 13 line 267, the method checks if the player's velocity at Y component is less thanor equal to 0. Then, for every element in the ground points, it attaches a collider at the point's position with a fixed radius, and in line 271 it checks if the colliders are overlapping with the layer that is set on "whatIs-Ground" layer mask. In the for loop, it checks that the colliders are overlapping with the layer,and it is supposed to return, whether the player is in the air or the ground. If all the conditionsare true, then the player is on the ground, else the method returns false.

Now that the player understands which layer is the ground, it becomes much easier to add jump function to it. Because it would be rather unrealistic to jump without even standing on a solid surface. To add jump is comparatively the same as moving. Instead of adding velocity to the X component, we should add it to the Y component. Of course, a float variable is required to state the jump distance.

```
223         if (Input.GetKeyDown(KeyCode.Space) && Grounded)
224         {
225             PlayerRigidbody.velocity = new Vector2(PlayerRigidbody.velocity.x, jumpHeight);
226             //Plays audio attached to Player.
227             GetComponent<AudioSource>().Play();
228             //Sets the trigger for jump in animator state machine.
229             ThisAnimator.SetTrigger("jump");
230         }
231
```

GRAPH 14. Adding velocity to the player's Y component

Graph 14 shows an if-statement, whenever the spacebar on the keyboard is pressed and the player is on the ground,the velocity of the player character's Y component is changed by the value of "jumpHeight". An audio file is also attached and is played wherever the player jumps and the jump animation is set to trigger.

Along with jump, I have also added a double jump function, where the player will be able to jump while in mid-air, but only once before landing on the ground.

```
//Allowing the Player to jump twice.
if(Input.GetKeyDown(KeyCode.Space) && !Grounded && !doubleJump)
{
    PlayerRigidbody.velocity = new Vector2(PlayerRigidbody.velocity.x, jumpHeight);
    GetComponent<AudioSource>().Play();

    ThisAnimator.SetTrigger("jump");
    doubleJump = true;
}
```

GRAPH 15. Double jump codes

Graph 15 shows a few lines of code for the player to be able to jump twice. For this purpose, a Boolean variable is required, in this case I have used "doubleJump", which is set to false whenever the player is on the ground. When the player jumps twice in quick successions, "doubleJump" returns true and restricts the player from executing the jump function any further until the player lands on the ground.

4.3.3 Attack System

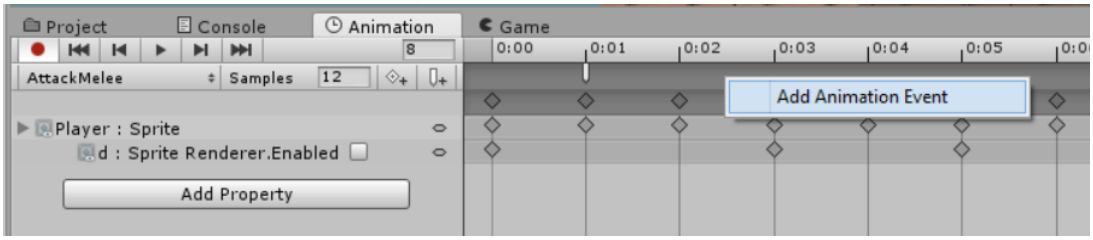
In this section, the player will be able to swing an attack and shoot projectiles in the direction it faces. When the player presses the predefined button, the character swings its laser sword, or melee attack, and damages an enemy if it is in range, or shoots projectiles that fly in a straight line and damage the first enemy they hit. To create such functionality, an if-statement is required to check whether the key is being pressed.

```
217         if (Input.GetKeyDown(KeyCode.Z) && !immortal)
218         {
219             ThisAnimator.SetTrigger("attack");
220         }
```

GRAPH 16. Codes for melee attacking

Graph 16 shows that an animation is set to trigger when the key "Z" is pressed. During this animation, an event will execute a function that will enable a collider thus damaging the enemy. The collider is disabled and is a child of the GameObject of the player character, and when the animation is played, this event will enable the collider for a brief moment.

To add an event to an animation, select the player character and in the animation window, right-click on the timeline and click “Add Animation Event”. Upon adding an event, a small white rectangle appears, where a function can be enabled from the Inspector window. The functions used in the scripts attached to the player will only be shown in the drop-down menu.



GRAPH 17. Animation window with an added event

First, add an edge collider to the scene view and make it a child object of the player character. I preferred using edge colliders, because they give the option to draw them freely, and then place them near the character sprite. In the scripts, an EdgeCollider2D type is needed, so Unity can identify the collider the script is supposed to act upon. I also made a read-only property of EdgeCollider2D, which returns the collider.Then a function is written,enabling the collider when it is called in an animation event and then the collider returns to its disabled state after the full animation runs once.

```
99      public void MeleeAttack()
100     {
101         SwordCollider.enabled = true;
102     }
103
```

GRAPH 18. Enabling the collider from "CharacterComponents" script

```
233         // Energy decreases on every use.
234         if(energy.CurrentValue>=5)
235         {
236             if (Input.GetKeyDown(KeyCode.X) && !immortal)
237             {
238                 energy.CurrentValue -= 5;
239                 ThisAnimator.SetTrigger("shoot");
240                 ShootingProjectile();
241
242             }
243         }
244
```

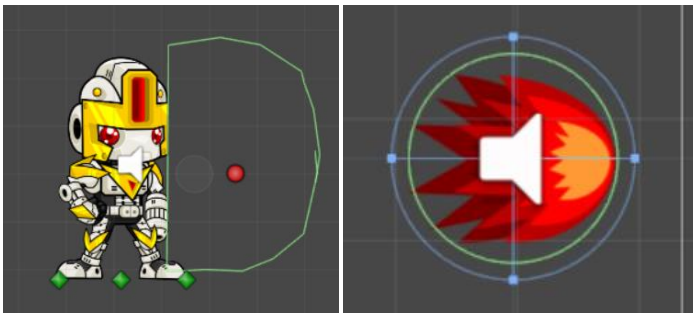
GRAPH 19. Codes for shooting in "PlayerController" script

Graph 19 shows a few simple lines of code to add the shooting function to the player character. When the player presses the "X" button on the keyboard, then an animation will be set to trigger and the "ShootingProjectile()" function will be executed from the "CharacterComponents" script. In line 236,if the current value of energy is less than 5,or if not immortal,then the player will not shoot. In scripting the "!" sign can be defined as not, for example "if(x !=5)" here, the if-statement will check for the condition if x is not equal to 5. The terms "energy" and "immortal" will be explained further,later in the thesis.

```
71      public virtual void ShootingProjectile()
72      {
73          if (facingRight)
74          {
75              GameObject temp = Instantiate(projectile, shootingPoint.transform.position, Quaternion.identity) as GameObject;
76              temp.GetComponent<ProjectileController>().Initialize(Vector2.right);
77          }
78          else
79          {
80              GameObject temp = Instantiate(projectile, shootingPoint.transform.position, Quaternion.Euler(new Vector3(0, 0, -180))) as GameObject;
81              temp.GetComponent<ProjectileController>().Initialize(Vector2.left);
82          }
83
84      }
85
```

GRAPH 20. ShootingProjectile function in CharacterComponents script

As line 71 in Graph 20 shows, the "ShootingProjectile()" function has virtual in its return type. When this function is called, a prefab GameObject will be instantiated. Just by dragging a GameObject on the Project window, it is possible to create a prefab. Prefab is an asset which copies the original GameObject and stores it for later use. To instantiate or to create clones of the prefab, we need a position in scene view and attach it to the player character. From this point onwards, whenever the "X" button is pressed, prefabs will appear. In the game, "projectile" is a GameObject variable which contains the prefab and "shootingPoint" is the point where prefabs instantiate. As velocity is added to the GameObject, whenever the function "ShootingProjectile" is called, instantiated objects will continue moving towards the left or right, depending on the direction the player character faces. The term "Quaternion" means rotation, where "Quaternion.identity" defines the default and "Quaternion.Euler(new Vector3(0,0,-180))" defines that the GameObject is to be rotated 180 degreeson the Z axis. (Unity Technologies 2017l, Unity Technologies 2017m.)



GRAPH 21. Sword collider and circle collider.

The purpose of this attack system is to eliminate other game characters, such as enemies. There is an edge-collider for the sword and a circle collider for the projectile prefab. The colliders are set to trigger and when they collide with an enemy, a certain amount of damage is dealt to their health.

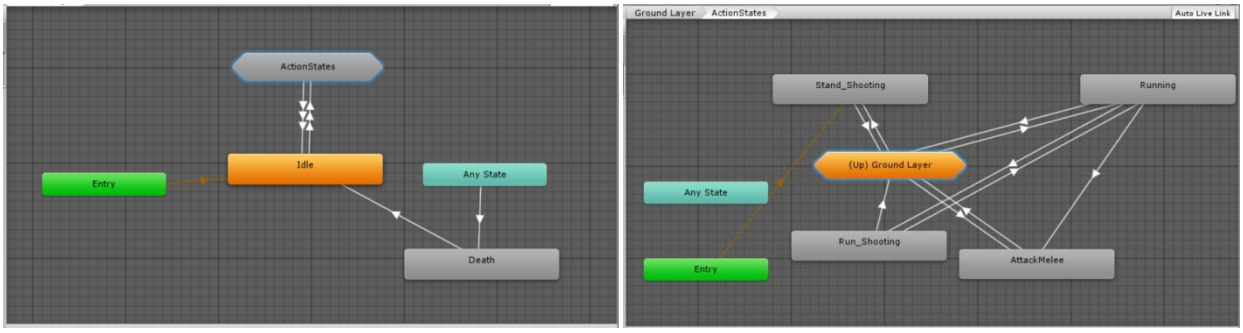
```
43          if(other.tag == "player_sword")
44          {
45              enemyHealth -= 15;
46
47          }
48
49
50          if(other.tag == "EnergyProjectile")
51          {
52              enemyHealth -= FindObjectOfType<ProjectileController>().dmg;
53              Instantiate(GameManager.Manager.BleedEffect, other.transform.position, other.transform.rotation);
54              Destroy(other.gameObject);
55
56          }
```

GRAPH 22. Enemyhealth decreases if collision occurs

In graph 22, colliders attached to the gameObjects triggers when collided with enemy's collider and decrease their health by a fixed amount.

4.3.4 Player Animations

Section 3.1.4 described how to create animations for a particular action. If the player character has more than one animation, we would need parameters to run those animations when various functions are called. To call the animator attached to GameObjects, it is required to store an animator in the script attached to it. To store an animator into a variable, "ThisAnimator = GetComponent<Animator>()" is included in the Start() function. For the game, I used animations like idling, running, jumping, attacking,shooting and a death animation.Graph 23shows two images, the player character's state machine and a sub-state machine named "ActionStates".



GRAPH 23. State Machine and Sub-State Machine

In the player character's state machine, the "Death" animation is linked to any state, meaning whenever the parameter "death" is set to trigger, it will animate. If the player character’s current health is below

or equal to 0, or it falls off-screen, the character is considered dead. Graph 24 shows a delegate function written before the main class so that it can be used by outside classes.

```
7   public delegate void EventHandler();
8   public class PlayerController : CharacterComponents
9   {
10
11      public event EventHandler dead;
```

GRAPH 24. Codes in "PlayerController" script

Delegates are a type of function pointers used to contain a reference to a method, and set parameters within, if the function or method has one. Line 8 shows that "PlayerController" is a derived class of the "CharacterComponents" class and in line 11, an event of type "EventHandler" delegate is declared.

```
156     public void OnDead()
157     {
158         if (dead != null)
159         {
160
161             dead();
162         }
163     }
```

GRAPH 25. "OnDead" function

Graph 25 shows a function that checks if the event is not empty and calls "dead" delegate as a function, whenever the "OnDead" function is called.

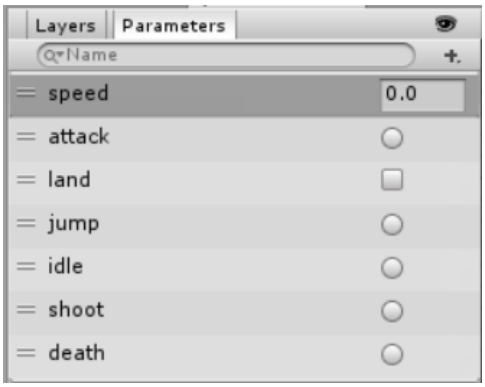
```
56      // Returns whether the Player is dead or not.
57      public override bool IsDead
58      {
59          get
60          {
61              if (health.CurrentValue <= 0)
62              {
63                  OnDead();
64              }
65
66              return health.CurrentValue <= 0;
67          }
68      }
```

GRAPH 26. "IsDead" bool type function is "PlayerController" script

In the graph above, "IsDead" is a read-only property that returns boolean value, if the player character's health is below or equal to zero. If the statement is true, then the "OnDead" function is called and the dead event is invoked. The main purpose of using an event is, that it lets Unity to execute functions asynchronously, which I will be showing in Non-Playable character section. The following Graph, 27 shows that if "IsDead" is true then death animation is set to trigger.

```
315             if (!IsDead)
316             {
317 
318                 immortal = true;
319                 StartCoroutine(WhileImmortal());
320                 yield return new WaitForSeconds(immortalDuration);
321                 immortal = false;
322             }
323             else
324             {
325 
326                 ThisAnimator.SetLayerWeight(1, 0);
327                 ThisAnimator.SetTrigger("death");
328 
329 
330             }
331         }
```

GRAPH 27. Codes in "PlayerController"



GRAPH 28. Animation parameters

In graph 28, "speed" is a float parameter that returns a value greater than zero. In graph 29 in line 179, the value of "GetAxis("Horizontal")" is set to "speed",and whenever the value returns a float number greater than zero, the run animation is animated. Other triggers like "attack", "shoot" and "jump" are only triggered when the predefined key or button is pressed.

```
179     ThisAnimator.SetFloat("speed",Mathf.Abs(Input.GetAxis("Horizontal")));
193     ThisAnimator.SetTrigger("jump");
217     ThisAnimator.SetTrigger("attack");
```

```
238     ThisAnimator.SetTrigger("shoot");
```

GRAPH 29. Different animations of the player character

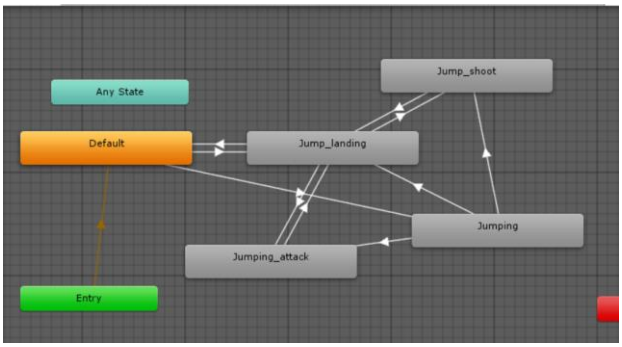
Graph 29line 179shows a code that checks the value and stores it into the "speed" parameter. "Mathf.Abs" is used to return a positive sign value, even if the value is negative. For example if the value is "-1", Mathf.Abs will return "1" as a value. It is also necessary to write the strings inside the brackets, same as the name of the parameter, else the animator will not be able to identify them.

As I mentioned before,there is a layer control in the Animator window. Layer control helps to control the character animations throughout different layers. To add a new layer, select the layer tab in the Animator window and click the small "+" sign. Suppose a player character is to shoot or attack while jumping in the air. Maintaining such actions through different motions is much easier with the layer control option.

```
250     private void AnimatorLayerHandler()
251     {
252         if (!Grounded)
253         {
254             ThisAnimator.SetLayerWeight(1, 1);
255         }
256         else
257             ThisAnimator.SetLayerWeight(1, 0);
258
259
260     }
```

GRAPH 30. Determines whether to change the layer on the animation

Graph 30 shows a function called "AnimatorLayerHandler()". This function helps to set the layer to either 0 or 1, 0 being the ground and 1 being the air. While jumping, the player character is able to shoot projectiles and carry out a melee attack, with animating ions. "SetLayerWeight" is a public method for Animator, which takes LayerIndex and Weight as parameters.



GRAPH 31. "AirLayer" in State Machine

When in air the "AnimatorLayerHandler" changes the layer, the transition then checks either for "shoot" or for "attack" triggers.

4.3.5 Player Health and Energy System

This section will be about the player character's health and energy levels. Health has a predefined value and decreases when the player character gets damaged by enemies. Whenever the value of health returns zero, as in graph 27, the player triggers the death animation. Energy also has a value which diminishes every time the player shoots a projectile, as seen in graph 19.

In Graph 19 line 238, "energy.CurrentValue>=5", explains that if the value of energy is greater than or equal to 5, the player character can perform the shoot action.

```
17      public float CurrentValue
18      {
19          get
20          {
21              return currentValue;
22          }
23
24          set
25          {
26             currentValue = Mathf.Clamp(value,0,MaxValue);
27              bar.Value = currentValue;
28          }
29      }
30
31      public float MaxValue
32      {
33          get
34          {
35              return maxValue;
36          }
37
38          set
39          {
40              maxValue = value;
41              bar.MaxValue = maxValue;
42          }
43      }
44
```

GRAPH 32. Getters and setters of max and current value

To set and get the values of the health and energy stats, we need two float properties, "maxValue" and "currentValue", as shown in graph 32.It can also be seen, that the "currentValue" property returns a float and the value is set to clamp between 0 and "maxValue". "Mathf.Clamp" is a static Mathf method, which restricts the value between minimum and maximum. For example, if the value is 100, and the minimum and maximum are 0 and 50, respectively, the Clamp method returns the value as 50, as it would exceed its maximum limit. Note that the "bar" term is used in the following graph, which will be explained in detail on the latter part of the thesis.

```
47      public void SetValues()
48      {
49          MaxValue = maxValue;
50          CurrentValue = currentValue;
51      }
52
```

GRAPH 33. Function for setting the values

Now that it is possible to get the values, we would require a function to determine and set them. In graph 33, the "SetValues()" function has a publicaccess modifier so that the values can be set from

outside the class.Both the player character and the enemieshave health stats and these are derived from its base class. Therefore, the "Stat" class is instantiated in the "CharacterComponents" scripts. To instantiate a class, set the type as class name and name a variable, as shown.In the line "private Stat statManager"private is an access modifier and "Stat" is the class type. As the stat variable is shared both by the derived classes, "protected" access is used in the scripts,so that it remains private to other classes but still accessible by its own derived class. After defining the variable, "variable name.SetValues()" is required to be called in the Start() function to set its values. When the scripting is done, to set the value of health, go to the Inspector view by selecting a GameObject, to which the "PlayerController" is attached, and change the max and current value under the health title.

```
49      public virtual  void Start()
50      {
51
52          facingRight = true;
53          ThisAnimator = GetComponent<Animator>();
54          health.SetValues();
55
56      }
57
```

GRAPH 34. Start function in "CharacterComponent" script

```
307     public override IEnumerator TakeDamage(float dmg)
308     {
309
310         if (!immortal)
311         {
312             health.CurrentValue -= dmg;
313
314
315             if (!IsDead)
316             {
317
318                 immortal = true;
319                 StartCoroutine(IfImmortal());
320                 yield return new WaitForSeconds(immortalDuration);
321                 immortal = false;
322             }
323             else
324             {
325
326                 ThisAnimator.SetLayerWeight(1, 0);
327                 ThisAnimator.SetTrigger("death");
328
329
330             }
331         }
```

GRAPH 35."TakeDamage" function in "PlayerController" script

After setting the health for the player character, a function is added to the "PlayerController" script. The script is called when the player takes damage from outside sources, whether these are attacks by enemies or traps and spikes on the ground. When a collision takes place, the "TakeDamage" function subtracts the damage amount from the player character's current health, as seen in graph 35, line 312.

IEnumerator is a return type that executes a function each frame of the game. It is required for the “WaitForSeconds”to process. “WaitForSeconds” is a class in the Unity Engine and inherits from “YieldInstruction” class. To suspend co-routine functions, “WaitForSeconds” is necessary. Co-routines usually call a function, and in any given moment the function can be suspended or stopped. To execute IEnumerators, “StartCoroutine” is used, as shown is graph 35, line 319. In this line, the “StartCoroutine” executes "IfImmortal" function and in the next line, after the “WaitForSeconds” time exceeds "ImmortalDuration" seconds, the co-rouine stops.As both “WaitForSeconds” and“Coroutines”are inherited from same base class, the "yield new" keyword is required.(Unity Technologies 2017o, Unity Technologies 2017p.)

In graph 35, the term "immortal" was used several times. It is a Boolean variable that it is used to determine whether the player character should be in invincible state or not. Once a player has taken damage, it sets the Boolean to true and for an estimated amount of seconds, the player will not take further damage. The co-routine ends when the Boolean returns false. I have also added a function that iterates, if the player is in this immortal state. This does not interfere with the performance, but makes the game more assertive in a visual sense. Several games that I have played had the same "immortal" function, which also made me include it in my thesis game.

```
337     private IEnumerator IfImmortal()
338     {
339         while (immortal)
340         {
341             playerRenderer.enabled = false;
342             yield return new WaitForSeconds(.1f);
343             playerRenderer.enabled = true;
344             yield return new WaitForSeconds(.1f);
345         }
```

GRAPH 36. "IfImmortal" function in "PlayerController" script

As seen from line 307 in graph 36, the function takes a float parameter, which is useful if the damage amount is received from an outside class.

Energy values are also set similarlyto health values. As it will only be used by the player character, I wrote them in "PlayerController" script.When the player shoots a projectile, the amount of energy decreases by a certain amount, as shown in graph 19, line 238.

GRAPH 37. Player health and energy bar in the game

Graph 37 shows how the player's health and energy are displayed in the game interface. For such purposes a "Bar" script was attached, which manipulates the fill bar. So when the player's health and energy is at max, the bar will be full. And if the player gets damaged, the value and the green bar decreases. The health and energy bars are sliders, which can be created by right-clicking on the Hierarchy, UI then sliders.UI elements will be further discussed in later sections.

```
294     //Adds health to Player.
295     public void GetHealth(float hp)
296     {
297
298         health.CurrentValue += hp;
299
300         if(health.CurrentValue >= health.MaxValue)
301         {
302             health.CurrentValue = health.MaxValue;
303         }
```

GRAPH 38. "GetHealth" function is "PlayerController" script

```
21      void OnTriggerEnter2D(Collider2D other)
22      {
23          if(other.tag == "Player")
24          {
25              PlayerController.PlayerInstance.GetHealth(giveHp);
26              sound.Play();
27              Destroy(gameObject);
28          }
```

GRAPH 39. Codes in "RefillHealth" script, which are attached to health pickup object in game

There are also some collectibles in the game. When picked up, these give a certain amount of health back to the player, if the current health is less than the maximum. Graph 38 shows a public "GetH-

ealth" function with a float parameter. When the function callsthe "RefillHealth" script, which is attached to a GameObject, health is added to the player as it can be seen in Graph 39. Energy pickup was also scripted using a similar method.

4.4 Non-Playable Characters

Non-playable characters (NPCs) in games are the characters not controlled by the players. Instead,these characters have preset action patterns and behaviours, which are executed when certain conditions are met. In this game NPCs are the enemies who want to prevent the player from reaching its destination. There are three different types of NPCs in the game, with different movement and attack behaviours.

GRAPH 40. NPCs used in the game

Graph 40 shows the three non-playable characters, enemies of the player. Though there is only one image of the "Archer" character, but in-game I have used two other archers with increased health and damage output.

4.4.1 Movements and Animations

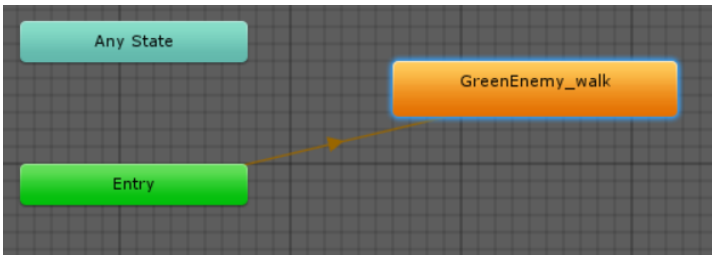
Each characterhas different animations and movements. The "Green" is the basic enemy with simple movements from one point to another. They have colliders attached to them and these are set as triggers. Whenever the player character collides with it, the player receives a certain amount of damage to

its health. Other than colliders, Rigidbody2D and an animator are also attached to it. It also has two GameObjects as children, which check if the character is on the ground or near an edge.

```
29      void FixedUpdate () {
30
31          wallHit = Physics2D.OverlapCircle(wallCheck.position, detectRad, whatIsWall);
32
33          atGround = Physics2D.OverlapCircle(edgeDetect.position, detectRad, whatIsWall);
34
35          if (wallHit || !atGround)
36          {
37              moveRight = !moveRight;
38          }
39
40          if (moveRight)
41          {
42              transform.localScale = new Vector3(1.5f, 1.5f, 1f);
43              enemy.velocity = new Vector2(moveSpeed, enemy.velocity.y);
44          }
45
46          else
47          {
48              transform.localScale = new Vector3(-1.5f, 1.5f, 1f);
49              enemy.velocity = new Vector2(-moveSpeed, enemy.velocity.y);
50
51          }
52
```

GRAPH 41. FixedUpdate function in "BasicEnemyController"

The Boolean variables "wallHit" and "atGround" in Graph 41, line 31, check if the circle in Vector3 positions are overlapping with the LayerMask variable, which is "whatIsWall". Here "whatIsWall" is set to the ground layer in the game. So if it is overlapping, then the variables return true. In line 35, if the statement states, that if the character is colliding with the wall or is not overlapping with the ground layer, then "moveRight", which is a Boolean variable, is not "moveRight". It means if "moveRight" were true, it would return false, and vice versa. In line 40 the if-else statement dictates the displacement of the character. In lines 42 and 48"transform.localscale" determines whether the character should face towards the left or the right, and lines 43 and 49 define the velocity with which the character will move. Here "enemy" is the Rigidbody component and "moveSpeed" is a float variable. The character has a single walk animation and it is set to default.



GRAPH 42. State Machine of "Green" enemy characters.

The movements and animations of "Archer" characters are different from "Green" characters. It has several animations attached to it and the movement pattern is random. There are two states in its movements, patrol state and idle state. To make the game more realistic and alive, "Archer" characters patrol for a certain time and then stay idle, as if they were looking for the player character for the next couple of seconds. The float numbers are generated randomly between a minimum and a maximum number. Such charactershave four children GameObjects attached. The health canvas, line of sight collider, position in the world from where the arrows or projectiles are instantiated and an edge collider for swords. I will further discuss about the health canvas in section 4.5. The line of sight collider is used to detect if the player character is within the enemy's range. While in collision with the line of sight collider, it would set the player as the enemy's target and it would attack the player.

```
5   public class EnemyLOSight : MonoBehaviour {
6       [SerializeField]
7       private EnemyController enemy;
8
9       void OnTriggerEnter2D(Collider2D other)
10      {
11          if(other.tag == "Player")
12          {
13              enemy.Target = other.gameObject;
14          }
15
16
17      }
18
19      void OnTriggerExit2D(Collider2D other)
20      {
21          if(other.tag == "Player")
22          {
23              enemy.Target = null;
24          }
25      }
26
27
```

GRAPH 43. "EnemyLOSight" script

The "OnTriggerEnter2D()" function in the graph above only executes if the collider is set to trigger."OnTriggerEnter2D()" is a built-in method that Unity uses when detecting collision.The term SerializeField allows private variables to be accessed from the Unity Engine and makes it possible to change the values.In line 7, an EnemyController type object is created, so that the "Target" instance can be set from this script. Lines 11 and 21 check if the collided GameObject's tag is "Player" or not. If this returns true, then the player is set as the target and if the player leaves the collision zone, then the target is set to null.

Although "Archer" characters are equipped with colliders, they are unable to perform melee attacks. The reason is, that the sprites were downloaded from the Asset Store, and melee attack animation was not available for this particular character. Thus, I have changed the tag to "OnlyRange" and added an if-statement that executes only if the characters' tag name is "OnlyRange". Also, there are two transform variables, which contain world positions. These are used so that the characters can move from one point to another, without falling off platforms.

```
5   public interface IEnemyState
6
7   {
8       void Execute();
9       void Enter(EnemyController enemy);
10
```

GRAPH 44. "IEnemyState" interface

For "Archer" and "Ninja" characters, coding was done in several scripts. The main script "EnemyController", derived from "CharacterComponent" class, is only attached to the character to handle its movements and stats. There is an interface script called "IEnemyState", which has four classes implementing from it. The four states are patrol, idling, shooting and melee attacking. Graph 44 shows an interface class "IEnemyState", containing two different functions.

The function "Enter" (See Graph 44, line 9) has "EnemyController" as a parameter, and whenever an enemy enters this state, "Enter" function is called and initializes codes from the "Execute" function. Every state should be implemented from the "IEnemyState", and must contain all the functions that are present in the interface. In each state script, a variable of type "EnemyController" is assigned and set equal to enemy parameter in "Enter" function.

```
22      public void Execute()
23      {
24          Idle();
25
26          if(enemyC.Target != null)
27          {
28              enemyC.ChangeState(new PatrolState());
29          }
30      }
31
32      public void Idle()
33      {
34          enemyC.ThisAnimator.SetFloat("speed", 0);
35
36          idleTimer += Time.deltaTime;
37
38          if(idleTimer >= idleDuration)
39          {
40              enemyC.ChangeState(new PatrolState());
41          }
42
```

GRAPH 45. Codes from "IdleState" script

In Graph 45 line 22, the "Execute" function calls the "Idle" function, where the character stays idle for a random duration of time and changes its state to "PatrolState". Note that "enemyC" is an "Enemy-Controller" type variable and is called by instantiating. In line 26, "Target" is preferred to the player character that is set when it collides with the enemy's line of sight collider as shown in graph 43, line 13.

```
20      public void Execute()
21      {
22          Patrol();
23          enemyC.Move();
24
25          if (enemyC.Target != null && enemyC.ShootRange)
26          {
27              enemyC.ChangeState(new RangedState());
28          }
29      }
30
31  public void Patrol()
32      {
33
34
35          patrolTimer += Time.deltaTime;
36
37          if (patrolTimer >= patrolDuration)
38          {
39              enemyC.ChangeState(new IdleState());
40          }
41
42      }
43
```

GRAPH 46. Codes from "PartolState" script

In patrol statethrough the "Patrol" function the enemy is set to move for a randomised amount of time before going back to idle state. However, when the line of sight detects the player, then it is set to change its state to range.

```
16          patrolDuration = UnityEngine.Random.Range(1, 10);
```

GRAPH 47. For generating a random number within range

Graph 47 shows a line that is used to generate a random number within a minimum and maximum number. Here the minimum number is inclusive and the maximum number is exclusive.

This concludes the movements of the enemies. The next section will continue with the attack and health systems of the enemy characters.

4.4.2 Attack and Health System

When the player is in sight, and if the enemy character is tagged as "OnlyRange", then it would maintain its distance and shoot arrows from afar. But if the character was a "Ninja" it would slowly approach the player character's position. While a "Ninja" character moves towards the player, it would continue throwing its shuriken stars until the gap is reduced to its defined melee range. When at melee range, "Ninjas" will animate sword strikes dealing damage to the player character. To determine the melee and range distance, float variables were used and also propertiesto read the distance between the player and the enemy.

```
24      public bool MeleeRange
25      {
26          get
27          {
28              if (Target != null)
29              {
30                  return Vector2.Distance(transform.position, Target.transform.position) <= meleeRange;
31              }
32
33              return false;
34          }
35
36
37
38      }
39      public bool ShootRange
40      {
41          get
42          {
43              if (Target != null)
44              {
45                  return Vector2.Distance(transform.position, Target.transform.position) <= shootRange;
46              }
47
48              return false;
49          }
50
```

GRAPH 48. Read-only properties in "EnemyController" script

In graph 48 the distance is calculated using “Vector2.Distance” method, which takes two parameters; the "transform.position" is the position of the GameObject that the script isattached to and the "Target.transform.position" is the position of the target, which is, in this case, the player character.

```
112     public void ChangeState(IEnemyState newState)
113     {
114         currentState = newState;
115         currentState.Enter(this);
116     }
```

GRAPH 49. "ChangeState" function in "EnemyController" script

In the "EnemyController" script, a variable of "IEnemyState" needs to be called by the different states.To change to a different state, the "ChangeStates" function is executred. It takes "IEnemyStates" as a pararmeter. Graph 49above shows when the function is called, it changes its current state to the new state in the parameter and calls the “Enter” function from the changed state.

```
87      private void RemoveTarget()
88      {
89          Target = null;
90          ChangeState(new PatrolState());
91      }
92
93
94      private void FaceTheTarget()
95      {
96          if(Target != null)
97          {
98
99              float xAxis = Target.transform.position.x - transform.position.x;
100
101             if(xAxis < 0 && facingRight || xAxis > 0 && !facingRight)
102             {
103                 changingDirection();
104
105             }
106
```

GRAPH 50. Functions for facing or removing the target in "EnemyController" script

In graph 50, the "FaceTheTarget" function checks if the target is not null and changes its direction according to the player character's position. In line 87, the function removes the target and changes its state to patrol.

```
243     public override void ShootingProjectile()
244     {
245         if (facingRight)
246         {
247             GameObject temp = (GameObject)Instantiate(projectile, shootingPoint.transform.position, Quaternion.identity);
248             temp.GetComponent<EnemyProjectileController>().Initialize(Vector2.right);
249         }
250         else
251         {
252             GameObject temp = (GameObject)Instantiate(projectile, shootingPoint.transform.position, Quaternion.Euler(new Vector3(0, 0, -180)));
253             temp.GetComponent<EnemyProjectileController>().Initialize(Vector2.left);
254         }
```

GRAPH 51. "ShootingProjectile" function in "EnemyController" script

As graph 51 shows, codes from the "EnemyController" script, "ShootingProjectile" function is not called from its base class function. As the enemy character has a separate projectile prefab, I wrote its code to a new script, "EnemyProjectileController". Though both the player's and the enemy's projectile scripts are similar, there are minor differences.

The enemy’s health system is quite similar to the player's health as both classes are derived from the "CharacterComponents" class. Instead of showing the health bar at the top of the screen like that of the player's, the enemy’s health bar hovers over its head and moves with it.

GRAPH 52. Enemy with its health bar

As the health bar is a child GameObject of the enemy, whenever the character changes its direction, the bar would also change with it. To maintain the position of the bar and to keep it from flipping, a few lines of code were written in the "EnemyController".

```
220     public override void changingDirection()
221     {
222
223         if(gameObject.tag == "ninjaboss")
224         {
225             base.changingDirection();
226         }
227         else
228         {
229             Transform temp = transform.FindChild("EnemyHealth Canvas").transform;
230             Vector3 currentpos = temp.position;
231             temp.SetParent(null);
232
233             base.changingDirection();
234
235             temp.SetParent(transform);
236             temp.position = currentpos;
237
238         }
239
```

Graph 53. "ChangingDirection" function in "EnemyController"

In graph 53, line 223, an if-else statement allows the game to recognize whether the character is an "Archer" or a "Ninja" enemy. This dictates if the current GameObject, to which the script is attached, is tagged "Ninjaboss" it would function as it is written in its base class. Otherwise, the script would try

to find a child object with string name "EnemyHealth Canvas" and set its parent to none. While doing this, the bar stays as it was before, without flipping with the character. After the function "changeDirection" is executed from the base class, the health bar's parent is set to enemy's transform position. This function only happens when the enemy character is about to flip from the left to the right, or vice versa.

GRAPH 54. Ninja character's health bar

In case of a "Ninja" character, the health bar is set in the main UI similar to the player character's health bar, as shown is Graph 54.

The health and attack system of "Ninja" and "Archer" enemies was a bit more complex compared to "Green" enemies. "Green" enemies are easier to kill and they do not have specific attacking methods. Instead, the colliders are set to trigger and damage the player if it comes in contact with the "Green" character.

```
20      // Update is called once per frame
21      void Update()
22      {
23
24          if (enemyHealth <= 0)
25          {
26
27              Instantiate(GameManager.Manager.CoinPrefab, transform.position, transform.rotation);
28              Destroy(gameObject);
29              sound.Play();
30          }
31      }
32
```

GRAPH 55. Update function in "BasicEnemyHealth" script

In the "BasicEnemyHealth" script, the health of the character is set by a float variable "enemyHealth", as shown in Graph 55, line 24. When the health reaches 0, a GameObject called "CoinPrefab" is instantiatedat the position where the enemy was destroyed. In line 29, a sound is played when the character is removed from the scene.

```
34      void OnTriggerEnter2D(Collider2D other)
35      {
36
37
38          if(other.tag == "player_sword")
39          {
40              enemyHealth -= 15;
41
42          }
43
44
45          if(other.tag == "EnergyProjectile")
46          {
47              enemyHealth -= FindObjectOfType<ProjectileController>().dmg;
48              Instantiate(GameManager.Manager.BleedEffect, other.transform.position, other.transform.rotation);
49              Destroy(other.gameObject);
50
51          }
52
53         if(other.tag == "Player")
54      {
55              StartCoroutine(PlayerController.PlayerInstance.TakeDamage(giveDmg));
56
57              if (PlayerController.PlayerInstance.IsDead)
58                  PlayerController.PlayerInstance.PlayerRigidbody.velocity = Vector2.zero;
59          }
60
61
```

GRAPH 56. "OnTriggerEnter2D" function in "BasicEnemyHealth" script

In Graph 56, when the character collides with "player_sword" or "EnergyProjectile", a certain amount is deducted from the "enemyHealth" variable. If it collides with the player, it starts a co-routine "TakeDamage" and gives a certain amount of damage to the player, which is set in the parameter.

```
12      private static PlayerController playerInstance;
13      public static PlayerController PlayerInstance
14      {
15          get
16          {
17              if (playerInstance == null)
18              {
19                  playerInstance = GameObject.FindObjectOfType<PlayerController>();
20
21              }
22              return playerInstance;
23          }
24      }
```

GRAPH 57. Code from the "PlayerController" script

In Graph 57, "PlayerInstance" is a read-only property that gets the GameObject that has a "PlayerController" script attached to it. Both the property and the variable are static, so they can be called by other scripts. This is useful, since it is not required to call the “FindObjectOfType” function every time other scripts call "PlayerController".

4.5 UI Canvas

The Canvas is a rectangular space in Unity's Scene view, where all the UI elements are placed. UI elements include images, text, sliders, buttons, input fields. In the Hierarchy window, Canvas is a GameObject and UI elements should be children of the canvas in order to process them. Whenever a Canvas is created, an EventSystem is also created at the same time, as the Canvas uses it for messaging the system. Basically, the Canvas makes the game more interactive, as it displays various texts, can show the player's health, points and many more. To create a canvas, simply right-click on the Hierarchy window then select UI > Canvas.

In this game, I used Canvas for displaying the character's health and the total number of points collected throughout the stage. (Unity Technologies 2017q.)

GRAPH 58. In Game UI Canvas

In graph 58, on the top left corner I have used a slider, an UI element, for displaying the player character's health and energy. There are also several texts, which are mostly informative. The number "0" beside the "coin" image and on the bottom left corner are counters. These numbers increase everytime the player picks up "coins" or dies either by falling off the platform or by an enemy character. For the character to pass each stage, a certain number of "coins" need to be collected, otherwise a text will pop up as a reminder, as seen in the picture.

To manipulate the texts and sliders, different scripts are attached to them. For the sliders, codes were written in the "Bar" script. Counting the number of "coins" happens in the "GameManager" script. In the "Bar" script, several variables were needed so that the bar's fill colour would change as the respective value decreases.

```
6   public class Bar : MonoBehaviour {
7       [SerializeField]
8       private float barSpeed;
9
10      private float fill;
11      [SerializeField]
12      private Image content;
13      [SerializeField]
14      private Text barText;
15
16      public float MaxValue { get; set; }
17
18      public float Value
19      {
20          set
21          {
22              string[] temp = barText.text.Split(':');
23              barText.text = temp[0] + ": " + value;
24              fill = FillMeter(value,MaxValue);
25          }
```

GRAPH 59. Variables in the "Bar" script

A number of variables used to set the health and energy bars can be seen in Graph 59. In line 16, "MaxValue" has get and set properties, so that its value can be stored from different classes. Referring back to Graph 32, lines 27 and 41 show that the float "Value" from Graph 59 line 18, is set to "currentValue" and "MaxValue" from the "Bar" script, and set to "maxValue" in the "Stat" script. "Value" has a write-only property, and it sets the current health of the characters. In line 22, an array variable of type string is set, which stores the strings before ":" character is located in the text. Then it adds the value of current health and displays it along the strings stored in "temp".

```
30      // Update is called once per frame
31      void Update () {
32          BarHandler();
33      }
34
35      private void BarHandler()
36      {
37          if(fill != content.fillAmount)
38          {
39              content.fillAmount = Mathf.Lerp(content.fillAmount, fill, Time.deltaTime * barSpeed);
40          }
41
42
43      }
44
45      private float FillMeter(float value,float maxValue)
46      {
47
48          return value / maxValue;
49      }
```

GRAPH 60. Update function in the "Bar" script

There are two other functions in Graph 60, apart from the update function, "BarHandler" and "FillMeter". The "BarHandler" function is mainly for smoothing purpose, when the colour in the bar increases or decreases. In line 45, "FillMeter" contains two float variables, value and maxHP. Here "value" is the current value and "maxValue" is the maximum value of the character's health or energy. The function has a float return type and basically returns a value between 0 and 1. For example, if the value equals 60 and the value of maxValue is 240, when calculated it returns 0.25. Since the minimum and maximum values of the fill amount is pre-set as 0 and 1, respectively, 0.25 amounts to 25% or one-fourth of the slider.The “fillAmount” is the value found under the slider component, in the Inspector window.In graph 59 line 24, the value of "fill" is set as the value of "FillMeter". In "BarHandler", the if-statement checks if the values of both "fill" and the image's fill amountare the same or not. If the values are different then it sets the fill amount value to that of the "fill" value, either by increasing or by decreasing it. In line 39, Mathf.Lerp helps to interpolate from one value to another in respect of time, multiplied to a float variable.

```
55      public int TotalPoints
56      {
57          get
58          {
59              return totalPoints;
60          }
61
62          set
63          {
64              pointsTxt.text = "x" + value.ToString();
65              totalPoints = value;
66          }
67      }
68
```

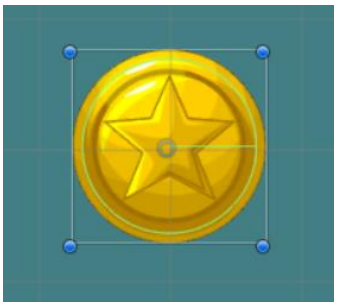
GRAPH 61. TotalPoints propertyin the "GameManager" script

In graph 61, the public property sets the "pointsTxt", which is a text-type variable, and changes its value to that of the value of “totalPoints”. In the "PlayerController" script, when the player character collides with the "coin" gameObjects, then it increments the value of "totalPoints" by 1.

```
373     //Interactions with other colliders.
374     public override void OnTriggerEnter2D(Collider2D other)
375     {
376
377         base.OnTriggerEnter2D(other);
378
379
380         if (other.gameObject.tag == "coins")
381         {
382             GameManager.Manager.TotalPoints++;
383             GameManager.Manager.sound.Play();
384             Destroy(other.gameObject);
385         }
386
387
388     }
389
```

GRAPH 62. "OnTriggerEnter2D" in the "PlayerController" script

Graph 62 shows a part of the "PlayerController" script regarding the increment of "totalPoints".After the collision, points increase, an audio file is played and the "coin" object is destroyed from the scene.



GRAPH 63. "Coin" GameObject used for the game.

The UI canvas also contains buttons, which can be interacted with in certain scenarios. In the game, buttons were used so that player can execute various functions just by clicking on them, for example to start a new game or to exit the game. There are different scenes in the game, like the main menu and the pause menu where buttons are used. While in-game, if the player presses a predefined key then a canvas will be enabled and the game will be paused for that duration. The purpose of this menu is to give the player flexibility and freedom while playing the game.

GRAPH 64. Pause menu

For the buttons to function properly, it is required to create a script and attach it to the pause menu canvas or to a GameObject. A script was attached to the canvas so that it can be enabled and disabled and to execute functions by clicking on the buttons. After adding buttons to the canvas, it is required to add a new list to the button from the Inspector window. A new list can be added simply by clicking (+) on the “OnClick()” box. After adding the list, simply drag and drop the GameObject that has the script attached to it. Now all the public functions that are in the script can be selected, so that it can be executed on the click of a button.

```
15      // Update is called once per frame
16      void Update () {
17
18          if (isPause)
19          {
20              pauseMenuCanvas.SetActive(true);
21              Time.timeScale = 0f;
22          }       else
23          {
24              pauseMenuCanvas.SetActive(false);
25              Time.timeScale = 1f;
26          }
27         if (Input.GetKeyDown(KeyCode.Escape))
28          {
29              isPause = !isPause;
30          }
31      }
32       public void Resume()
33       {
34           isPause = false;
35       }
36      public void MainMenu()
37       {
38           SceneManager.LoadScene(mainMenu);
39       }
40       public void MuteMusic()
41       {
42           music.mute = !music.mute;
```

GRAPH 65. "PauseMenu" script

Graph 65 shows the script attached to the pause menu. Several variables are used, such as "isPause", a Boolean, "pauseMenuCanvas", a GameObject type variable, "mainMenu",a stringthat stores the name of the scene and "music", an AudioSource type variable for accessing audio attached to an object. In line 27, when the keycode "Escape" is pressed on the keyboard, then the value of the "isPause" Boolean changes. If it is true, the GameObject stored in the "pauseMenuCanvas" variable will become active and the “timeScale” will be set to 0. The “timeScale” is a method of class Time, which determines the overall speed of the game, 1 being normal speed and 0 being fully stopped. The functions "Resume", "MainMenu" and "MuteMusic" are set to different buttons in the pause menu for desirable outcomes. Audio and loading scenes will be explained in later sections.

4.6 Level Design

Creating and designing levels took most of the time while making the game for this thesis. It is about personal opinion and creativeness how the author wants the game to look like. Apart from appearance,

some extra features were added to the game that can be seen quite often in modern games.These aspects might not be important for the development of a game, but it helped me learn more about it while experimenting with some features. For example, changing scenes, adding audio files, and creating checkpoints.

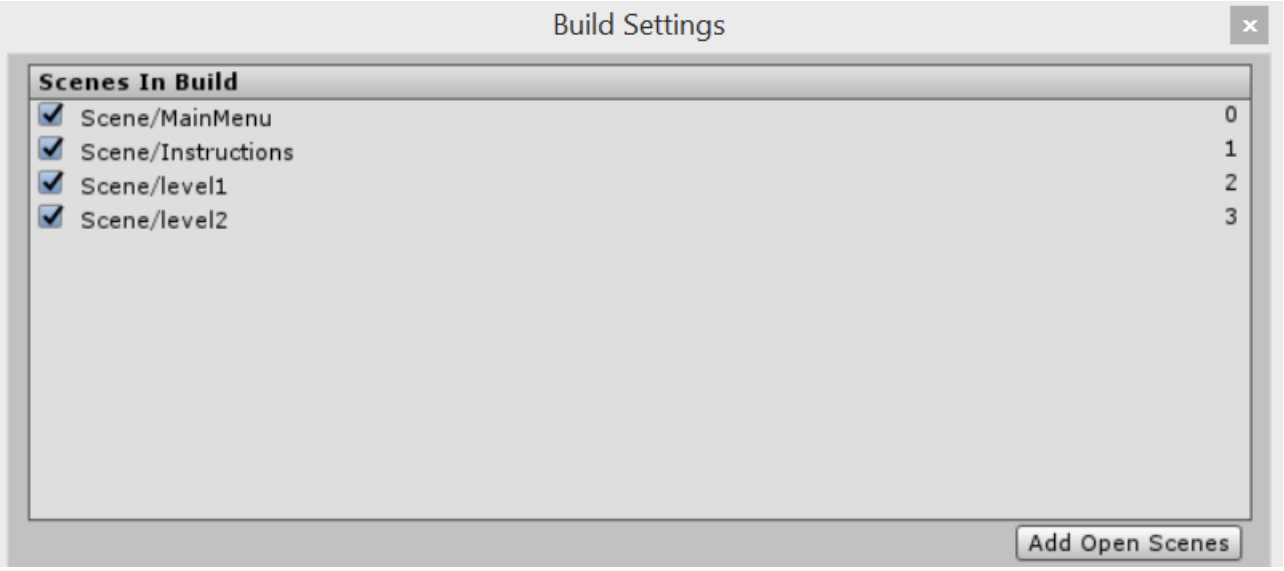
4.6.1 Loading New Scenes

The Loading scene is an important part for the development of a game, though it is certainly not compulsory. If a game consists of small levels, then there is no need for such, however, if the game is large, then it might be easier for the computer to process smaller levels at a time, rather than the whole game at the same time. It is necessary to understand, that the more GameObjects and textures are present in a scene, the time required to load is proportional to it. Therefore, it is a sensible choice to add scenes for faster loading times. To change from one scene to another, it is required to create a script that is attached to a GameObject.

```
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4   using UnityEngine.SceneManagement;
5
6   public class LoadNewScene : MonoBehaviour {
7
8       void Start()
9       {
10          SceneManager.LoadScene("newScene");
11
12      }
```

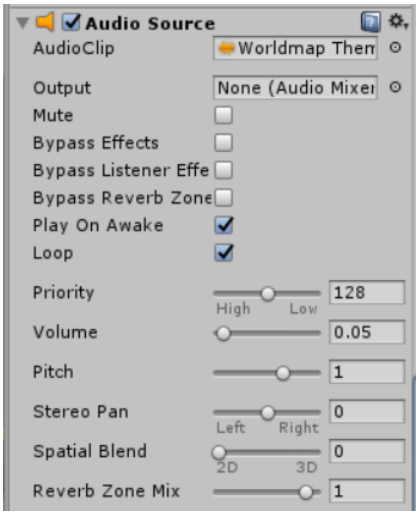
GRAPH 66. Codes to load a new scene

Graph 66 shows a simple script with a few lines of code. For the "SceneManager" class to execute, it is required to write "using UnityEngine.SceneManagement;" in the namespace. "LoadScene" is a function in the "SceneManager" that takes a string as a scene name, and an int variable for the scene index. In the graph "newScene" is the name of the scene that is to be loaded. Before loading the new scene, it is required to add scenes in the build settings. To open the build settings tab, go to File > Build Settings. In the settings window, press “Add Open Scenes” to insert the scene that is currently open in the Scene view, as seen in Graph 67.



GRAPH 67. Build Settings windowwith numberson the right indicating the index of the scene

4.6.2 Audio



GRAPH 68. Audio Source Component

There are various audio files used in the game for different purposes, for example, while shooting, jumping, collecting coins. To attach audio files to GameObjects, simply drag and drop a file in the Inspector window while selecting the GameObject. The attached audio files can be seen in the Audio Source Component, shown in Graph 68. There are various options in Audio source, such as play on awake, which then plays audio as soon as the game starts, or loop, to play an audio file continuously. To play from the script, simply make a variable of AudioSource type, and call the play() function.

4.6.3 Checkpoints and Exit signs

There are several checkpoint signs throughout the levels and an exit sign at the end of each level. Checkpoints are used so when the player is removed, either because of dying or falling off platformsfrom the scene, the player can be respawned and be placed in a point in the game world. When the player character passes or collides with a checkpoint flag, then the position is stored in a variable. This variable is rewritten every time the player reaches a new checkpoint.

```
14      void OnTriggerEnter2D(Collider2D other)
15      {
16
17          if (other.name == "Player")
18          {
19              PlayerController.PlayerInstance.playerSpawn = new Vector2(transform.position.x, transform.position.y);
20              Debug.Log("Activated Checkpoint " + transform.position);
21              Destroy(gameObject);
22              sound.Play();
23              checkpoint.GetComponent<Animator>().SetBool("ifplayer", true);
24
25          }
```

GRAPH 69. Storing Vector2 positions in “playerSpawn” variable in the "Checkpoint" script

Graph 69 shows when the player triggers a collision with the checkpoint, it saves a new vector2 position to the "playerSpawn" variable, which is in the "PlayerController" script. Graph 70 shows codes for respawning the player in the position saved in "playerSpawn".

```
354     public override void Death()
355     {
356         GameManager.Manager.TimesDied++;
357         PlayerRigidbody.velocity = Vector2.zero;
358         ThisAnimator.SetTrigger("idle");
359         health.CurrentValue = health.MaxValue;
360         transform.position = playerSpawn;
361         playerRespawn = true;
362         Instantiate(spawnParticle, transform.position, transform.rotation);
363         deathCounter = 0;
364
365     }
```

GRAPH 70. In line 360 the player is respawned to the "playerSpawn" position.

Exit signs are used to progress to the next stage. When the requirements in each stage are met and the player reaches a certain point, the next level is loaded. The scripts are written in "LoadNewLevel" and attached to the exit GameObject sign.

```
42      // Update is called once per frame
43      void Update()
44      {
45
46          if (playerInZone == true && currentScene.name != "level3")
47          {
48              if (GameManager.Manager.TotalPoints >= coinsNeeded)
49              {
50
51                  SceneManager.LoadScene(levelToLoad);
52              }
53              else
54              {
55                  txt.text = "You need to collect " + (coinsNeeded - GameManager.Manager.TotalPoints) + " more coins";
56
57
58              }
59          }
60
61          if(GameManager.Manager.TotalPoints == coinsNeeded)
62          {
63              txt.text = "You may proceed to exit";
64
65
66          }
67
68          if(currentScene.name == "level3")
69          {
70             if(boss == null)
71              {
72                  txt.text = "GO TO EXIT";
73                  door.GetComponent<Animator>().SetBool("ifplayer", true);
74              }
75             if(playerInZone)
76              {
77                  SceneManager.LoadScene(levelToLoad);
78              }
79
80          }
81
```

GRAPH 71. Update function in the "LoadNewLevel" script

```
84      void OnTriggerEnter2D(Collider2D other)
85      {
86          if (other.name == "Player")
87          {
88
89              playerInZone = true;
90
91          }
92
93      }
94
95      void OnTriggerExit2D(Collider2D other)
96      {
97          if (other.name == "Player")
98          {
99
100             playerInZone = false;
101             txt.text = "";
102
103         }
104
105     }
```

GRAPH 72. OnTrigger functions in the "LoadNewLevel" script

As Graph 72 reveals, both OnTrigger functions check if the player is in contact with the collider or not. When the player enters the collider, the "playerInZone" Boolean is set to true and, when the player leaves, the Boolean is set to false. In the Update function in line 44, Graph 71the if-statement checks whether the player is in the zone and the current scene name is not "level3". If it is true, then it checks whether the points in "GameManager" script are equal to the value of "coinsNeeded". If this condition returns true, then a new scene is loaded, otherwise a text is enabled, informing the player about coins required to pass the stage.

4.7 Building the Game

For the purposes of testing the full game as a separate application, it is required to build it first. In Unity it is quite simple to build a game, just by clicking build and run options in the Filetab. All the necessary scenes should be inserted in Scenes in the Build windowbefore building the game, as shown in Graph 67. Through this window it is possible to build the game for different platforms and systems, such as PC, iOS or Android. The icon and the name of the application can also be changed from the player settings, in the bottom left corner of the window. After all the scenes are inserted, simply click build and run. After building the program is done, anyone can play the game that has been created.

5 CONCLUSION

In this thesis, I explained about the game development and programming with the Unity game engine along with the various features that make developing games easier. Developing games is a quite difficult and lengthy process without proper planning and execution .Working on the thesis not only helped me acquire knowledge about the Unity game engine and game development, but also about programming. As I was able to gain in-depth ideas about the game making and programming which I would not have come through during normal day to day practice. The game developed was for learning purposes and to introduce the latest features in Unity2D. This project will be developed further and improved in the future.

With a suitable game engine and a bit of knowledge about programming, it is possible to create games. Even though there are many game engines, each having its own perks and features. At the end it all comes down to individual choices, on which game engine to work on as every game engine serves for the same purpose. Over the last few years, Unity has developed a lot and now includes many more features for 2D game development.The Unity game engine has seen significant changes and with its growing popularity more developers have chosen it over other engines, mainly because of its simpler user interface.

To conclude, an overall idea about a modern day, two-dimensional game has been presented in this thesis. It is important to acquire knowledge in hopes of creating more complicated and wonderful games, but in a simple way.

REFERENCES

Alexsandr. 2014. Documentation, Unity scripting languages and you.Available: https://blogs.unity3d.com/2014/09/03/documentation-unity-scripting-languages-and-you/. Accessed 10 October 2017.

Corazza, S. 2013. History of the Unity Engine Freerunner 3D Animation Project. Available: https://seraphinacorazza.wordpress.com/2013/02/14/history-of-the-unity-engine-freerunner-3d-animation-project/. Accessed 15 September 2017.

Downie, C. 2016. Evolution of our Products and Pricing. Available: https://blogs.unity3d.com/2016/06/16/evolution-of-our-products-and-pricing/. Accessed 14 September 2017.

Unity Technologies.2017a. Learning the Interface. Available: https://docs.unity3d.com/Manual/LearningtheInterface.html. Accessed 14 September 2017.

Unity Technologies.2017b. Project view.Available: https://docs.unity3d.com/Manual/ProjectView.html. Accessed 14 September 2017.

Unity Technologies.2017c. Scene View.Available: https://docs.unity3d.com/Manual/UsingTheSceneView.html. Accessed 14 September 2017.

Unity Technologies.2017d. Game View.Available: https://docs.unity3d.com/Manual/GameView.html. Accessed 14 September 2017.

Unity Technologies.2017e. Hierarchy.Available: https://docs.unity3d.com/Manual/Hierarchy.html. Accessed 14 September 2017.

Unity Technologies. 2017f. Inspector Window.Available: https://docs.unity3d.com/Manual/UsingTheInspector.html. Accessed 14 September 2017.

Unity Technologies. 2017g. Toolbar. Available:https://docs.unity3d.com/Manual/Toolbar.html. Accessed 14 September 2017.

Unity Technologies. 2017h. Sprite Editor. Available:https://docs.unity3d.com/Manual/SpriteEditor.html. Accessed 10 October 2017.

Unity Technologies. 2017i. Sprite Creator. Available: https://docs.unity3d.com/Manual/SpriteCreator.html. Accessed 10 October 2017.

Unity Technologies. 2017j. MonoBehaviour LateUpdate. Available: https://docs.unity3d.com/ScriptReference/MonoBehaviour.LateUpdate.html. Accessed 10 October 2017.

Unity Technologies. 2017k. MonoBehaviour FixedUpdate. Available: https://docs.unity3d.com/ScriptReference/MonoBehaviour.FixedUpdate.html. Accessed 10 October 2017.

Unity Technologies. 2017l. Prefab. Available: https://docs.unity3d.com/Manual/Prefabs.html. Accessed 10 October 2017.

Unity Technologies. 2017m. Quaternion. Available: https://docs.unity3d.com/ScriptReference/Quaternion.html. Accessed 18 October 2017.

Unity Technologies. 2017n. Mathf. Available: https://docs.unity3d.com/ScriptReference/Mathf.html. Accessed 18 October 2017.

Unity Technologies. 2017o. WaitForSeconds. Available: https://docs.unity3d.com/ScriptReference/WaitForSeconds.html. Accessed 18 October 2017.

Unity Technologies. 2017p. Coroutines. Available: https://docs.unity3d.com/Manual/Coroutines.html. Accessed 18 October 2017.

Unity Technologies. 2017q. UICanvas. Available: https://docs.unity3d.com/Manual/UICanvas.html. Accessed 18 October 2017.

Unity Technologies. 2017r. Sprites. Available: https://docs.unity3d.com/Manual/Sprites.html. Accessed 10 October 2017.

Unity Technologies. 2017s. Sprite Renderer. Available: https://docs.unity3d.com/Manual/class-SpriteRenderer.html. Accessed 10 October 2017.

Unity Technologies. 2017u. RigidBody2D. Available: https://docs.unity3d.com/Manual/class-Rigidbody2D.html. Accessed 11 October 2017.

Unity Technologies. 2017v. Collider2D. Available: https://docs.unity3d.com/Manual/Collider2D.html. Accessed 11 October 2017.

Unity Technologies. 2017w. Physics Material 2D. Available: https://docs.unity3d.com/Manual/class-PhysicsMaterial2D.html. Accessed 11 October 2017.

Unity Technologies. 2017x. Joints2D. Available: https://docs.unity3d.com/Manual/Joints2D.html. Accessed 11 October 2017.

Unity Technologies. 2017y. Effectors2D. Available: https://docs.unity3d.com/Manual/Effectors2D.html. Accessed 11 October 2017.

Game Scripts

Bar.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class Bar : MonoBehaviour {
   [SerializeField]
   private float barSpeed;

   private float fill;
   [SerializeField]
   private Image content;
   [SerializeField]
   private Text barText;

   public float MaxValue { get; set; }

   public float Value
   {
      set
      {
         string[] temp = barText.text.Split(':');
         barText.text = temp[0] + ": " + value;
         fill = FillMeter(value,MaxValue);
      }


   }

               // Update is called once per frame
               void Update () {
      BarHandler();
               }

   private void BarHandler()
   {
      if(fill != content.fillAmount)
      {
         content.fillAmount = Mathf.Lerp(content.fillAmount, fill, Time.deltaTime * barSpeed);
      }


   }

   private float FillMeter(float value,float maxValue)
```

```
    {

        return value / maxValue;
    }




}
```

BasicEnemyController.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class BasicEnemyController : MonoBehaviour {
    [SerializeField]
    private float moveSpeed;
    private bool moveRight;

    public Transform wallCheck;
    public float detectRad;
    public LayerMask whatIsWall;
    private bool wallHit;
    private bool atGround;
    public Transform edgeDetect;

    private Rigidbody2D enemy;


    // Use this for initialization
    void Start () {
        enemy = GetComponent<Rigidbody2D>();



    }

                // Update is called once per frame
                void FixedUpdate () {

        wallHit = Physics2D.OverlapCircle(wallCheck.position, detectRad, whatIsWall);

        atGround = Physics2D.OverlapCircle(edgeDetect.position, detectRad, whatIsWall);

        if (wallHit || !atGround)
        {
```

```
            moveRight = !moveRight;
        }

        if (moveRight)
        {
            transform.localScale = new Vector3(1.5f, 1.5f, 1f);
            enemy.velocity = new Vector2(moveSpeed, enemy.velocity.y);
        }

        else
        {
            transform.localScale = new Vector3(-1.5f, 1.5f, 1f);
            enemy.velocity = new Vector2(-moveSpeed, enemy.velocity.y);

        }

    }




}
```

BasicEnemyHealth.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class BasicEnemyHealth : MonoBehaviour {
    [SerializeField]
    private float enemyHealth;


    public float giveDmg;

    public AudioSource sound;


                // Use this for initialization
                void Start () {

                }

    // Update is called once per frame
    void Update()
    {

        if (enemyHealth <= 0)
        {
```

```
            Instantiate(GameManager.Manager.CoinPrefab, transform.position, transform.rotation);
            Destroy(gameObject);
            sound.Play();
        }
    }


    void OnTriggerEnter2D(Collider2D other)
    {


        if(other.tag == "player_sword")
        {
            enemyHealth -= 15;

        }


        if(other.tag == "EnergyProjectile")
        {
            enemyHealth -= FindObjectOfType<ProjectileController>().dmg;
            Instantiate(GameManager.Manager.BleedEffect, other.transform.position, oth-
er.transform.rotation);
            Destroy(other.gameObject);

        }

       if(other.tag == "Player")
    {
            StartCoroutine(PlayerController.PlayerInstance.TakeDamage(giveDmg));

            if (PlayerController.PlayerInstance.IsDead)
                PlayerController.PlayerInstance.PlayerRigidbody.velocity = Vector2.zero;
        }


    }




}
```

CameraFollow.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
```

```
public class CameraFollow : MonoBehaviour {
    [SerializeField]
    private float xMax;
    [SerializeField]
    private float xMin;
    [SerializeField]
    private float yMax;
    [SerializeField]
    private float yMin;

    private Transform player;

    // Use this for initialization
    void Start () {

        player = GameObject.Find("Player").transform;

                }

    // LateUpdate is called after the player is moved
    void LateUpdate()
    {
        transform.position = new Vector3(Mathf.Clamp(player.position.x, xMin, xMax),
            Mathf.Clamp(player.position.y, yMin, yMax),
            transform.position.z);
    }
}
```

CharacterComponents.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public abstract class CharacterComponents : MonoBehaviour
{
    public Animator ThisAnimator { get; private set; }

    [SerializeField]
    protected Transform shootingPoint;
    [SerializeField]
    private EdgeCollider2D swordCollider;
    [SerializeField]
    protected GameObject projectile;
    public bool Attack { get; set; }
```

```
    [SerializeField]
    protected float moveSpeed;
    protected bool facingRight;

    [SerializeField]
    protected Stat health;



    public abstract bool IsDead { get;  }



    public EdgeCollider2D SwordCollider
    {
        get
        {
            return swordCollider;
        }


    }

    // Use this for initialization
    public virtual  void Start()
    {

        facingRight = true;
        ThisAnimator = GetComponent<Animator>();
        health.SetValues();

    }

    // Update is called once per frame
    void Update()
    {

    }

    public virtual void changingDirection()
    {
        facingRight = !facingRight;

        transform.localScale = new Vector3(transform.localScale.x * -1, transform.localScale.y, trans-
form.localScale.z);
    }
```

```
    public virtual void ShootingProjectile()
    {
        if (facingRight)
        {
            GameObject temp = Instantiate(projectile, shootingPoint.transform.position, Quaterni-
on.identity) as GameObject;
            temp.GetComponent<ProjectileController>().Initialize(Vector2.right);
        }
        else
        {
            GameObject temp = Instantiate(projectile, shootingPoint.transform.position, Quaterni-
on.Euler(new Vector3(0, 0, -180))) as GameObject;
            temp.GetComponent<ProjectileController>().Initialize(Vector2.left);
        }

    }




    public abstract IEnumerator TakeDamage(float dmg);

   public abstract void Death();






    public void MeleeAttack()
    {
        SwordCollider.enabled = true;
    }






    public virtual void OnTriggerEnter2D(Collider2D other)
    {




    }
```

```
}
```

Checkpoints.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Checkpoints : MonoBehaviour {



    [SerializeField]
    private GameObject checkpoint;

    public AudioSource sound;

    void OnTriggerEnter2D(Collider2D other)
    {

        if (other.name == "Player")
        {
            PlayerController.PlayerInstance.playerSpawn = new Vector2(transform.position.x, trans-
form.position.y);
            Debug.Log("Activated Checkpoint " + transform.position);
            Destroy(gameObject);
            sound.Play();
            checkpoint.GetComponent<Animator>().SetBool("ifplayer", true);

        }


    }




}
```

EnemyController.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class EnemyController : CharacterComponents
{
```

```
private IEnemyState currentState;

public GameObject Target { get; set; }

public float meleeRange;

public float shootRange;
[SerializeField]
private Transform edgeRight;
[SerializeField]
private Transform edgeLeft;
private bool dropItem = true;

private Canvas healthCanvas;

public bool MeleeRange
{
   get
   {
      if (Target != null)
      {
         return Vector2.Distance(transform.position, Target.transform.position) <= meleeRange;
      }

      return false;
   }



}
public bool ShootRange
{
   get
   {
      if (Target != null)
      {
         return Vector2.Distance(transform.position, Target.transform.position) <= shootRange;
      }

      return false;
   }



}

public override bool IsDead
{
   get
   {
```

```
        return health.CurrentValue <= 0;

    }
}



// Use this for initialization
public override void Start () {
    base.Start();
    PlayerController.PlayerInstance.dead += new EventHandler(RemoveTarget);
    ChangeState(new IdleState());
    healthCanvas = transform.GetComponentInChildren<Canvas>();



}

// Update is called once per frame
void Update()
{
    if (!IsDead)
    {
        currentState.Execute();
        FaceTheTarget();
    }
}

private void RemoveTarget()
{
    Target = null;
    ChangeState(new PatrolState());
}


private void FaceTheTarget()
{
    if(Target != null)
    {

        float xAxis = Target.transform.position.x - transform.position.x;

        if(xAxis < 0 && facingRight || xAxis > 0 && !facingRight)
        {
            changingDirection();

        }

    }
```

```
    }

    public void ChangeState(IEnemyState newState)
    {
        currentState = newState;
        currentState.Enter(this);
    }




    public void Move()
    {

        if (!Attack)

        {
            if((GetDirection().x > 0 && transform.position.x < edgeRight.position.x)  || (GetDirection().x <
0 && transform.position.x > edgeLeft.position.x))
            {
                ThisAnimator.SetFloat("speed", 1);

                transform.Translate(GetDirection() * moveSpeed * Time.deltaTime);

            }
            else if(currentState is PatrolState)
            {
                changingDirection();
            }

        }
    }

    public Vector2 GetDirection()
    {
        return facingRight ? Vector2.right : Vector2.left;
    }

    public override void OnTriggerEnter2D(Collider2D other)
    {
        base.OnTriggerEnter2D(other);
        if (other.tag == "EnergyProjectile")
        {
         if(Target == null && gameObject.tag != "ninjaboss")
            {
                changingDirection();
            }
            StartCoroutine(TakeDamage(FindObjectOfType<ProjectileController>().dmg));
            Destroy(other.gameObject);
```

```
        Instantiate(GameManager.Manager.BleedEffect, other.transform.position, oth-
er.transform.rotation);
    }

    if(other.tag == "player_sword")
    {
        StartCoroutine(TakeDamage(20));
        Instantiate(GameManager.Manager.BleedEffect, other.transform.position, oth-
er.transform.rotation);
    }









}

public override IEnumerator TakeDamage(float dmg)
{
    if(gameObject.tag != "ninjaboss")
    {
        if (!healthCanvas.isActiveAndEnabled)
        {
            healthCanvas.enabled = true;
        }

    }


    health.CurrentValue -= dmg;


    if(IsDead || health.CurrentValue <= 0)
    {

        ThisAnimator.SetTrigger("death");
        if(gameObject.tag != "ninjaboss")
        {
            healthCanvas.enabled = false;
        }




        if (dropItem)
        {
            if(gameObject.tag != "ninjaboss")
            {
```

```
                Instantiate(GameManager.Manager.CoinPrefab, new Vector3(transform.position.x, transform.position.y - 1), Quaternion.identity);

                dropItem = false;
            }


        }


    }

    yield return null;
}

public override void Death()
{

    Destroy(gameObject);
}

public override void changingDirection()
{

    if(gameObject.tag == "ninjaboss")
    {
        base.changingDirection();
    }
    else
    {
        Transform temp = transform.FindChild("EnemyHealth Canvas").transform;
        Vector3 currentpos = temp.position;
        temp.SetParent(null);

        base.changingDirection();

        temp.SetParent(transform);
        temp.position = currentpos;

    }


}

public override void ShootingProjectile()
{
    if (facingRight)
    {
        GameObject temp = (GameObject)Instantiate(projectile, shootingPoint.transform.position, Quaternion.identity);
```

```
            temp.GetComponent<EnemyProjectileController>().Initialize(Vector2.right);
        }
        else
        {
            GameObject temp = (GameObject)Instantiate(projectile, shootingPoint.transform.position,
Quaternion.Euler(new Vector3(0, 0, -180)));
            temp.GetComponent<EnemyProjectileController>().Initialize(Vector2.left);
        }

    }



}
```

EnemyLOSight.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class EnemyLOSight : MonoBehaviour {
    [SerializeField]
    private EnemyController enemy;

                    void OnTriggerEnter2D(Collider2D other)
    {
        if(other.tag == "Player")
        {
            enemy.Target = other.gameObject;
        }


    }

    void OnTriggerExit2D(Collider2D other)
    {
        if(other.tag == "Player")
        {
            enemy.Target = null;
        }
    }


}
```

EnemyProjectileController.cs

```
using System.Collections;
using System.Collections.Generic;
```

```
using UnityEngine;

public class EnemyProjectileController : MonoBehaviour {



    public float speed;

    private Rigidbody2D thisRigidBody;

    private Vector2 direction;

    public float dmg;







    // Use this for initialization
    void Start()
    {
        thisRigidBody = GetComponent<Rigidbody2D>();

    }

    // Update is called once per frame
    void Update()
    {


    }

    void FixedUpdate()
    {
        thisRigidBody.velocity = direction * speed;

        if (gameObject.tag == "shuriken")
        {
            gameObject.transform.Rotate(Vector3.forward * 45);
        }


    }
```

```
    public void Initialize(Vector2 d)
    {
        this.direction = d;
    }

    void OnBecameInvisible()
    {
        Destroy(gameObject);
    }

    void OnTriggerEnter2D(Collider2D other)
    {
      if(other.tag == "Player")
        {
            StartCoroutine(PlayerController.PlayerInstance.TakeDamage(dmg));
        }

    }

}
```

GameManager.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using UnityEngine.SceneManagement;


public class GameManager : MonoBehaviour {

    private static GameManager manager;
    [SerializeField]
    private GameObject coinPrefab;
    [SerializeField]
    private GameObject bleedEffect;
    [SerializeField]
    private Text pointsTxt;
    [SerializeField]
    private Text triesLeftTxt;



    public AudioSource sound0;
    public AudioSource sound1;


    private int totalPoints;
    private int timesDied;
```

```
public GameObject tryAgainMenu;

public static GameManager Manager
{
   get
   {
      if(manager == null)
      {
         manager = FindObjectOfType<GameManager>();
      }
      return manager;
   }


}

public GameObject CoinPrefab
{
   get
   {
      return coinPrefab;

   }



}

public int TotalPoints
{
   get
   {
      return totalPoints;
   }

   set
   {
      pointsTxt.text = "x" + value.ToString();
      totalPoints = value;
   }
}

public GameObject BleedEffect
{
   get
   {
      return bleedEffect;
   }
```

```
    }

    public int TimesDied
    {
        get
        {
            return timesDied;
        }

        set
        {
            triesLeftTxt.text = "" + value.ToString();
            timesDied = value;
        }
    }




    // Use this for initialization
    void Start () {
        Time.timeScale = 1;

    }

                // Update is called once per frame
                void Update () {

        if (GameManager.Manager.TimesDied >= 5)
        {
            tryAgainMenu.SetActive(true);
        }



    }

}
```

LoadNewLevel.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;
using UnityEngine.UI;

public class LoadNewLevel : MonoBehaviour {
```

```
    private bool playerInZone;
    [SerializeField]
    private string levelToLoad;
    [SerializeField]
    private Text txt;
    public Scene currentScene;

    public GameObject boss;
    public GameObject door;



    public int coinsNeeded;



    // Use this for initialization
    void Start () {
        playerInZone = false;
        txt.text = "";


         currentScene = SceneManager.GetActiveScene();


    }


    // Update is called once per frame
    void Update()
    {

        if (playerInZone == true && currentScene.name != "level3")
        {
            if (GameManager.Manager.TotalPoints >= coinsNeeded)
            {

                SceneManager.LoadScene(levelToLoad);
            }
            else
            {
                txt.text = "You need to collect " + (coinsNeeded - GameManager.Manager.TotalPoints) + "
more coins";
```

```
        }
    }

    if(GameManager.Manager.TotalPoints == coinsNeeded && currentScene.name != "level3")
    {
        txt.text  = "You may proceed to exit";


    }

   if(currentScene.name == "level3")
    {
      if(boss == null)
       {
          txt.text = "GO TO EXIT";
          door.GetComponent<Animator>().SetBool("ifplayer", true);
          if (playerInZone)
          {
             SceneManager.LoadScene(levelToLoad);
          }
       }


    }

}

void OnTriggerEnter2D(Collider2D other)
{
    if (other.name == "Player")
    {

        playerInZone = true;

    }

}

void OnTriggerExit2D(Collider2D other)
{
    if (other.name == "Player")
    {

        playerInZone = false;
        txt.text = "";

    }

}
```

```
}
```

MainMenu.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;

public class MainMenu : MonoBehaviour {

    public string startGame;

    public string instructions;



    public void NewGame()
    {
        SceneManager.LoadScene(startGame);

    }
    public void Instruct()
    {
        SceneManager.LoadScene(instructions);
    }

    public void Quit()
    {
        Application.Quit();

    }
}
```

PauseMenu.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;

public class PauseMenu : MonoBehaviour {

    public string mainMenu;
    public AudioSource music;
    private bool isPause;

    [SerializeField]
    private GameObject pauseMenuCanvas;
```

```
            // Update is called once per frame
            void Update () {

        if (isPause)
        {
            pauseMenuCanvas.SetActive(true);
            Time.timeScale = 0f;
        }       else
        {
            pauseMenuCanvas.SetActive(false);
            Time.timeScale = 1f;
        }
       if (Input.GetKeyDown(KeyCode.Escape))
        {
            isPause = !isPause;
        }
    }
     public void Resume()
     {
        isPause = false;
     }
    public void MainMenu()
     {
        SceneManager.LoadScene(mainMenu);
     }
     public void MuteMusic()
     {
        music.mute = !music.mute;


     }

}
```

PlayerController.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;


public delegate void EventHandler();
public class PlayerController : CharacterComponents
{

    public event EventHandler dead;
    private static PlayerController playerInstance;
    public static PlayerController PlayerInstance
    {
```

```
    get
    {
        if (playerInstance == null)
        {
            playerInstance = GameObject.FindObjectOfType<PlayerController>();

        }
        return playerInstance;
    }
}



public Stat energy;



[SerializeField]
private float jumpHeight;
[SerializeField]
private float groundPointRadius;
[SerializeField]
private LayerMask whatIsGround;
[SerializeField]
private Transform[] groundPoints;

private bool immortal = false;
[SerializeField]
private float immortalDuration;

private float deathtimer = 1.5f;
private float deathCounter;

public Rigidbody2D PlayerRigidbody { get; set; }

private SpriteRenderer playerRenderer;


private bool doubleJump;
public bool Grounded { get; set; }

// Returns whether the Player is dead or not.
public override bool IsDead
{
    get
    {
        if (health.CurrentValue <= 0)
        {
            OnDead();
        }
```

```
            return health.CurrentValue <= 0;
        }
    }

    // Returns whether Player's y position is less than zero or not.
    public bool isFalling
    {
        get
        {
            return PlayerRigidbody.velocity.y < 0;
        }
    }

    public Vector2 playerSpawn;
    public bool playerRespawn = false;

    [SerializeField]
    private GameObject spawnParticle;

    // Use this for initialization
    public override void Start()
    {
        base.Start();
        PlayerRigidbody = GetComponent<Rigidbody2D>();
        playerRenderer = GetComponent<SpriteRenderer>();

        // Initializing energy values
        energy.SetValues();

        // Respawning Player
        playerSpawn = transform.position;
        Vector3 temp = new Vector3(transform.position.x, transform.position.y -1.5f , trans-
form.position.z);
        Instantiate(spawnParticle, temp, transform.rotation);



    }

    // Update is called once per frame
    void Update()
    {

        if (!IsDead)
        {
            // Falling off screen
            if (transform.position.y <= -20f)
            {
```

```
            // Velocity is set to zero
            PlayerRigidbody.velocity = Vector2.zero;

            deathCounter += Time.deltaTime;
            if(deathCounter >= deathtimer)
            {
               Death();

            }



         }

      }

      if (Grounded)
         doubleJump = false;

      InputHandler();



   }


   void FixedUpdate()
   {
      if (!IsDead)
      {
         PlayerMovements();
         TurnPlayer();
         Grounded = IsGround();
         AnimatorLayerHandler();


      }




   }

   public void OnDead()
   {
      if (dead != null)
      {

         dead();
      }
```

```
    }


    // For Moving the character
    private void PlayerMovements()
    {
        if (isFalling)
        {

            ThisAnimator.SetBool("land", true);
        }
        //Moves Player in X axis with constant movespeed.
        if (!Attack)
        {

            PlayerRigidbody.velocity = new Vector2(Input.GetAxis("Horizontal") * moveSpeed, Play-
erRigidbody.velocity.y);
            ThisAnimator.SetFloat("speed",Mathf.Abs(Input.GetAxis("Horizontal")));
        }
        // Adds velocity to Player's y axis.




        //Allowing the Player to jump twice.
        if(Input.GetKeyDown(KeyCode.Space) && !Grounded && !doubleJump)
        {
            PlayerRigidbody.velocity = new Vector2(PlayerRigidbody.velocity.x, jumpHeight);
            GetComponent<AudioSource>().Play();

            ThisAnimator.SetTrigger("jump");
            doubleJump = true;
        }




    }


    // Flipping the character
    private void TurnPlayer()
    {
        if (Input.GetAxis("Horizontal") < 0 && facingRight || Input.GetAxis("Horizontal") > 0 && !fac-
ingRight)
        {
            changingDirection();
        }
```

```
}
//Different action inputs.
private void InputHandler()
{
    if (Input.GetKeyDown(KeyCode.Z) && !immortal)
    {
        ThisAnimator.SetTrigger("attack");
    }


    if (Input.GetKeyDown(KeyCode.Space) && Grounded)
    {
        PlayerRigidbody.velocity = new Vector2(PlayerRigidbody.velocity.x, jumpHeight);
        //Plays audio attached to Player.
        GetComponent<AudioSource>().Play();
        //Sets the trigger for jump in animator state machine.
        ThisAnimator.SetTrigger("jump");
    }


    // Energy decreases on every use.
    if(energy.CurrentValue>=5 || !immortal)
    {
        if (Input.GetKeyDown(KeyCode.X))
        {

            energy.CurrentValue -= 5;
            ThisAnimator.SetTrigger("shoot");
            ShootingProjectile();

        }
    }







}
// Checks if the Player is in ground level or air.
private void AnimatorLayerHandler()
{
    if (!Grounded)
    {
        ThisAnimator.SetLayerWeight(1, 1);
    }
    else
        ThisAnimator.SetLayerWeight(1, 0);
```

```
    }
    // Multiple points used to check if Player is in ground.
    private bool IsGround()
    {

        if (PlayerRigidbody.velocity.y <= 0)
        {
            foreach (Transform point in groundPoints)
            {
                Collider2D[] colliders = Physics2D.OverlapCircleAll(point.position, groundPointRadius,
whatIsGround);

                for (int i = 0; i < colliders.Length; i++)
                {
                    if (colliders[i].gameObject != gameObject)
                    {
                        return true;
                    }
                }
            }
        }

        return false;

    }
    //Overriding function in CharacterComponents.
    public override void ShootingProjectile()
    {
        base.ShootingProjectile();


    }
    //Enables and disables Player's Sprite Renderer to create flashing effect.

    //Adds health to Player.
    public void GetHealth(float hp)
    {

        health.CurrentValue += hp;

        if(health.CurrentValue >= health.MaxValue)
        {
            health.CurrentValue = health.MaxValue;
        }

    }
    private IEnumerator IfImmortal()
    {
        while (immortal)
        {
```

```
        playerRenderer.enabled = false;
        yield return new WaitForSeconds(.1f);
        playerRenderer.enabled = true;
        yield return new WaitForSeconds(.1f);

    }



}

public override IEnumerator TakeDamage(float dmg)
{

    if (!immortal)
    {
        health.CurrentValue -= dmg;


        if (!IsDead)
        {

            immortal = true;
            StartCoroutine(IfImmortal());
            yield return new WaitForSeconds(immortalDuration);
            StopCoroutine(IfImmortal());
            immortal = false;
        }
        else
        {

            ThisAnimator.SetLayerWeight(1, 0);
            ThisAnimator.SetTrigger("death");


        }
    }



}



public override void Death()
{
    GameManager.Manager.TimesDied++;
    PlayerRigidbody.velocity = Vector2.zero;
    ThisAnimator.SetTrigger("idle");
    health.CurrentValue = health.MaxValue;
```

```
        transform.position = playerSpawn;
        energy.CurrentValue = energy.MaxValue;
        playerRespawn = true;
        Vector3 temp = new Vector3(transform.position.x, transform.position.y - 1.5f , trans-
form.position.z);
        Instantiate(spawnParticle, temp, transform.rotation);
        deathCounter = 0;

    }


    //Interactions with other colliders.
    public override void OnTriggerEnter2D(Collider2D other)
    {

        base.OnTriggerEnter2D(other);


        if (other.gameObject.tag == "coins")
        {
            GameManager.Manager.TotalPoints++;
            GameManager.Manager.sound0.Play();
            Destroy(other.gameObject);
        }

        if (other.gameObject.tag == "pickups")
        {
            GameManager.Manager.sound1.Play();

        }

    }

}
```

ProjectileController.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;


public delegate void EventHandler();
public class PlayerController : CharacterComponents
{

    public event EventHandler dead;
    private static PlayerController playerInstance;
```

```
public static PlayerController PlayerInstance
{
   get
   {
      if (playerInstance == null)
      {
         playerInstance = GameObject.FindObjectOfType<PlayerController>();

      }
      return playerInstance;
   }
}



public Stat energy;



[SerializeField]
private float jumpHeight;
[SerializeField]
private float groundPointRadius;
[SerializeField]
private LayerMask whatIsGround;
[SerializeField]
private Transform[] groundPoints;

private bool immortal = false;
[SerializeField]
private float immortalDuration;

private float deathtimer = 1.5f;
private float deathCounter;

public Rigidbody2D PlayerRigidbody { get; set; }

private SpriteRenderer playerRenderer;


private bool doubleJump;
public bool Grounded { get; set; }

// Returns whether the Player is dead or not.
public override bool IsDead
{
   get
   {
      if (health.CurrentValue <= 0)
      {
```

```
            OnDead();
        }

        return health.CurrentValue <= 0;
    }
}

// Returns whether Player's y position is less than zero or not.
public bool isFalling
{
    get
    {
        return PlayerRigidbody.velocity.y < 0;
    }
}

public Vector2 playerSpawn;
public bool playerRespawn = false;

[SerializeField]
private GameObject spawnParticle;

// Use this for initialization
public override void Start()
{
    base.Start();
    PlayerRigidbody = GetComponent<Rigidbody2D>();
    playerRenderer = GetComponent<SpriteRenderer>();

    // Initializing energy values
    energy.SetValues();

    // Respawning Player
    playerSpawn = transform.position;
    Vector3 temp = new Vector3(transform.position.x, transform.position.y -1.5f , trans-
form.position.z);
    Instantiate(spawnParticle, temp, transform.rotation);



}

// Update is called once per frame
void Update()
{

    if (!IsDead)
    {
        // Falling off screen
        if (transform.position.y <= -20f)
```

```
        {

            // Velocity is set to zero
            PlayerRigidbody.velocity = Vector2.zero;

            deathCounter += Time.deltaTime;
            if(deathCounter >= deathtimer)
            {
                Death();

            }



        }

    }

    if (Grounded)
        doubleJump = false;

    InputHandler();



}


void FixedUpdate()
{
    if (!IsDead)
    {
        PlayerMovements();
        TurnPlayer();
        Grounded = IsGround();
        AnimatorLayerHandler();


    }




}

public void OnDead()
{
    if (dead != null)
    {
```

```
            dead();
        }
    }


    // For Moving the character
    private void PlayerMovements()
    {
        if (isFalling)
        {

            ThisAnimator.SetBool("land", true);
        }
        //Moves Player in X axis with constant movespeed.
        if (!Attack)
        {

            PlayerRigidbody.velocity = new Vector2(Input.GetAxis("Horizontal") * moveSpeed, Play-
erRigidbody.velocity.y);
            ThisAnimator.SetFloat("speed",Mathf.Abs(Input.GetAxis("Horizontal")));
        }
        // Adds velocity to Player's y axis.




        //Allowing the Player to jump twice.
        if(Input.GetKeyDown(KeyCode.Space) && !Grounded && !doubleJump)
        {
            PlayerRigidbody.velocity = new Vector2(PlayerRigidbody.velocity.x, jumpHeight);
            GetComponent<AudioSource>().Play();

            ThisAnimator.SetTrigger("jump");
            doubleJump = true;
        }




    }


    // Flipping the character
    private void TurnPlayer()
    {
        if (Input.GetAxis("Horizontal") < 0 && facingRight || Input.GetAxis("Horizontal") > 0 && !fac-
ingRight)
        {
```

```
            changingDirection();
        }
    }
    //Different action inputs.
    private void InputHandler()
    {
        if (Input.GetKeyDown(KeyCode.Z) && !immortal)
        {
            ThisAnimator.SetTrigger("attack");
        }


        if (Input.GetKeyDown(KeyCode.Space) && Grounded)
        {
            PlayerRigidbody.velocity = new Vector2(PlayerRigidbody.velocity.x, jumpHeight);
            //Plays audio attached to Player.
            GetComponent<AudioSource>().Play();
            //Sets the trigger for jump in animator state machine.
            ThisAnimator.SetTrigger("jump");
        }


        // Energy decreases on every use.
        if(energy.CurrentValue>=5 || !immortal)
        {
            if (Input.GetKeyDown(KeyCode.X))
            {

                energy.CurrentValue -= 5;
                ThisAnimator.SetTrigger("shoot");
                ShootingProjectile();

            }
        }







    }
    // Checks if the Player is in ground level or air.
    private void AnimatorLayerHandler()
    {
        if (!Grounded)
        {
            ThisAnimator.SetLayerWeight(1, 1);
        }
        else
            ThisAnimator.SetLayerWeight(1, 0);
```

```
    }
    // Multiple points used to check if Player is in ground.
    private bool IsGround()
    {

        if (PlayerRigidbody.velocity.y <= 0)
        {
            foreach (Transform point in groundPoints)
            {
                Collider2D[] colliders = Physics2D.OverlapCircleAll(point.position, groundPointRadius,
whatIsGround);

                for (int i = 0; i < colliders.Length; i++)
                {
                    if (colliders[i].gameObject != gameObject)
                    {
                        return true;
                    }
                }
            }
        }

        return false;

    }
    //Overriding function in CharacterComponents.
    public override void ShootingProjectile()
    {
        base.ShootingProjectile();


    }
    //Enables and disables Player's Sprite Renderer to create flashing effect.

    //Adds health to Player.
    public void GetHealth(float hp)
    {

        health.CurrentValue += hp;

        if(health.CurrentValue >= health.MaxValue)
        {
            health.CurrentValue = health.MaxValue;
        }

    }
    private IEnumerator IfImmortal()
    {
```

```
    while (immortal)
    {
        playerRenderer.enabled = false;
        yield return new WaitForSeconds(.1f);
        playerRenderer.enabled = true;
        yield return new WaitForSeconds(.1f);

    }



}

public override IEnumerator TakeDamage(float dmg)
{

    if (!immortal)
    {
        health.CurrentValue -= dmg;


        if (!IsDead)
        {

            immortal = true;
            StartCoroutine(IfImmortal());
            yield return new WaitForSeconds(immortalDuration);
            StopCoroutine(IfImmortal());
            immortal = false;
        }
        else
        {

            ThisAnimator.SetLayerWeight(1, 0);
            ThisAnimator.SetTrigger("death");


        }
    }



}



public override void Death()
{
    GameManager.Manager.TimesDied++;
    PlayerRigidbody.velocity = Vector2.zero;
```

```
        ThisAnimator.SetTrigger("idle");
        health.CurrentValue = health.MaxValue;
        transform.position = playerSpawn;
        energy.CurrentValue = energy.MaxValue;
        playerRespawn = true;
        Vector3 temp = new Vector3(transform.position.x, transform.position.y - 1.5f , trans-
form.position.z);
        Instantiate(spawnParticle, temp, transform.rotation);
        deathCounter = 0;

    }







    //Interactions with other colliders.
    public override void OnTriggerEnter2D(Collider2D other)
    {

        base.OnTriggerEnter2D(other);


        if (other.gameObject.tag == "coins")
        {
            GameManager.Manager.TotalPoints++;
            GameManager.Manager.sound0.Play();
            Destroy(other.gameObject);
        }

        if (other.gameObject.tag == "pickups")
        {
            GameManager.Manager.sound1.Play();

        }

    }


}
```

Stat.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
```

```
[Serializable]
public class Stat

{
    [SerializeField]
    private Bar bar;
    [SerializeField]
    private float maxValue;
    [SerializeField]
    private float currentValue;

    public float CurrentValue
    {
        get
        {
            return currentValue;
        }

        set
        {
           currentValue = Mathf.Clamp(value,0,MaxValue);
            bar.Value = currentValue;
        }
    }

    public float MaxValue
    {
        get
        {
            return maxValue;
        }

        set
        {
            maxValue = value;
            bar.MaxValue = maxValue;
        }
    }



    public void SetValues()
    {
        MaxValue = maxValue;
        CurrentValue = currentValue;
    }
```

```
}
```

RefillHealth.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class RefillHealth : MonoBehaviour {

    public float giveHp;



    void OnTriggerEnter2D(Collider2D other)
    {
        if(other.tag == "Player")
        {
            PlayerController.PlayerInstance.GetHealth(giveHp);

            Destroy(gameObject);
        }

        if (other.tag == "ground")
        {
            GetComponent<Rigidbody2D>().bodyType = RigidbodyType2D.Static;
        }

    }
}
```

IEnemyState.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public interface IEnemyState

{
    void Execute();
    void Enter(EnemyController enemy);


}
```

IdleState.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;


public class IdleState : IEnemyState
{

    private EnemyController enemyC;

    private float idleTimer;

    private float idleDuration;

    public void Enter(EnemyController enemy)
    {
        idleDuration = UnityEngine.Random.Range(1, 5);
        enemyC = enemy;
    }

    public void Execute()
    {
        Idle();

        if(enemyC.Target != null)
        {
            enemyC.ChangeState(new PatrolState());
        }
    }

    public void Idle()
    {
        enemyC.ThisAnimator.SetFloat("speed", 0);

        idleTimer += Time.deltaTime;

        if(idleTimer >= idleDuration)
        {
            enemyC.ChangeState(new PatrolState());
        }

    }




}
```

MeleeState.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class MeleeState : IEnemyState
{
   private EnemyController enemyC;

   private float meleeTimer;
   private float meleeDelay = 2;
   private bool canMelee = true;
   public void Enter(EnemyController enemy)
   {
      enemyC = enemy;
   }

   public void Execute()
   {
      Melee();
      if(enemyC.ShootRange && !enemyC.MeleeRange)
      {
         enemyC.ChangeState(new RangedState());
      }

      else if (enemyC.Target == null)
      {
         enemyC.ChangeState(new IdleState());       }
   }

  private void Melee()
   {
      meleeTimer += Time.deltaTime;

      if (meleeTimer >= meleeDelay)
      {
         canMelee = true;
         meleeTimer = 0;
      }

      if (canMelee)
      {
         canMelee= false;
         enemyC.ThisAnimator.SetTrigger("attack");


      }
```

```
    }


}
```

RangedState.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class RangedState : IEnemyState
{
    private EnemyController enemyC;

    private float shootTimer;
    private float shootDelay = 2;
    private bool canShoot = true;

    public void Enter(EnemyController enemy)
    {
        enemyC = enemy;
    }

    public void Execute()
    {
        Shoot();
        if (enemyC.MeleeRange)
        {
            // Executes only when enemy is tagged OnlyRange
            if (enemyC.tag != "OnlyRange")
            {
                enemyC.ChangeState(new MeleeState());
            }

        }
        else if (enemyC.Target != null)
        {
            if (enemyC.tag != "OnlyRange")
            {

                enemyC.Move();
            }

        }
        else
        {

            enemyC.ChangeState(new IdleState());
```

```
        }


    }
      private void Shoot()
    {
        shootTimer += Time.deltaTime;

        if(shootTimer >= shootDelay)
        {
            canShoot = true;
            shootTimer = 0;
        }

        if(canShoot)
        {
            canShoot = false;
            enemyC.ThisAnimator.SetTrigger("shoot");


        }

    }

}
```

PatrolState.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;


public class PatrolState : IEnemyState

{
    private EnemyController enemyC;
    private float patrolTimer;
    private float patrolDuration;


    public void Enter(EnemyController enemy)
    {
        patrolDuration = UnityEngine.Random.Range(1, 10);
        enemyC = enemy;
    }

    public void Execute()
```

```
    {
        Patrol();
        enemyC.Move();

        if (enemyC.Target != null && enemyC.ShootRange)
        {
            enemyC.ChangeState(new RangedState());
        }
    }

  public void Patrol()
    {


        patrolTimer += Time.deltaTime;

        if (patrolTimer >= patrolDuration)
        {
            enemyC.ChangeState(new IdleState());
        }

    }


}
```