LAB University of Applied Sciences

Platformer Game

From Concept to Working Prototype

LAB University of Applied Sciences
Bachelor of Engineering, Industrial Information Technology
2025
Anton Ivanov

Abstract

Author(s)	Publication type	Completion year
Anton Ivanov	Thesis, UAS	2025
	Number of pages 39 (thesis), 11 (appendices)	

Title of the thesis

Platformer Game

From Concept to Working Prototype

Degree, Field of Study

Engineer (UAS), Industrial Information Technology

Abstract

This thesis describes the creation process of the platformer game prototype. The development process started with the game's concept definition and went through production stages, including the development of core mechanics, enemies, environment, boss enemy, UI, and graphics.

In conclusion, a fully functional prototype of a platformer game was developed. The prototype is planned to be further improved by incorporating new textures, features, and other enhancements.

Keywords

Game, Platformer, Game Design, Programming

Contents

1 Introduction 1
2 Relevant Theory and Technologies 2
2.1 Relevance of the Chosen Topic 2
2.2 Platformers as a Game Genre 2
2.3 Unity as a Game Development Tool 3
2.4 Visual Studio Code 4
2.5 Unity Editor & Scripting API 4
2.5.1 C# Programming Language 4
2.5.2 Frame Rate 4
2.5.3 Game Objects & Components 5
2.5.4 Prefabs 5
2.5.5 Layers, layerMasks and Tags 5
2.5.6 Transform 5
2.5.7 Sprites and Sprite Renderer 5
2.5.8 RigidBody 6
2.5.9 Colliders & Platform Effector 2D 6
2.5.10 Raycast and OverlapCircle 6
2.5.11 Canvas 6
2.5.12 TextMesh Pro 7
2.6 Game Design Principles 7
3 Game Development Process Overview 8
4 Game Concept 9
5 Core Mechanics 13
5.1 Development Environment 13
5.2 First Steps 14
5.3 Player Movement Script 14
5.3.1 Script Layout 15
5.3.2 Player Movement Logic 16
5.3.3 Attack Logic 16
5.4 Health Scripts 16
5.5 Script Integration 17
6 Tower Generation 18
6.1 Tower Elements Creation 18
6.2 Tower Generator Script 19

6.3 Generation Controller 19
7 Enemies 21
7.1 Ghost Guards 21
7.2 Spiders 21
7.3 Enemy Generation 22
7.4 Heart Containers 22
8 Boss Enemy 24
8.1 Designing The Boss Enemy 24
8.2 Transition to Boss Battle Scene 25
8.3 Boss Battle Scene Construction 26
8.4 Boss Battle Logic 26
9 Graphics & UI 28
9.1 User Interface 28
9.1.1 Main Menu 28
9.1.2 Pause, Death and Victory Screens 29
9.1.3 Health Panel 30
9.2 Visual style & Graphics 31
10 Future Development Plans 33
11 Summary 34
References 36

Appendix 1. Produced Graphic Assets

Appendix 2. Gameplay Screenshots

List of Abbreviations

AI: Artificial Intelligence

API: Application Programming Interface

C#: C sharp

2D: two-dimensional

3D: three-dimensional

DLC: Downloadable Content

FPS: Frames per Second

HP: Hit Points or Health Points

OS: Operational System

PC: Personal Computer

PCG: Procedural Content Generation

UI: User Interface

VS Code: Visual Studio Code

WebGL: Web Graphics Library

1 Introduction

From the beginning of human civilization, entertainment was a crucial part of human culture and everyday lives. At first there were theatres, then board games, much later movies and now there are video games. For some, they are a form of entertainment like any other. For others, it is a highly interactive and expressive form of art. Yet both parties can agree that video games have become an important part of the modern entertainment industry.

Making a game takes a lot of patience and skill due to a number of reasons. First of all, a game needs a concept, which is a blueprint for what the project will become. Secondly, building game logic requires time, programming skills and knowledge of core game development principles. Thirdly, every game needs a visual aspect, which is represented in a form of images, models and animations. Those components need to be produced by hand according to the chosen style and theme of the product. Finally, a game requires music, which has to be fitting and be as active as it is subtle not to divide too much attention from gameplay.

Even though game development is challenging, some developers decide to undertake such a task, and they rarely succeed. One example of a successful game is the recent game Balatro, which was developed by a solo developer by the nickname “LocalThunk”. The video game uses the concept of the poker card game and adds additional gameplay mechanics to make it unpredictable. (Wrobleski 2024.) The game won awards in several Game Awards categories, including “Debut Indie Game” and “Best Mobile Game” (Bonelli 2024).

Inspired by stories of solo game developers and personal interest in the game industry, a decision was made to develop a prototype of a game with the aim of learning game development practices and exploring this process as a person unfamiliar with them.

2 Relevant Theory and Technologies

This chapter lists all required theory, terminology and technologies used in the creation of the game and the thesis. Justification for the chosen topic is given first, followed by explanations of utilized technologies and game development principles.

2.1 Relevance of the Chosen Topic

While many theses explore aspects of the game industry, few focus specifically on game development. A decision has been made to pick 3 theses that are close to the chosen topic.

The “Developing a Turn-based Strategy Game on Unity Engine” (Xuejian 2021) explores Unity game engine as a development tool and goes stage by stage through the game development process. The game offers strategic gameplay through completion of a variety of missions while controlling several distinct characters. The main features include the possibility to choose the difficulty level of a mission and a thoroughly developed UI. The game files are managed in JavaScript Object Notation (JSON) format, which is known for its readability and lightweight nature.

Similarly, the “Create an Endless Running Game in Unity” (Yancan 2016) dives deep into game development with Unity and shows its complexity and challenges but explores a completely different genre. Moreover, game design principles and game logic are thoroughly explained. In the developed game, players navigate an endless world by avoiding obstacles and enemies. Differences between developing games of two non-identical genres can be easily seen while comparing these two theses.

The “Game Development: Sci-Fi Endless Runner” (Boronin 2017) describes the development of a game in the same genre but uses Unreal game engine as a development environment. The author describes the history of video game development, the chosen genre, and utilities, highlighting advancements in game engine technology. The subject of the documentation is an Endless Runner game with a similar concept to the one previously described by Yancan (2016).

Unlike the chosen theses, this report describes the development process of a platformer game using Unity Engine. It may serve as a guide for readers unfamiliar with game development or beginning programmers looking for such information.

2.2 Platformers as a Game Genre

A platformer game is defined by traversing the game world utilizing running and jumping to overcome various obstacles and enemies. The most well-renowned and classic example of

a platformer is Super Mario Bros, which defined the staples of the genre for years to come. The effect of platformers being one of the oldest game genres is that numerous subgenres were born. To name a few:

- Action platformers are characterized by fast-paced gameplay and a focus on active interaction with the environment and enemies.
- Metroidvania, whose name comes from the game called Super Metroid, with its main feature being the continuous upgrade of the player character and newly opening sections of the previously visited maps.
- Precision platformers are difficult to master. They challenge the player to study these games' mechanics and environments to succeed.
- Even though originally platformers were produced with 2D graphics, 3D platformers were and are an innovative solution for the ever-evolving game industry and offer players the ability to traverse three-dimensional space changing their perspective on the genre itself. (Kelleher 2024.)

Platformers are popular in the modern day with new additions being released frequently. Newest ones to date are Neva, with its unique art style and creative storytelling, and Astro Bot – a "Game of The Year" award winner, which combines the PlayStation 5 console's main features with an engaging gameplay.

2.3 Unity as a Game Development Tool

A game engine is software that has capabilities and instruments for producing games and game related products. First games were made with embedded game engines, which were written exclusively for every game. Later, the decision was made to convert the main logic of games into a set of rules. Each game company eventually developed its own set of rules for game design and programming, which then evolved into the software templates or game engines. Modern game engines are equipped with a functional user interface which allows users who are not familiar with game programming to develop their products with ease and learn as they go. (Wikipedia 2018; Arm; Perforce.)

The Unity game engine was released in 2005 by Unity Technologies and has become one of the most widely used engines as of today. Unity is versatile and easy-to-use game development software with a built-in scripting API and an ever-expanding library of assets such as textures, animations, and music. It went through several version iterations, each adding more features than the previous. The most recent version, Unity 6, has provided

users with generative AI tools (Agate 2023.). Games such as Pokémon GO, Cuphead, and Hollow Knight were developed utilizing Unity.

2.4 Visual Studio Code

Visual Studio Code is a code editor software, which is available on MacOS, Linux and Windows. Its main features include being lightweight, supporting various programming languages, and having a consistently updated expansions library. (Microsoft 2025a.)

2.5 Unity Editor & Scripting API

Unity Real Time Development Platform or Unity Editor is an environment distributed by Unity for game development (Unity Technologies 2024). At the core of the Unity Editor is the Unity Engine, which handles the compilation of projects.

A script is a set of commands to be executed by a computer, written in a single file. The main difference from program files is that a script is not compiled. It is executed in real-time. (Brave 2023.)

As mentioned previously, the Unity Scripting API is one of the Unity Editor's functionalities, which allows for more flexible development of a project's logic through coding (Unity Technologies, Scripting). Coding is done in the C# programming language in the Visual Studio Code editor.

2.5.1 C# Programming Language

C# is one of the most popular programming languages of the .NET platform. C# is a versatile general-purpose language and can run on a wide array of platforms including phones, tablets, and the cloud, which makes it stand out in Internet of Things development. It belongs to the C language family and has a similar syntax to C++, Java and JavaScript. (Microsoft 2025b.)

2.5.2 Frame Rate

Frames per second is a value describing the number of frames in each second. A frame in this context is a still image. A game or video becomes more fluid with a higher number of FPS. (Brunner 2024.) That is the reason for its importance in game development. Frame rate is usually set by the developers, but it can vary depending on the hardware.

2.5.3 Game Objects & Components

A Game Object is a single instance of an object created in Unity and is everything a player sees in a game, ranging from a background image to a player character. They contain components, which are functional modules that modify their appearance or behavior. (Unity Technologies, Game Objects.)

2.5.4 Prefabs

The Unity Prefab system enables the creation of Game Object templates in the form of reusable assets. It is done by moving a Game Object to the Project window, where a Prefab is created. The original Game Object is stored with all its children and components. The main advantage of using Prefabs is that they can be easily instantiated through the editor or scripting and be accessed between scenes. (Unity Technologies, Prefabs.)

2.5.5 Layers, layerMasks and Tags

Layers allow Game Objects to be differentiated into levels for controlling rendering order and collision detection. Simply put, layerMasks are containers for layers in the Scripting API. They are assigned in the Inspector and used in code in collision detection functions. Tags are developer-made labels to distinguish objects mainly in code. They are both easy to define and use. (Unity Technologies, Layers; Introduction to layerMasks; Tags.)

2.5.6 Transform

The Transform is a default component for any Game Object, which cannot be removed. It governs position, rotation and scale of an object. In a 3D environment, the x-axis, y-axis and z-axis are available both in the Scene View and the Inspector view. In a 2D environment, only the x-axis and y-axis are available, though the z-axis is still modifiable in the Inspector. (Unity Technologies, Transform.)

2.5.7 Sprites and Sprite Renderer

According to the Unity Manual, a Sprite is a type of an asset in a Unity project, which can be an image or a texture. Sprite Renderer is a component responsible for the appearance of a Game Object. Sprites and therefore the Sprite Renderer are only present in 2D. In 3D, their counterparts are Meshes and the Mesh Renderer. (Unity Technologies, Sprites; Sprite Renderer)

2.5.8 RigidBody

The Unity Engine has built-in real-world physics simulation. A RigidBody component attached to a Game Object allows to use the physics engine for this object. Parameters like gravity and mass can be modified in the Inspector. Additionally, the component can be “frozen” in one or more axes. (Unity Technologies, RigidBody.)

2.5.9 Colliders & Platform Effector 2D

Colliders are used to simulate physical interactions between Game Objects. Primitive colliders are of basic shapes such as Box Collider or Capsule Collider. Primitive colliders are used in combination to create compound colliders. If the primitive collider does not fit a Game Object, then the Mesh Collider for 3D and the Polygon Collider for 2D can be used for more controlled distribution, but they strain hardware more than other colliders. Almost all colliders are present in 2D and 3D versions. A Collider can be set to the “Is Trigger” property to interact with script functionality. (Unity Technologies, Colliders)

The Platform Effector 2D is a component which is usually used together with 2D Colliders to allow one-way collisions. Use cases include a character passing through a platform in an upward direction but staying on the platform while moving downward. (Unity Technologies, Platform Effector 2D.)

2.5.10 Raycast and OverlapCircle

Both Raycast and OverlapCircle are parts of the Physics2D library. When Raycast is called, an invisible ray is drawn from a Game Object in a predefined direction. The method is primarily used to detect colliders. OverlapCircle functions similarly and detects all colliders in a circular area around a given Game Object. (Unity Technologies, Physics2D.Raycast; Physics2D.OverlapCircle.)

2.5.11 Canvas

In the Unity Editor, all UI elements are placed and manipulated on the Canvas, which the Unity Manual defines as a Game Object with a Canvas component. A Canvas is automatically created when any UI element is added to a scene. (Unity Technologies, Canvas.)

2.5.12 TextMesh Pro

TextMesh Pro is used as a tool for text creation and manipulation. It provides the user with various text customization options such as fonts, styles, and paragraph spacing. (Unity Technologies, TextMesh Pro Documentation.)

2.6 Game Design Principles

Regarding core game design principles, practices vary between companies and individual game designers; therefore, this section presents a selection of relevant principles. Volkov (2023) worked as a game developer for 13 years by the time he has written the referenced article. He highlights four main game design principles. “Keep it simple” principle states that each gameplay system should be simple and understandable in order to build and maintain it easily. “You aren’t gonna need it” principle specifies that gameplay features should be developed according to necessity at a particular development cycle to avoid having unnecessary or unused mechanics that complicate development. “Single responsibility” principle defines that a single system should govern a single part of a gameplay. It aims to simplify the game development process by keeping a connection between a single mechanic and the feature it encapsulates. “Loose coupling” principle states that every system should have minimal connection to other systems. This is done to ease the process of cutting and adding systems to a game. All four principles are clearly connected and work best if utilized together, which is illustrated throughout the development process of Jumpy Knight.

3 Game Development Process Overview

Game development is a time-consuming complex process carried out by a team of various specialists. The aim of this thesis was to produce the prototype of a platformer game. Given this limited scope and its development by a single person, the decision was made to describe a simplified overview of the main game development stages, with the focus on the implementation of the game's core functional components.

The game development process started with the planning stage. The initial concepts, their implementation methods, and narrative elements were defined, leading to the finalization of the game concept in the Game Design Document (GDD).

With the concept defined, development environment fundamentals were acquired to start the creation process of the game's core mechanics. Their purposes, design, and implementation methods were addressed during this phase.

Consequent stages could be completed in any order. Functionalities are presented in this thesis according to their relative development priority.

Subsequently, a fitting level generation method was chosen. All necessary assets and scripts were produced to create the desired game environment.

In the next stage, the focus shifted towards the design principles and implementation of enemies within the game's environment. Standard enemies were designed to populate the game levels and challenge the player. The "Boss" enemy was created based on the game's core mechanics to make the encounter fair and entertaining.

Afterwards, graphic assets and user interface elements were produced. Their general purpose is to immerse the player into the game world and support storytelling. Finally, the prototype was tested to exclude any errors or unintended behavior before further development.

4 Game Concept

Development of any game starts with a concept. It is expected to be concise, informative and appealing. The concept of a game encapsulates the following characteristics:

- game genre
- core mechanics
- setting
- art style
- story
- characters
- further development and support plans
- online aspect
- monetization.

The Game Design Document (GDD) was developed to help structure complex game concepts. It takes all the previously mentioned parts and organizes them in a neat and clear way. The GDD can also help form a feasible concept and features of the future game. These documents are used to pitch game concepts to publishers. Numerous GDD templates exist, and they slightly vary in size and structure. Figure 1 illustrates the structure of a variant of a GDD template.

1. Executive Summary, Quick overview

2. Target Audience

3. Main Characters

4. Main Features

4.1 Main mechanics

4.2 Movement

4.3 Physics

4.4 Multiplayer mode

5. Genre, Setting, Concept Art book*

6. Enemies, NPCs, Other objects

7. Story board, script*

7.1 Story overview

7.2 Progression, World 1

…

7.9 Progression, World 8

8. Technical definitions, Tech guide*

8.1 Platforms, versions

8.2 Control Scheme

8.3 Limitations

9. Business definitions*

9.1 In-app purchases

9.2 DLC packs

10. Outsourced/Bought Assets

Topics with (*) usually extended to separate detailed documents

Figure 1. The Game Design Document Structure (Kasurinen, 4)

Prior to the development of the game, the GDD was written to solidify the game concept, mechanics, and design. The concept of the game described there is as follows: “Jumpy Knight” is a platformer in which you take on the role of a knight ascending through a castle tower filled with various enemies, traps, and obstacles. The aim of the game is to reach the top floor and defeat an Evil Dragon.

The target audience of the game is expected to be average players of various age groups due to a gentle learning curve and non-complex mechanics. Fans of the platformer genre are also included in the target audience.

The only main character planned at the early development stage is a knight, who is the player character. His special trait is having “very strong legs”, with which he can perform powerful kicks. This is one of the core mechanics, and by utilizing it, the player can interact with the environment, for example, by kicking down doors or taking down enemies. The other core mechanics are movement in left and right directions and jumping.

Story and setting are core elements of any modern game. Early games like Pong or Super Mario Bros focused more on the gameplay than narrative. Later with the evolution of graphics and game development practices, stories became richer and more well-written (Bleeding Edge 2024).

The setting of the game is a giant medieval castle tower on top of which resides an Evil Dragon. It is protected by the hordes of monsters such as ghosts and spiders.

The game is done in a pixel art style. Pixel art originally appeared in the time of early computers, when they were unable to produce complex graphics because of hardware limitations. Then it allowed to save on computer resources and create a new form of entertainment – games. Nowadays, pixel art is used because of its simplicity in production and visual appeal. (Kordic 2025).

There is a variety of platforms that the game can be released on, ranging from consoles like PlayStation 5 or Nintendo Switch to PCs and laptops. They differ in their hardware limitations, development practices and publishing guidelines. If the game is developed by a solo developer or a small team, it is usually a good practice to target one platform and release a game there. Later in a game’s lifetime, it can be released on other platforms to reach a wider audience. (Neto 2024.) Jumpy Knight is planned to release first as a browser game to thoroughly test it and then later as a Windows distribution.

Monetization and DLCs are lifelines of games, because they allow players to directly support developers. A good example of this is that many developers release changes to the visual appearance of characters and charge relatively small sums of money for it. This

allows for players to customize their experience and developers to receive income on a regular basis. It is important to note that those purchases are voluntary. There is no planned downloadable content for Jumpy Knight at the time of writing, but it may be added in the future.

This concludes the GDD of Jumpy Knight. After the concept is finalized, the game development process can be started.

5 Core Mechanics

Filimowicz (2023) defines core mechanics as logic crucial for progression through a game and interaction with the game environment. The most common examples of such mechanics are movement, jumping, health, and attacking. This chapter is an overview of the logic behind above-mentioned core mechanics and their development and implementation.

5.1 Development Environment

Jumpy Knight is developed on the Unity 6 game engine and uses the Unity Editor as a development environment. Figure 2 depicts the user interface of the Unity Editor with marked fields for better visualisation.

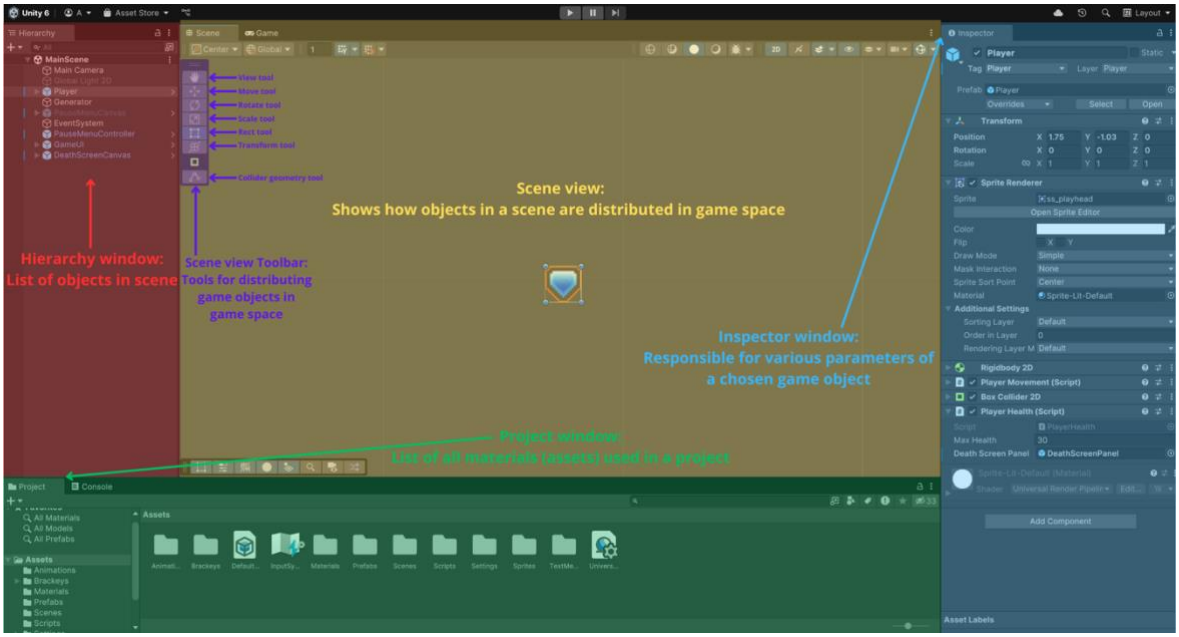

Figure 2. The Unity Editor User Interface

When a Game Object is created it is seen in the Scene view and can be manipulated using the Toolbar. The Unity Editor has a handful of tools to help interact with scale and position a Game Object.

The Hierarchy window on the left side of the Unity Editor shows a list of Game Objects and their children in the current scene. All the components belonging to the current Game Object can be found and modified in the Inspector window.

The Project window is a collection of all materials used in the project. These materials are usually called assets, and they include scripts, textures, scenes, Prefabs, and materials. All the imported files are also considered assets. Game Objects can be moved to the project window to create Prefabs. Prefabs are marked with a blue icon in the Hierarchy window. Assets can be organized into folders for a cleaner view and quicker access.

5.2 First Steps

Jumpy Knight is created as a 2D platformer, so a 2D Unity project fits this need perfectly. Any project opens in an empty scene. A scene is akin to a level in the game.

After getting familiar with the interface and tools, the first step is creating the player character Game Object. This is done by choosing a 2D sprite and its shape. The simplest one is a square. Default objects are created with Transform and Sprite Renderer components. At an early development stage, if a developer does not have sprites ready for Game Objects, it is a good practice to change their color to better distinguish them from one another.

For the player character to move, it needs to be affected by the physics engine. A Rigidbody2D component is added to the player object for that purpose. Next, a Box Collider 2D is added to simulate interaction with other game objects. For instance, standing on a platform is such an interaction. At the top of the Inspector, the layer and tag fields are assigned to “player” for later use.

Regarding the platform, it is created the same way as the player, except the RigidBody2D component is replaced with the Platform Effector 2D to allow the character to jump through it but to stay on it afterwards. The layer and tag fields are both set to “platform”. All previous and following objects are converted to prefabs for ease of instantiating them.

5.3 Player Movement Script

Turning now to player movement, it can be done in two ways. The first being through Unity Editor Visual Scripting, which is not covered in this thesis. The second way is using the Scripting API and Visual Studio Code. Prior to this, Unity should be linked to VS Code. It should be noted that only the main logic of scripts is described in this thesis.

5.3.1 Script Layout

A script can be created in the Project window or as a component of any Game Object in the Inspector. After a script is created, it can be opened and edited in VS Code. Figure 3 illustrates the layout of a newly created script.

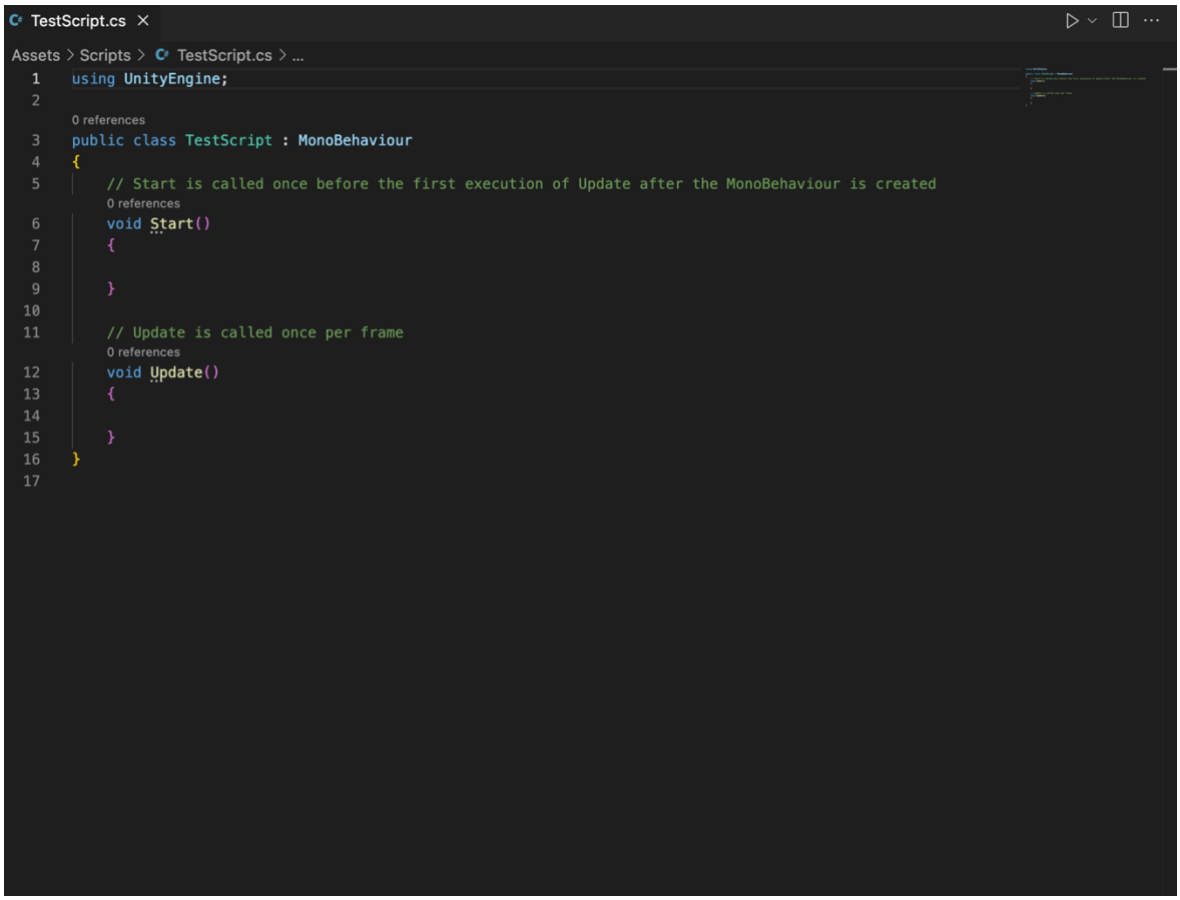

Figure 3. The Default Unity Script Layout

At the top of the script, the default UnityEngine library can be seen, which governs most of the functionality used in scripts. A library is a collection of classes and methods that are imported and used in projects. A standard class definition can be seen next. Among other features, it makes using this script's functionality available for other scripts in the project.

Two main functions are visible in the defined class. The Start function is called only once at the beginning of the script's execution (Unity Technologies, MonoBehaviour.Start()). It usually consists of setup code and variable assignment.

The Update function is called every frame and has a few variations (Unity Technologies, MonoBehaviour.Update()). The FixedUpdate version is used by the physics engine and has a different call frequency, which can be modified. The general rule is the lower the frames

per second count, the more frequently FixedUpdate is called. (Unity Technologies, MonoBehaviour.FixedUpdate().) LateUpdate is another version, which is executed after all Update functions in a frame. It is useful for fluid camera movement. (Unity Technologies, MonoBehaviour.LateUpdate().)

5.3.2 Player Movement Logic

Jumpy Knight's Player Movement script defines core movement mechanics such as moving in left and right directions, jumping, and attacking by kicking enemies. The logic of these mechanics will be discussed one by one in the given order.

Player movement is done through giving the player character a constant speed in the given direction. The speed is set in a variable and the direction is read from the player input. This enables the player to move left if the "A" key is pressed and to move right if the "D" key is pressed.

As for jumping, the logic is similar with the addition of new steps. The jump force is set in a variable and applied in an upward direction. To prevent the player from jumping in mid-air, a GroundCheck function is added. It works by creating a circle collider at the bottom of the player game object and detecting collisions with the "ground" or "floor" layer mask. The Jump function is executed when the "Space" key is pressed.

5.3.3 Attack Logic

Moving on to attack logic, in Jumpy Knight, the main way of interaction with enemies, objects, and the environment is "kicking". At the core of this mechanic is a ray of defined length, which is drawn from an assigned "kick" start point after the "K" key is pressed, and it checks for a collision with other game objects. In a case of a collision, it checks for either a Health component, tag, or a layer mask depending on the type of the object. Then, it executes the function responsible for registering a collision. For example, in a case of a collision with an enemy, it checks for a Health component and invokes the function that calculates damage taken.

5.4 Health Scripts

For attack logic to work, enemies must have a parameter, which depletes after they receive damage. Health Points or Hit Points typically serve as this required parameter, defining total amount of damage character can sustain before dying. The concept of HP came from the role-playing game called Dungeons and Dragons and later carried over to video games. (Uke 2024.) Early video games had a different health tracking system defined by the amount

of attempts a player can take before losing all their progress. This was aimed to make games replayable. Modern games moved away from such a system in favour of points that deplete after taking damage. (Macgregor 2018.)

A decision was made to create two separate health scripts to avoid parameters being used by a wrong entity: one for enemies and one for the player. They have similar functionality with a few differences.

Both scripts have a maximum health parameter set to the desired number. They share functions responsible for calculating damage by subtracting it from the maximum health amount, while checking if health is above zero. If the health is equal to or lower than zero, the death function is called, which stops the game and activates the Death Screen in the case of the Player Health script or destroys the enemy Game Object in the case of an Enemy Health script. Additionally, they share a function that returns the current health.

However, the Player Health script has logic that handles UI elements. This functionality is discussed later in the Graphics & UI chapter.

5.5 Script Integration

After scripts are complete, they are saved. As a result, they can be attached to Game Objects as components. All necessary parameters are set in the Inspector window. Ultimately, the player character has the ability to move, jump, attack, and has a defined health value. The next major step is tower generation.

6 Tower Generation

The main feature of the chosen genre is platforms, which make up a level. There are various approaches to level generation. One example is Procedural Content Generation (PCG), which implies that game content is not placed by hand but generated according to a set of rules. The advantages of the PCG method are high replay value and flexibility. (Pawson 2021.) Due to these reasons, Procedural Generation was chosen as the level generation method.

6.1 Tower Elements Creation

Utilizing this approach in a platformer means creating prefabs of all environment elements and then instantiating and arranging them through script logic. Consequently, to create a tower, the following elements are required:

- walls
- floors
- background
- roof
- platforms.

First of all, the wall prefab is created with Box Collider 2D and RigidBody2D components. The RigidBody2D type is set to “static”, which makes it unaffected by the physics engine and, therefore, immovable by any means. The layer and tag are set to the “wall” value.

Secondly, the ground Prefab is created exactly like the platform Game Object, except it is wider in size and has “floor” set as its tag. Its main function is defining the bottom of the tower and separating the tower’s upper level.

Thirdly, the background is constructed as a dark blue Game Object to fit the game colour scheme with default components. To ensure this Game Object is rendered behind all others, the “Order in Layer” parameter of the Sprite Renderer is set to “- 2”.

Lastly, the roof of the tower is added as a triangle sprite with a Polygon Collider 2D to visually finish the construction and prevent the player from jumping out of bounds. Platforms were created earlier and stay unchanged.

6.2 Tower Generator Script

The next step is the creation of a script, which encapsulates generation logic. It starts with walls, floor and background generation. Their position and size are set according to the level width and a given scale. The scene structure consists of a floor defining the base, walls enclosing the horizontal space on both sides, and a background residing at the back.

Following this, the script fills the environment by iteratively adding platforms in ascending layers. The number and exact placement of platforms on each layer is calculated based on the level width and a limited range of random numbers, introducing variability. The maximum vertical height reached by these platforms is tracked during this process. After platform generation is complete, the script adjusts the height of the walls and background to match the maximum height of the generated structure with an offset to account for the top of the tower.

Finally, a concluding structure is generated at the top of the level, consisting of a floor separating the apex from the platforming section, a roof, and an interactive door leading to the boss stage. The entire generation process relies on instantiating previously made Prefabs, where key parameters controlling counts, spacing, and probabilities are configurable through the Inspector window.

6.3 Generation Controller

To actualize the script logic, an empty Game Object is created with the Tower Generator script attached as a component. All the previously made prefabs are assigned in the corresponding fields in the Inspector. Figure 4 showcases one of the possible results of tower generation. The tower is always unique and introduces the potential for repeated engagement.

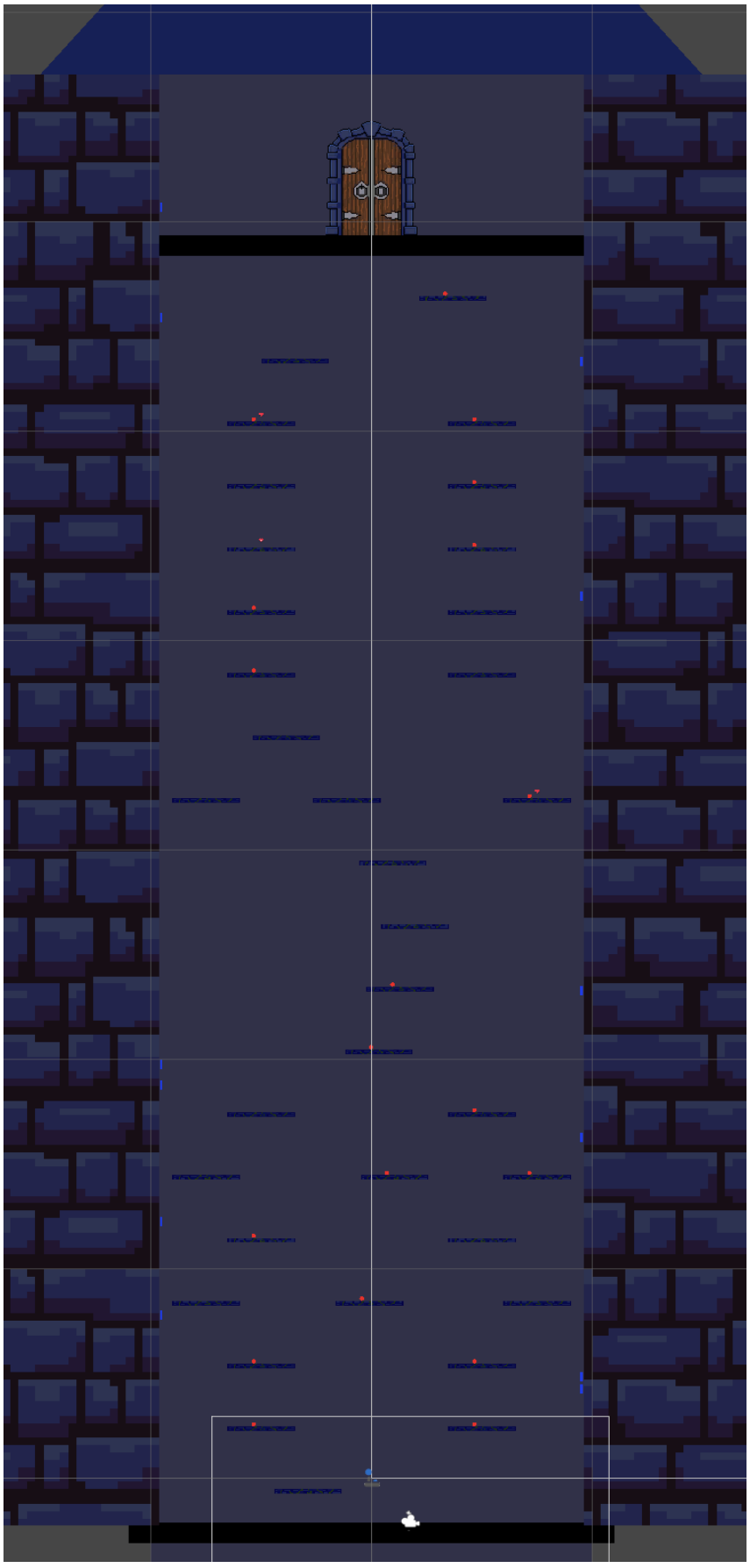

Figure 4. The Tower Generation Variant

7 Enemies

Enemies in games are created to challenge the player. They oppose the player's progression and test their game knowledge, reflexes, or philosophy. Enemies are made with the protagonist abilities in mind and are designed to have both counters and weaknesses to those skills. (Kraj 2020a.)

In Jumpy Knight, enemies were designed following the same principles. For instance, the Ghost Guard's main objective is to prevent players from staying on platforms for too long. They traverse between the edges of platforms and deal damage to the player on collision. However, any attack from the player is fatal for a Ghost Guard. The other type of an enemy is Spiders. They navigate walls within set intervals and shoot players with web projectiles. On hit, they deal damage and slow the player down. Spiders could be defeated the same way as Guards, with a possibility of being eliminated by a reflected web projectile, if it was hit with an attack within a short opportunity window.

7.1 Ghost Guards

The Ghost Guard is created as a red circle sprite. A Circle Collider 2D is attached to accurately match the shape of the Game Object. The RigidBody2D component "Body Type" is set to "kinematic". Additionally, the Game Object is configured with the Enemy Health script, with its maximum health parameter set equivalent to the player's attack damage. This ensures the enemy is dispatched by a single hit. The layer is set to "enemy" to detect attacks.

The GuardAI script defines the behavioral pattern for Ghost Guards. Initially, a guard is instantiated slightly above a platform and descends until it detects a surface. Upon ground contact, it transitions to executing a patrol pattern along the platform. The guard utilizes a downward ray, which upon detecting a platform edge, automatically reverses the guard's direction of movement. Furthermore, this enemy inflicts a predefined amount of damage upon physical contact with the player by activating the OnCollisionEnter2D function. It checks if the Collider 2D has detected an object with the "player" tag and calls the PlayerTakeDamage function of the Player Health script.

7.2 Spiders

Constructing the Spider Game Object largely mirrors the same process for the previous enemy. However, the Spider utilizes a blue, horizontal rectangular sprite, and its Rigidbody2D component's "Body Type" is set to "Dynamic", facilitating proper wall

spawning. Furthermore, a web projectile Game Object is added with Circle Collider 2D and RigidBody2D components attached.

Spider behavior is split between the SpiderAI and Web Projectile scripts. When a Spider is instantiated, proximity to wall surface is checked by a Raycast method. With a wall detected, it starts a linear vertical movement on the wall surface, with a periodically inverted direction based on a timer. It utilizes OverlapCircle to detect the player's collider. Upon detection, the Spider initiates ranged attacks, periodically launching web projectiles towards the player's position, delayed by a cooldown. At the same time, the Web Projectile script methods are called. The OnTriggerEnter2D executes a call to the PlayerTakeDamage function and slows the player down. However, if the projectile is reflected, its color is changed, homing behavior is applied, and it deals devastating damage to the Spider.

Consequently, the Player Movement script is modified to handle web projectile deflection logic. Deflection is triggered by attacking the web projectile right before it hits the player.

7.3 Enemy Generation

Due to the tower being procedurally generated, enemies cannot be placed manually in designated spots and need to be instantiated by a script. To achieve this, the Tower Generator script must be expanded.

Initially, the SpawnGhostGuardsOnPlatforms function receives the number and placement of all platforms and the desired number of guards to be instantiated. Next, it randomly sets a target platform for each guard and calculates the deployment point coordinates. The Y coordinate is calculated with an offset, generating the enemy slightly above the platform to account for any errors.

Similarly to the previous function, the SpawnSpidersOnWalls requests references to both walls and the required number of spiders. Then, the instantiation position is compiled using the wall width, spacing, and random coordinates. Lastly, spiders are rotated to match the target wall. The position of each spider is validated to ensure it does not intersect with any existing enemies.

7.4 Heart Containers

As a result of enemy implementation, the game has become more challenging. Player can withstand only 3 instances of damage before losing. This can be punishing, so it needs to be balanced. In order to keep the difficulty on a challenging level, but let the player recover

from their mistakes, health replenishing can be implemented. It is achieved through the creation of heart containers, which recover health on contact.

A heart container is created as a relatively small 2D sprite compared to the platform Game Object. A RigidBody2D component is attached, with its “Body Type” set to “kinematic”, for it to be controlled solely by the script. To allow player interaction with the heart container, fitting Collider 2D component is added with the “Is Trigger” parameter ticked.

Heart container logic is programmed in three scripts:

- Heart Container script
- Player Health script
- Tower Generation script

The OnTriggerEnter2D function is initialized in the Heart Container script. In case of a collision with the player, it calls a function responsible for health replenishing from the Player Health script. Referred function increments the health value by a set amount.

The generation of hearts is handled in the Tower Generator script. The position of each heart is decided by a formula, which starts the creation probability at zero percent and increases it with the progression through the tower. That way, hearts are more likely to be instantiated at the end of the level, where the player needs them, but have a moderate chance to appear at the beginning or middle.

8 Boss Enemy

Boss enemies in various games are made to test the player's skills and resolve. They usually act as a barrier and separate the player from the next level or area. (Doan 2017.) However, such enemies also challenge developers. Creating an appropriate Boss is a complex task that requires a deep understanding of the game's mechanics, themes, and complexity.

8.1 Designing The Boss Enemy

From the start, the Evil Dragon was meant to be one of the Bosses of Jumpy Knight. Story-wise, it resides at the top of the tower and has personal animosity towards the Knight. The main function of this Boss is to test the player's understanding of the game mechanics and provide them with a fair challenge.

The boss battle commences at the top of the tower. The stage layout is presented in Figure 5.

Figure 5. The Boss Stage Layout

Four platforms were placed equally on both sides. Space was delimited by two walls, the floor and a roof. The Dragon’s head was the only interactable part of the enemy and was placed in between the platforms. Such placement was based on the player’s attack range. The rest of the body consisted of the wings and tail, and acted as a visual representation of the Dragon’s size.

The Boss was designed to have two phases. In the first phase, the Dragon used the fire breath attack covering the bottom area of the arena. Hot Pebbles flew out from the fire and land on platforms. When enough damage was dealt to the Dragon by launching hot pebbles into its head, the transition to phase two was activated.

The transition to the second phase was initiated by a visual tremor affecting the tower, caused by the Dragon’s wrath. Following this event, the tower started to crumble with debris falling from the ceiling. Debris dealt devastating damage on impact and sent deadly shockwaves across the stage floor. Once a specific time had elapsed, the Dragon's claws emerged and became vulnerable to player attacks. Successfully hitting both claws caused the Dragon's head to lower, exposing it for the final attack that concluded both the battle and the game.

All of the previously mentioned elements have made it into the current version of the game after being refined. Adjustments were made based on technical performance requirements, ensuring a smooth player experience.

8.2 Transition to Boss Battle Scene

Prior to the Boss enemy implementation, the transition to the new Boss Battle scene had to be addressed. Initially, a new scene, which will contain the Boss Battle stage, was created. A trigger was required to transition to the created scene.

In Jumpy Knight, the boss room door acts as a Game Object transitioning the player into the Boss Battle scene. It is generated as a part of the top of the tower in the Tower Generator script. The Door Game Object is a 2D sprite with a Box Collider 2D attached to detect player attacks. Its functionality is mainly set in the Giant Doors script, where the transition logic is executed. However, the Kick function of the Player Movement script was additionally modified to interact with the door collider component, activating the Giant Doors script.

8.3 Boss Battle Scene Construction

The construction of the Boss Battle scene began with the tower room and followed the layout discussed in the preceding section. Previously manufactured prefabs were used to create the floor, walls, roof, and the platforms of the stage.

Next, the Dragon's Head, Tail, and Wings Game Objects were added with the required components. As a way of simulating the Dragon's fire breath attack, an empty Game Object containing a Particle System component was placed on top of the floor, covering the area under the platforms. The Particle System was responsible for generating a set number of particles within the defined area. Its activation was triggered via script.

Rest of the objects are created as prefabs to be instantiated when necessary. For example, the Hot Pebble prefab comprises an orange circular sprite, along with Circle Collider 2D and RigidBody2D components essential for its physics interactions. Conversely, the Debris and Shockwave prefabs utilize rectangular sprites and include only a Box Collider 2D component. A RigidBody2D was omitted as these objects do not require simulation by the physics engine. Furthermore, distinct prefabs were developed for the Dragon's left and right claws, each incorporating a Box Collider 2D. These claw objects were initially positioned in the same space as the two lower platforms and were scripted for activation late in the second phase.

8.4 Boss Battle Logic

Implementing logic for the Boss Battle was a highly complex task and involved the modification of many previously written scripts. Dragon Fire Attack script activates the Particle System after a short delay, so the player has time to react. If the player is detected in the Dragon's "fire breath" collider area, Fire Damage script removes 10 health points for each second player remains within it, which forces them to escape to the platforms.

Simultaneously, the Dragon Fire Attack script signals the Pebble Spawner script to begin instantiating "hot pebble" prefabs. These prefabs are then launched from the "fire breath" area towards the platforms, following a parabolic trajectory. Upon landing, the associated Pebble Movement script changes the pebble's "Body Type" property to "static". This change prevents further physics interactions, aside from collision checks with player attacks. Additionally, the Pebble Movement script is responsible for managing the object pool by removing excess "hot pebble" instances from the scene.

When the modified Kick function of the Player Movement script detects a collision with a "hot pebble", the pebble is propelled towards the Dragon's Head. This action is handled by

executing the Pebble Kicked script attached to the “pebble”, which also increments a counter tracking successful hits against the Dragon. Once this counter reaches three, a phase transition is triggered: the “fire breath” attack, “hot pebble” instantiation, and four platforms are disabled. Immediately, Dragon Phase Two script is activated.

Phase transition is signified by the tower visually shaking. This effect is achieved by randomly offsetting the coordinates of every Game Object in the scene by insignificant amounts. Subsequently, debris prefabs begin instantiating just below the ceiling. A Tower Debris script configures each instantiated debris object, adding a RigidBody2D component and reducing its gravity scale to ensure a slow descent. When the debris hits the floor, it deals significant damage to the player, and shockwaves are emitted horizontally. These shockwaves, instantiated and oriented correctly by the Shockwave script, inflict damage upon contact with the player and dissipate upon hitting a wall.

After 10 seconds of this sequence, two platforms supporting the Dragon’s Claws appear, which are vulnerable to player attacks. Once both claws have been hit, all active hazards and attack sequences are disabled, and the Dragon’s Head is slowly lowered towards the floor. To finish the game, the player must hit the lowered head, triggering the logic that displays the Victory Screen.

With the Boss Battle successfully implemented, all the main mechanical aspects of Jumpy Knight are finished. Consequently, the next chapter of the thesis will focus on implementing the user interface, textures, and animations.

9 Graphics & UI

The visual aspect or graphics of a game are as important as the gameplay. It affects the player's immersion into the game and, consequently, the overall experience from it. If the design of the game is made poorly, the player may be distracted from the game's world and story. (Addo 2017.)

In addition to graphics, the game's user interface must be as appealing and stylistically fit as it is functional to attract the player. Several approaches to making UI exist. For instance, the "non-diegetic" approach is comprised of overlay menus, which exist outside of the game world. In contrast, the "diegetic" approach entails embedding UI into the game and making menus a part of the game world, for example, presenting UI elements as holographic displays or maps held by characters. (Kraj 2020b.)

9.1 User Interface

The user interface in Jumpy Knight was made utilizing the "non-diegetic" approach to shorten the production time. All created menus follow a similar logic and were created by utilizing shared functions.

9.1.1 Main Menu

The Main Menu was the first implemented UI element. To make it, a new scene was created named Main Menu. The layout of the Main Menu is presented in Figure 6.

Figure 6. The Main Menu UI Layout

Initially, a sprite of a brick wall was set as the background image on the automatically created Canvas. The brick wall sprite is a part of the imported free asset package provided by Brackeys (2020). Next, TextMesh Pro was used to create the game title and the game version text. In the lower part of the screen are the play and quit buttons. For them to become functional, corresponding script functions must be created and assigned in the Inspector window. Therefore, the Main Menu script was created. The Play function attached to the play button loads the Game scene. The behavior of the Quit function varies based on the platform the game is running on:

- Quits the game view if played in the Unity Editor.
- Closes a browser window if played in the browser.
- Quits the application on all other platforms.

9.1.2 Pause, Death and Victory Screens

The Death and Victory screens utilize shared logic, pausing gameplay upon activation. The Death screen triggers when the player health reaches zero, while the Victory screen appears after defeating the Dragon boss. Both screens offer identical “Restart” and “Quit to Menu” options, which load the Game scene or the Main Menu scene respectively. (Appendix 2.)

Activated by pressing the “Escape” key, the Pause menu halts game time. It features “Restart” and “Quit to Menu” buttons functioning identically to those on the Death and Victory screens. Additionally, a “Resume” button allows the player to return to gameplay by restoring the normal time scale. (Appendix 2.)

9.1.3 Health Panel

It is important to keep the player aware of their character’s health status so that they can adapt their approach and playstyle to any situation. Health displays exist for that purpose and can be presented in many variations. The health display design chosen for Jumpy can be seen in Figure 7.

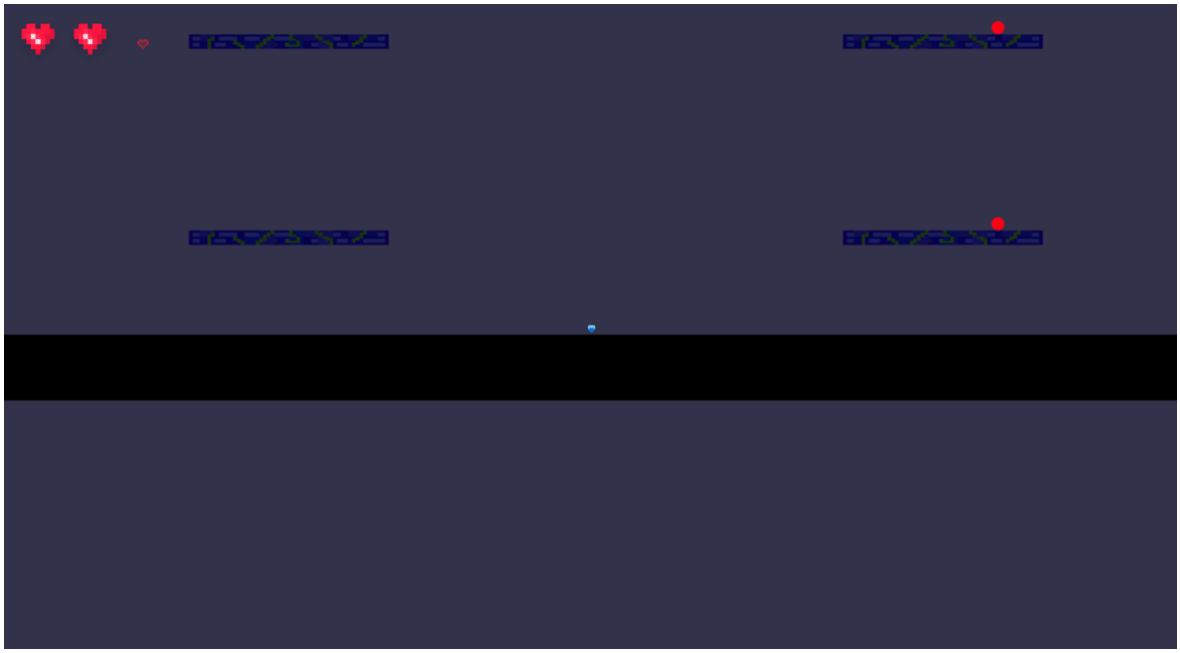

Figure 7. The Health Display Design

The heart panel is created as a part of the overlay UI canvas. In upper-left corner of the screen, three heart sprites, arranged in a row, depict the player’s current health status. The heart sprites are part of the previously mentioned asset package (Brackeys 2020). When the health is depleted, the original sprite is replaced by a smaller empty heart sprite. This process is handled by the Player Health script.

9.2 Visual style & Graphics

Mechanically, the game is working as intended but is quite incomprehensible for the player. Simple sprites were used to represent characters and objects in the development process. Still, they are not appealing or immersive. To replace them proper sprites must be drawn.

As the game design was planned to be done in Pixel Art style, sprite production was done in the Piskel web-based service, which provides many tools for Pixel Art and animation development (Piskel). The Piskel editor layout is visualized in Figure 8.

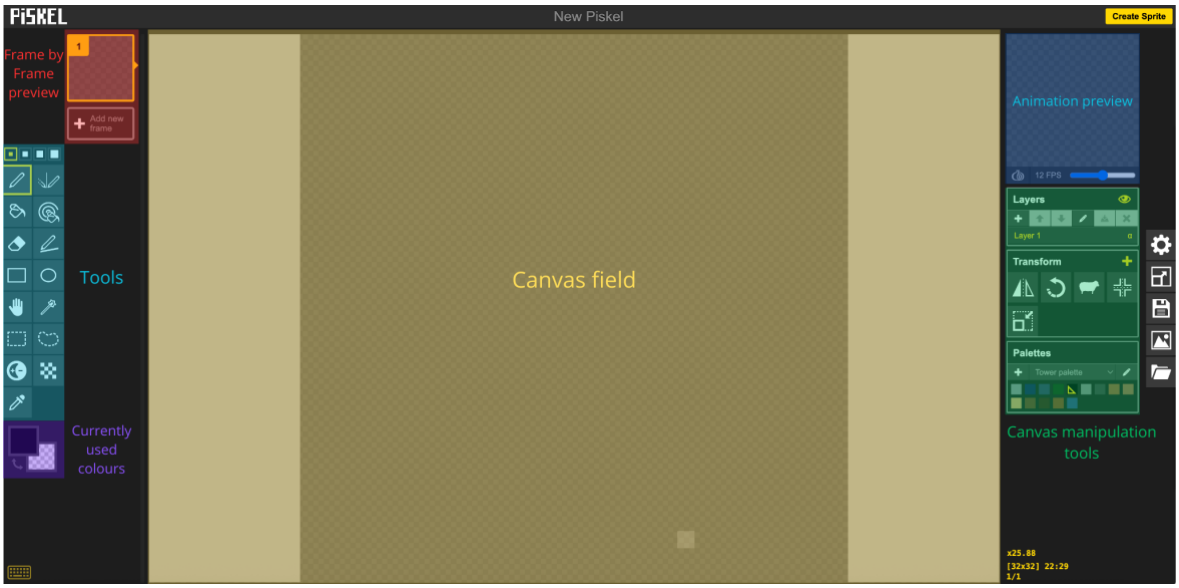

Figure 8. The Piskel Editor Layout

A design is drawn in the canvas field by utilizing the various provided tools seen on the left. The currently used colors are displayed under the tools. The upper left and lower right colors are applied by pressing the left and right mouse buttons respectively. New frames can be added in the upper-left corner, while the complete animation is previewed on the opposite side. Canvas manipulation tools are situated lower and allow the creation of additional layers, palettes, and canvas orientation alterations.

The initial visual development phase began with acquiring pixel art fundamentals, using works by various online artists as references for creating sprites and animations. The first asset completed was the Dragon’s Head sprite, designed in two variations: open and closed eyes, followed by the Dragon’s Body sprite, primarily drawn to establish the enemy's scale. To provide a visual cue for the Dragon's “fire breath” attack, a continuous animation was

created by combining 13 distinct sprites of a spark. Then, sprites for the platforms, Dragon's claws, debris, shockwaves, and empty hearts were designed to further refine newly acquired skills. Among these initial assets, the door sprite is the most complex, created utilizing shading and highlighting techniques. (Appendix 1.)

10 Future Development Plans

The development of Jumpy Knight is not yet finished. The game can still benefit from new mechanics, design elements, and level expansion.

Music has not been mentioned in this thesis because of the difficulties with asset search and music production. Nonetheless, the addition of music will greatly increase the player immersion and the game's overall appeal.

New assets for enemies, the tower interior and the player character are planned to be produced. Depending on the success of the designing process, animations will be added to introduce fluidity into the game process.

Furthermore, new mechanics will expand the gameplay of Jumpy Knight. For example, traps were originally planned for the alpha version of the game but then were put aside to shorten the production time. Additional enemies and bosses will be added to new sections of the tower to make the player experience diverse and engaging.

In-app purchases, DLCs, and other paid content are not planned in the early development stage. Nevertheless, after the game's publication, they can be added, for instance, in the form of character customization.

Initially, the game is planned to be published as a web-based project to be accessible both for the developer and for the player. There it will be thoroughly tested and subsequently improved. Later in a game's lifetime it might be released on other platforms including PC and consoles.

11 Summary

In conclusion, a working prototype of Jumpy Knight has been developed successfully. This prototype effectively demonstrates the core gameplay loop, integrating the key platforming mechanics and the multi-phase boss battle discussed within this thesis.

The development process started with conceptualizing. The concept was described in the Game Design Document, which made it both presentable and comprehensive. The developed game was set to be a platformer in a castle tower setting filled with various enemies. The objective of the game is to ascend through the tower and defeat the Evil Dragon.

Concept realization started with the implementation of fundamental mechanics, namely linear movement, jumping, and attacking, alongside the health system. At this stage, initial assets for the player character and the environment were created.

The next step was the creation of the tower and its structure. A procedural generation method was utilized to generate a unique level with every attempt, which highly diversified the player experience.

The process of enemy implementation followed, with the creation of Ghost Guards and Spiders. Ghost Guards followed a patrolling behavior and inflicted damage to the player on impact. Spiders moved vertically and shot the player with projectiles, which reduced the player's speed. As a result of the chosen level generation method, enemies were included in the generation logic.

The Evil Dragon was created at the later stages of the game prototype development as a boss enemy to test the player's skills and knowledge of the game. The boss battle was split into two distinct stages built upon the game's core mechanics. Defeating the boss enemy marked the completion of the game process.

Lastly, the UI and the visual aspect of the game were designed and implemented. UI functional elements such as the Main and Pause menus and the Health Panel were created with the addition of the Death and Victory screens. A few graphic assets were produced to establish the general style of the game's prototype.

The game is planned to be expanded further with additional levels, enemies and features. Valuable knowledge and insight were gained in the production process of the game and this thesis.

Nevertheless, numerous challenges were encountered, such as the lack of knowledge and time. For instance, sprite production took longer than anticipated and several features were

put aside to finish the main elements of the game. Fortunately, these issues were solved, and the target of this thesis was achieved.

Screenshots of the Jumpy Knight gameplay are stored in Appendix 2. All the scripts referenced throughout this thesis can be found by accessing the following link: https://github.com/gluck243/JumpyKnightGame.

References

Addo, A. 2017. Pretty Pixels – The Importance of Visual in Game Design. Medium. Retrieved on 21 April 2025. Available at https://medium.com/@AndersonAddo/pretty-pixel-the-importance-of-visuals-in-game-design-5f3ae148a41e

Agate. 2023. History of Unity Game Engine. Retrieved on 4 March 2025. Available at https://agate.id/history-of-unity-game-engine/

Arm. Gaming Engines. Retrieved on 7 April 2025. Available at https://www.arm.com/glossary/gaming-engines

Bonelli, J. 2024. Balatro Creator Jokes About Winning Multiple Game Awards. Game Rant. Retrieved on 15 March 2025. Available at https://gamerant.com/balatro-creator-game-awards-comment/

Boronin, D. 2017. Game Development: Sci-Fi Endless Runner. Savonia University of Applied Sciences. Retrieved on 6 February 2025. Available at https://www.theseus.fi/bitstream/handle/10024/122550/Boronin_Dmitry.pdf?sequence=1&isAllowed=y

Brunner, D. 2024. Frame Rate: A Beginner's Guide. TechSmith Corporation. Retrieved on 8 April 2025. Available at https://www.techsmith.com/blog/frame-rate-beginners-guide/?srsltid=AfmBOoq22KGFMtgr3FopAF0PXTCMnpqypkUONlVzeaNDFl8BjIJPb3ZG

Bleeding Edge. 2024. The Evolution of Narrative in Video Games: A Journey Through Time 2024. Retrieved on 2 April 2025. Available at https://bleedingedge.studio/blog/exploring-narrative-in-video-games-2024/

Brackeys. Free 2D Mega Pack. Unity Asset Store. Retrieved on 21 April 2025. Available at https://assetstore.unity.com/packages/2d/free-2d-mega-pack-177430#publisher

Descottes, J. Piskel. Retrieved on 22 April 2025. Available at https://www.piskelapp.com

Doan, D. 2017. GameDev Thoughts: Designing Engaging Boss Fights in Action Games. Medium. Retrieved on 16 April 2025. Available at https://medium.com/black-shell-media/gamedev-thoughts-designing-engaging-boss-fights-in-action-games-27e68e6109ab

Filimowicz, M. 2023. Core vs Non-Core Mechanics. Medium. Retrieved on 3 April 2025. Available at https://medium.com/understanding-games/core-vs-non-core-mechanics-9360e5e90f93

Kasurinen, J. Game Design Document. LUT Game Development Project course (CT60A5401). Retrieved on 1 April 2025. Limited availability at https://moodle.lut.fi/course/view.php?id=19303

Kraj, N. 2020a. Keys to Rational Enemy Design. GD Keys. Retrieved on 14 April 2025. Available at https://gdkeys.com/keys-to-rational-enemy-design/

Kraj, N. 2020b. Designing Efficient User Interfaces For Games. Medium. Retrieved on 21 April 2025. Available at https://medium.com/@nicolaskraj/designing-efficient-user-interfaces-for-games-be20b516f1c2

Kelleher, L. 2024. Defining the Platformer genre. Kelleher Bros. Retrieved on 4 March 2025. Available at https://www.kelleherbros.com/blog/2024/3/4/defining-the-platformer-genre

Kordic, A. 2025. What Exactly is Pixel Art and How Did It Come Back To Life? Artsper Magazine. Retrieved on 1 April 2025. Available at https://blog.artsper.com/en/a-closer-look/art-movements-en/pixel-art/

Microsoft. 2025a. Visual Studio Code. Retrieved on 7 April 2025. Available at https://code.visualstudio.com

Microsoft. 2025b. A tour of the C# language. Retrieved on 8 April 2025. Available at https://learn.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/overview

Macgregor, J. 2018. The history of hit points. PC GAMER. Retrieved on 9 April 2025. Available at https://www.pcgamer.com/the-history-of-hit-points/

Neto, M. 2024. Why Developers Port Games to Other Platforms & How It Affects Engagement. Retrieved on 24 April 2025. Available at https://80.lv/articles/why-developers-port-games-to-other-platforms-how-it-affects-engagement/

Pawson, K. 2021. Procedural Generation: An Overview. Medium. Retrieved on 10 April 2025. Available at https://kentpawson123.medium.com/procedural-generation-an-overview-1b054a0f8d41

Perforce. The Complete Game Engine Overview. Retrieved on 7 April 2025. Available at https://www.perforce.com/resources/vcs/game-engine-overview

Volkov, A. 2022. Clean game design principles. Medium. Retrieved on 15 March 2025. Available at https://medium.com/@tricky_fat_cat/clean-game-design-principles-8000ffdd48e1

Uke, J. 2024. HP Meaning in Games: Definition, and Use Cases. GameTree PBC. Retrieved on 9 April 2025. Available at https://gametree.me/gaming-

terms/hp/#:~:text=HP%20in%20games%20stands%20for,your%20character's%20health%20or%20durability.

Unity Technologies. Canvas. Unity Manual. Retrieved on 21 April 2025. Available at https://docs.unity3d.com/Packages/com.unity.ugui@2.0/manual/UICanvas.html

Unity Technologies. Colliders. Unity Manual. Retrieved on 7 April 2025. Available at https://docs.unity3d.com/560/Documentation/Manual/CollidersOverview.html

Unity Technologies. Game Object. Unity Manual. Retrieved on 7 April 2025. Available at https://docs.unity3d.com/Manual/class-GameObject.html

Unity Technologies. Introduction to layerMasks. Unity Manual. Retrieved on 9 April 2025. Available at https://docs.unity3d.com/6000.0/Documentation/Manual/layermask-introduction.html

Unity Technologies. Layers. Unity Manual. Retrieved on 9 April 2025. Available at https://docs.unity3d.com/6000.0/Documentation/Manual/Layers.html

Unity Technologies. MonoBehaviour.Start(). Unity Manual. Retrieved on 8 April 2025. Available at https://docs.unity3d.com/6000.0/Documentation/ScriptReference/MonoBehaviour.Start.html

Unity Technologies. MonoBehaviour.Update(). Unity Manual. Retrieved on 8 April 2025. Available at https://docs.unity3d.com/6000.0/Documentation/ScriptReference/MonoBehaviour.Update.html

Unity Technologies. MonoBehaviour.FixedUpdate(). Unity Manual. Retrieved on 8 April 2025. Available at https://docs.unity3d.com/6000.0/Documentation/ScriptReference/MonoBehaviour.FixedUpdate.html

Unity Technologies. MonoBehaviour.LateUpdate(). Unity Manual. Retrieved on 8 April 2025. Available at https://docs.unity3d.com/6000.0/Documentation/ScriptReference/MonoBehaviour.LateUpdate.html

Unity Technologies. Prefab. Unity Manual. Retrieved on 8 April 2025. Available at https://docs.unity3d.com/Manual/Prefabs.html

Unity Technologies. Platform Effector 2D. Unity Manual. Retrieved on 8 April 2025. Available at https://docs.unity3d.com/550/Documentation/Manual/class-PlatformEffector2D.html

Unity Technologies. Physics2D.Raycast. Unity Manual. Retrieved on 15 April 2025. Available at https://docs.unity3d.com/6000.0/Documentation/ScriptReference/Physics2D.Raycast.html

Unity Technologies. Physics2D.OverlapCircle. Unity Manual. Retrieved on 15 April 2025. Available at https://docs.unity3d.com/6000.0/Documentation/ScriptReference/Physics2D.OverlapCircle.html

Unity Technologies. 2024. Real Time 3D Development Platform & Editor. Retrieved on 7 April 2025. Available at https://unity.com/products/unity-engine

Unity Technologies. Rigidbody. Unity Manual. Retrieved on 7 April 2025. Available at https://docs.unity3d.com/550/Documentation/Manual/class-Rigidbody.html

Unity Technologies. Scripting. Unity Manual. Retrieved on 8 April 2025. Available at https://docs.unity3d.com/6000.0/Documentation/Manual/scripting.html

Unity Technologies. Sprites. Unity Manual. Retrieved on 4 April 2025. Available at https://docs.unity3d.com/550/Documentation/Manual/Sprites.html#:~:text=Sprites%20are%202D%20Graphic%20objects,efficiency%20and%20convenience%20during%20development

Unity Technologies. Sprite Renderer. Unity Manual. Retrieved on 24 April 2025. Available at https://docs.unity3d.com/550/Documentation/Manual/class-SpriteRenderer.html

Unity Technologies. Tags. Unity Manual. Retrieved on 9 April 2025. Available at https://docs.unity3d.com/Manual/Tags.html

Unity Technologies. Transforms. Unity Manual. Retrieved on 7 April 2025. Available at https://docs.unity3d.com/Manual/class-Transform.html

Unity Technologies. TextMesh Pro Documentation. Unity Manual. Retrieved on 21 April 2025. Available at https://docs.unity3d.com/Packages/com.unity.ugui@3.0/manual/TextMeshPro/index.html

Wikipedia. 2018. Game engine. Retrieved on 7 April 2025. Available at https://en.wikipedia.org/wiki/Game_engine

Wrobleski, X. 2024. The 10 best games made by single person. CBR. Retrieved on 15 March 2025. Available at https://www.cbr.com/best-games-made-by-single-person/

Xuejian, X. 2021. Developing a Turn-based Strategy Game on Unity Engine. Vaasa University of Applied Sciences. Retrieved on 6 February 2025. Available at https://www.theseus.fi/bitstream/handle/10024/500525/XiaoXuejian.pdf?sequence=2&isAllowed=y

Yancan, Z. 2016. Create an Endless Running Game on Unity. Mikkeli University of Applied Sciences. Retrieved on 6 February 2025. Available at https://www.theseus.fi/bitstream/handle/10024/119971/bachelor%20thesis%20ZhangYancan.pdf?sequence=1&isAllowed=y

Appendix 1. Produced Graphic Assets

Figure 1. Dragon’s Head

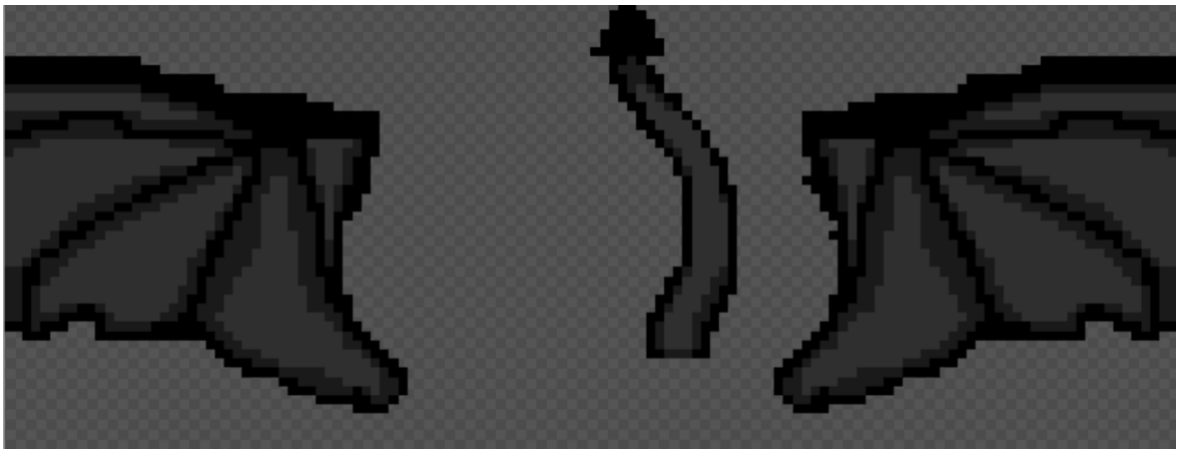

Figure 2. Dragon’s Tail and Wings

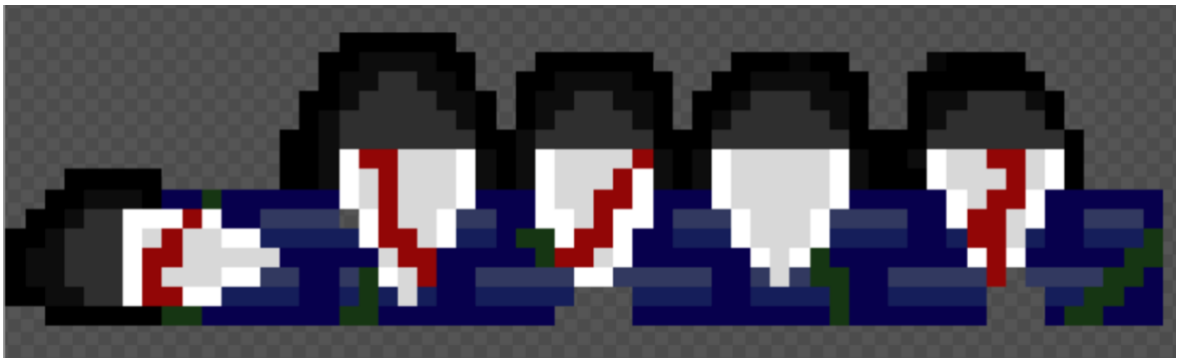

Figure 3. Dragon’s Left Claw

Figure 4. Dragon’s Right Claw

Figure 5. Platform Variant 1

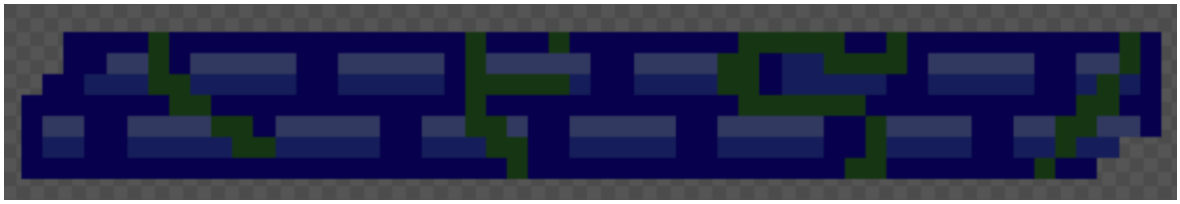

Figure 6. Platform Variant 2

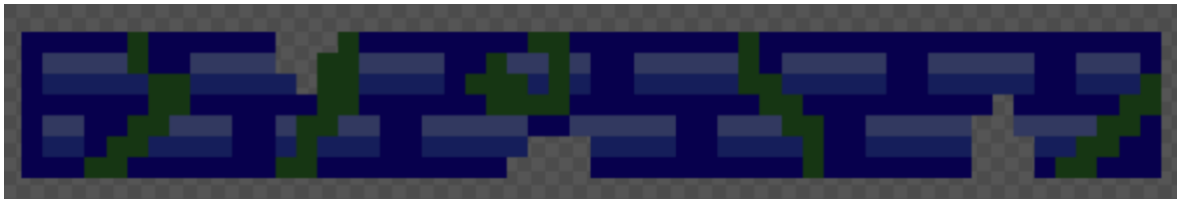

Figure 7. Platform Variant 3

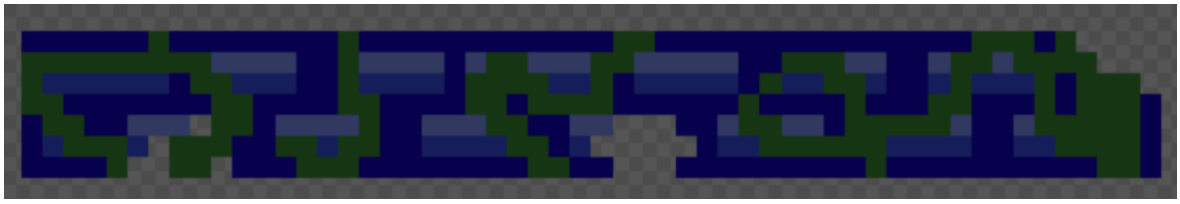

Figure 8. Platform Variant 4

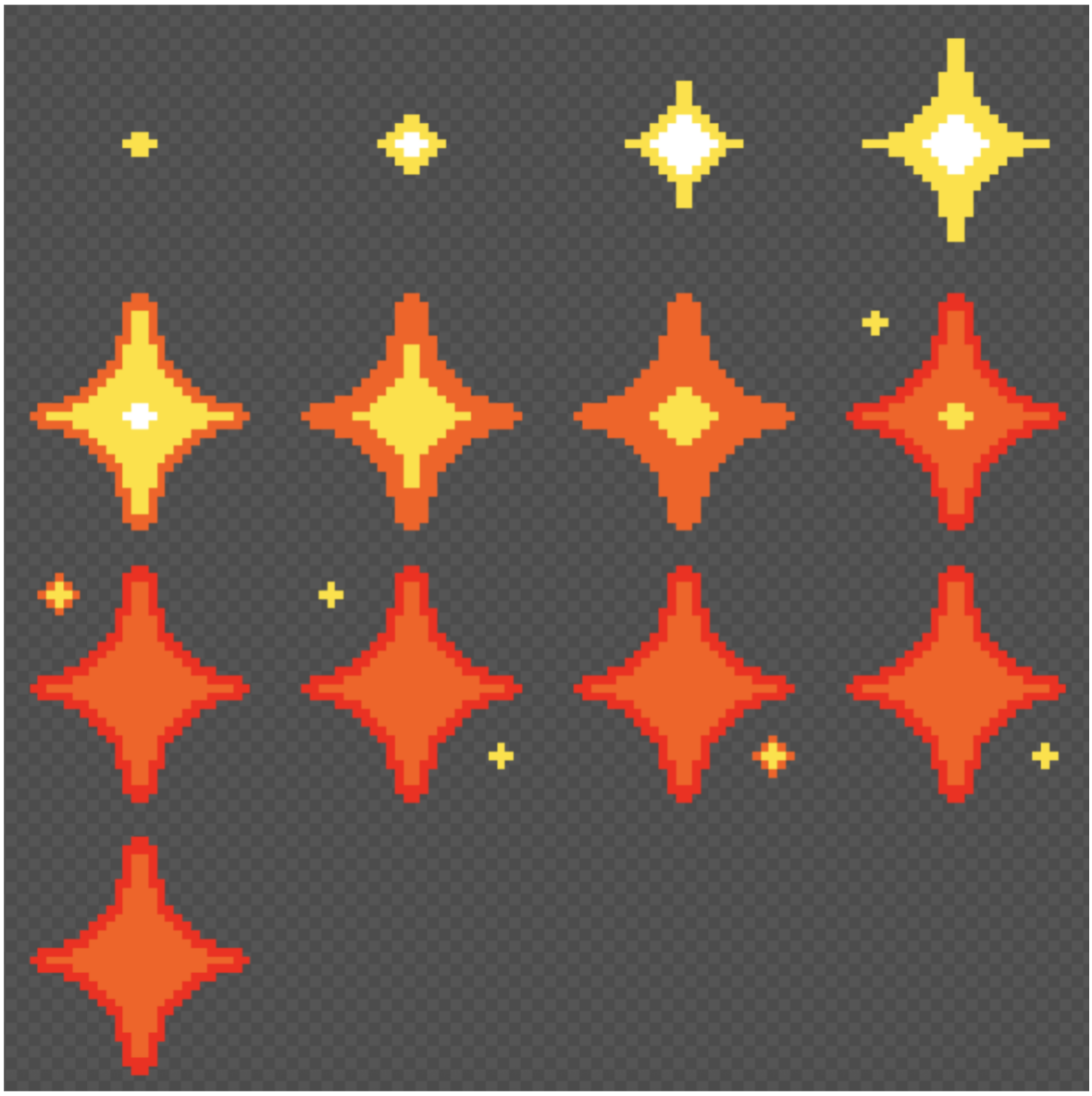

Figure 9. Spark Animation

Figure 10. Tower Door

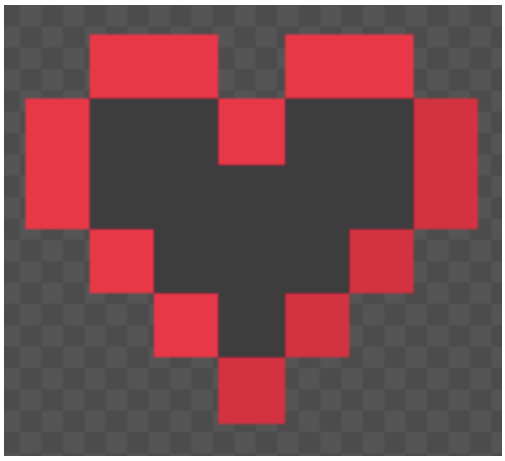

Figure 11. Empty Heart

Appendix 2. Gameplay Screenshots

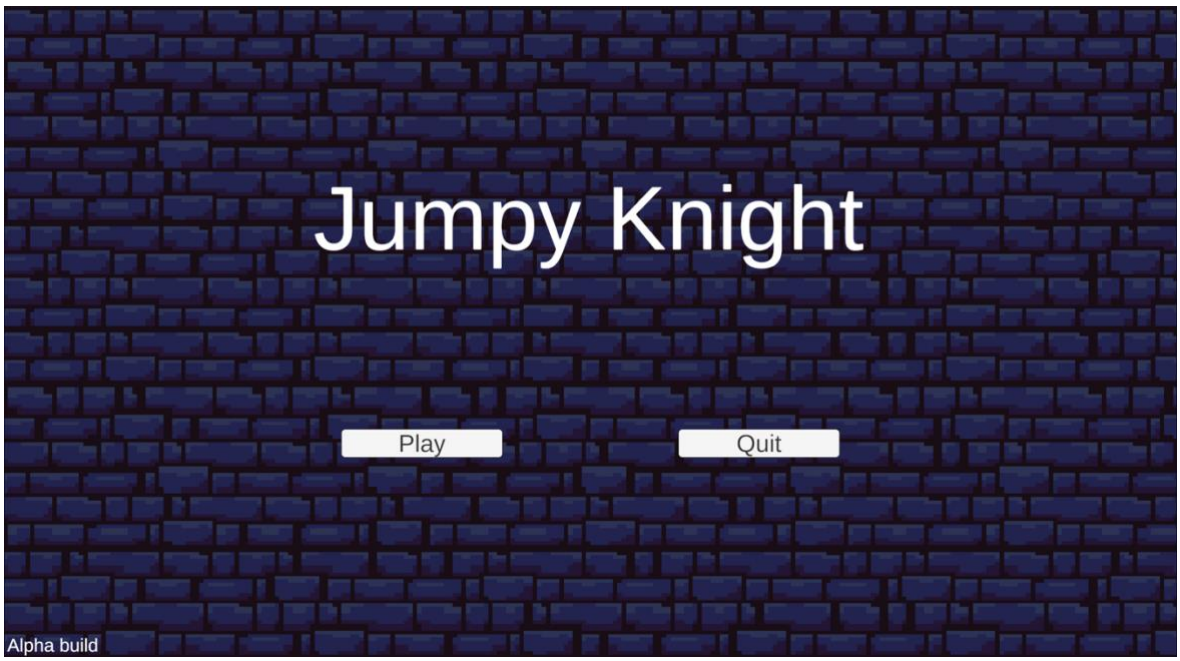

Figure 1. Main Menu

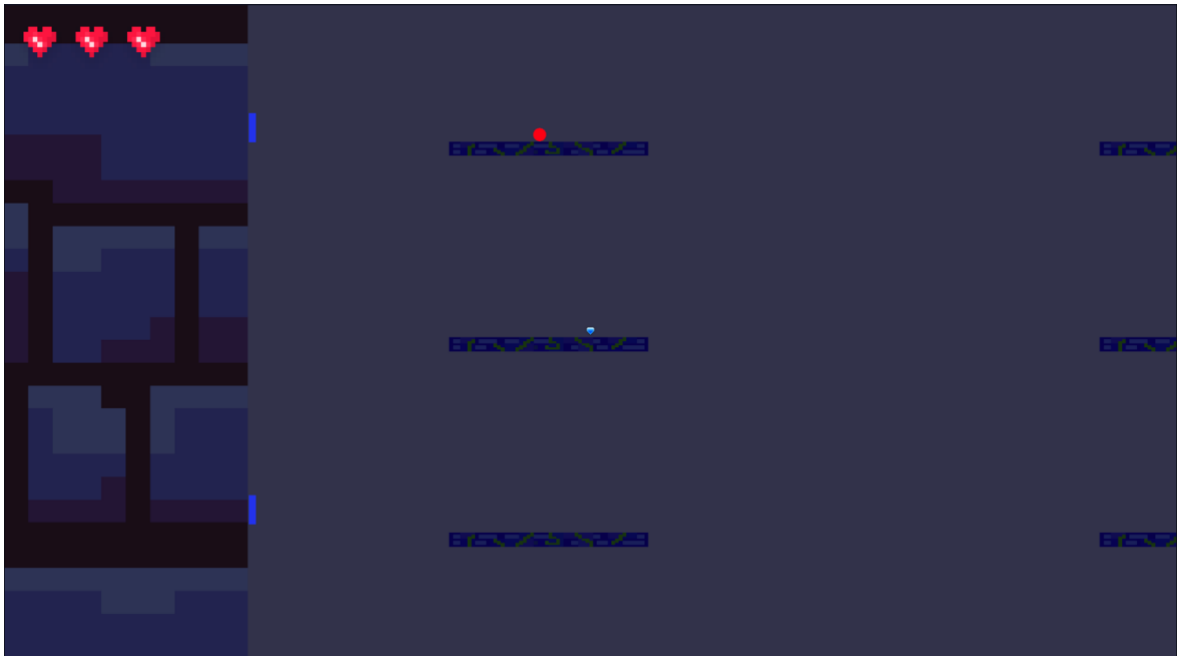

Figure 2. Enemies

Figure 3. Death Screen

Figure 4. Top of the Tower

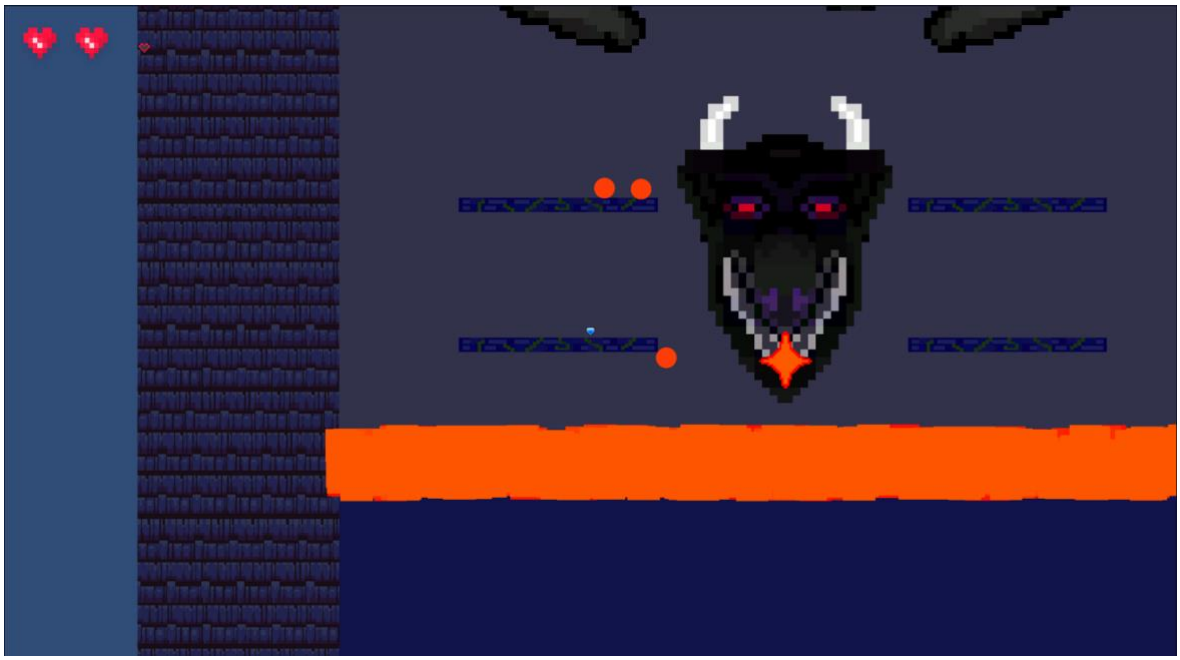

Figure 5. Boss Battle First Phase

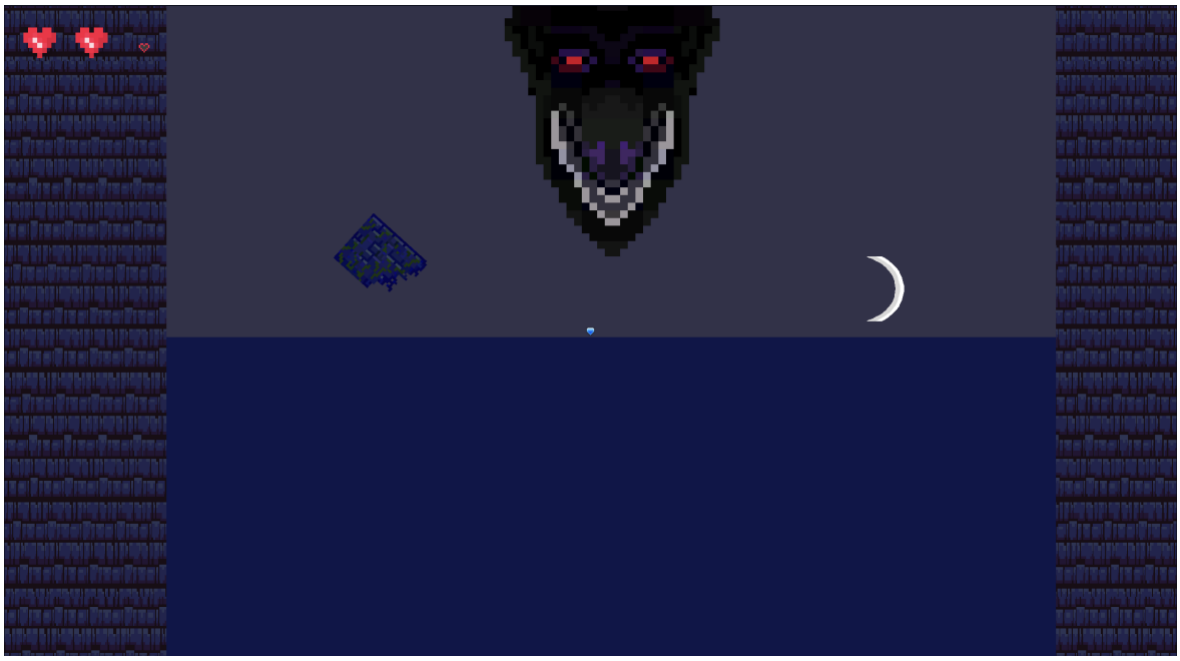

Figure 6. Boss Battle Second Phase

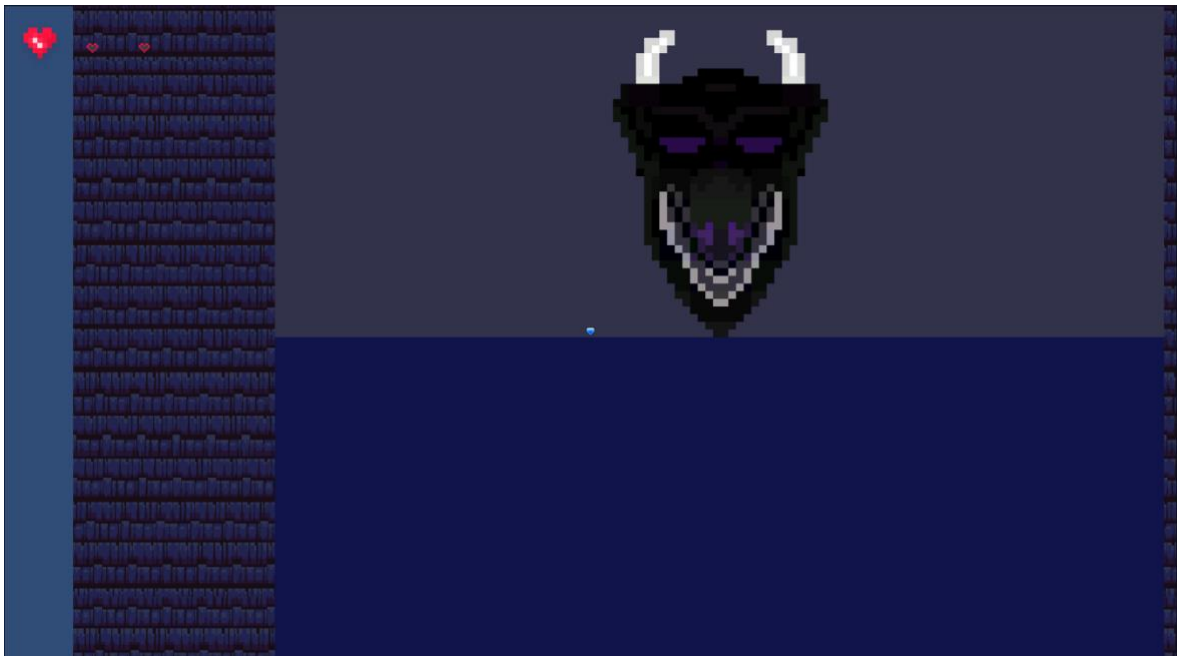

Figure 7. Final Blow

Figure 8. Victory Screen

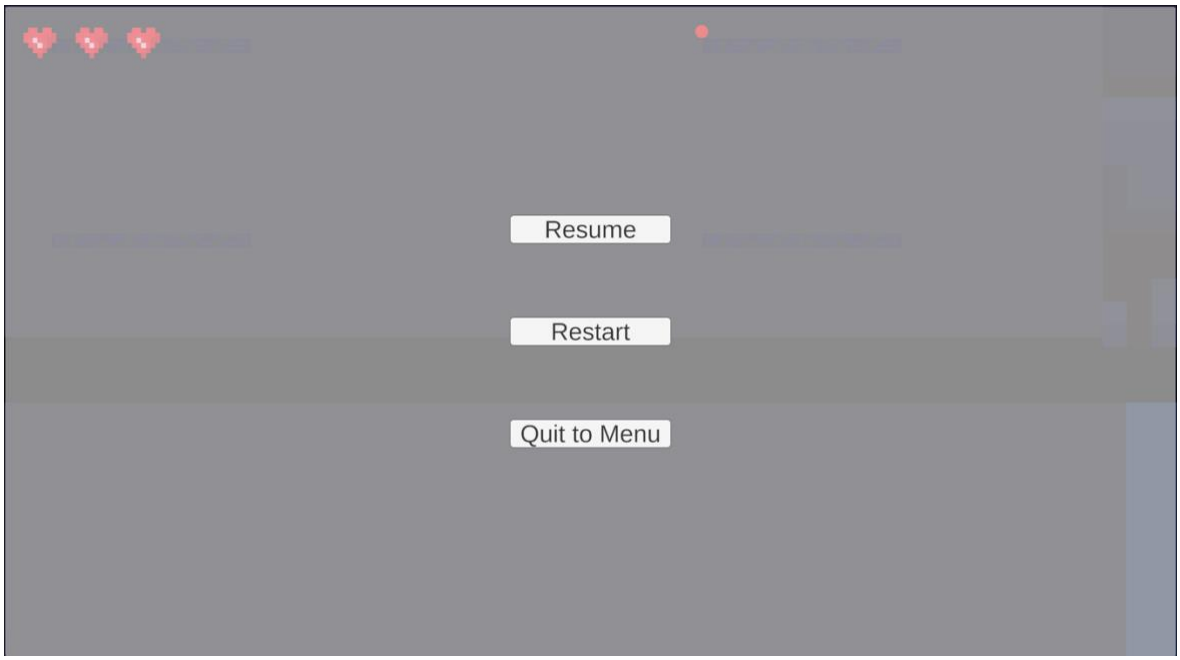

Figure 9. Pause Menu