8-Bit Platform Gaming

Final Year Project - BSc in Computer Science

Project Author - **Mark Daly**

Supervisor - **Dr. Jason Quinlan**

Department of Computer Science

UCC

University College Cork, Ireland

Coláiste na hOllscoile Corcaigh

25th April 2022

Abstract

Video games have always been a great way to form bonds with new people and bring everyone together in a fun, carefree environment. It can be easy to forget that there is a lot that goes into the creation of a game behind the scenes before we get to play it. This project was to learn about this process and to develop a substantial scrolling platform game that could be used as an interactive gaming experience for TY students, first year CS students and other visitors.

The Godot (game) Engine was used to develop the game from the original concept to the final release, a much more difficult process than what was anticipated in the early stages of the project. The goal was to gain valuable game development skills and experience. To learn everything that goes into the creation of a game, the different technologies that are involved, the architecture or how it is built and the design and animation.

Declaration of Originality

In signing this declaration, you are confirming, in writing, that the submitted work is entirely your own original work, except where clearly attributed otherwise, and that it has not been submitted partly or wholly for any other educational award.

I hereby declare that:

- this is all my own work, unless clearly indicated otherwise, with full and proper accreditation;
- with respect to my own work: none of it has been submitted at any educational institution contributing in any way to an educational award;
- with respect to another's work: all text, diagrams, code, or ideas, whether verbatim, paraphrased or otherwise modified or adapted, have been duly attributed to the source in a scholarly manner, whether from books, papers, lecture notes or any other student's work, whether published or unpublished, electronically or in print.

Signed: Mark Daly

Date: 25th April 2022

Acknowledgements

First and foremost, I would like to thank Dr. Jason Quinlan for agreeing to supervise my project and take me under his wing. He was always there to offer guidance and support when it was most needed. He also did a good job in setting his students straight if he believed they were lagging behind. In times of struggle, I would push through because I didn't want to let him down.

I'm also grateful to all the other lecturers for their efforts over the past four years, especially during the worst of the pandemic when morale was at its lowest. They continued to make themselves available and provide any extra help that was requested. A lot of the time, their hard work and support doesn't get enough recognition.

Thank you to all my friends for the best college experience I could have asked for. Covid-19 only brought us closer and inspired us to share a house together for our final year. They have always reassured me in times of self-doubt.

Finally, I give special thanks to my family, I'll be forever in my parents' debt for their continuous support and would not have gotten anywhere without them. Hopefully this will have made them proud.

Table of Contents

Table of Figures

1 Introduction

1.1 Purpose

This project is to develop a substantial scrolling platform game that can be used as an interactive gaming experience for secondary school students, first year Computer Science students and other visitors. The aim is to showcase an example of what can be achieved from doing Computer Science in UCC. An effective way to grab the attention of a younger crowd was in the form of a game. It may spark interest in secondary school students considering a future in technology to enrol in the course or inspire first year students to stick with it in times of struggle.

From first year in this course, some valuable programming skills are taught, predominantly with Python, in CS1117: Introduction to Programming. It is when a comprehension of some of the core programming constructs is gained and the ability to design and write computer programs at a basic level is developed. The intention is that this project will show how to use some of these skills, that everybody learns from the very beginning, to develop something fun and enjoyable such as a side-scrolling game.

Video games have always been a great way to form bonds with new people and bring everyone together in a fun, carefree environment. It can be easy to forget that there is a lot that goes into the creation of a game behind the scenes before it is played. Since game development is a subject that is not looked at in this course, a big part of this project was to learn everything about this process and what is involved.

Generally speaking, delving into any new skillset comes with a steep learning curve. Hence, there is no surprise that this is also true when your goal is to develop your first game. The project needed to be something a complete stranger to game development would be capable of creating, yet still impressive enough to strike up a decent interest. For this reason, the decision was made that it would be an 8-bit platform game.

1.2 **What is 8-Bit?**

First and foremost, a bit is a single binary digit (0 or 1), that can be arranged alongside other bits to represent information that is processed by a computer. Data must be encoded into binary so that computers can understand and perform actions based on the inputted data. By itself, one bit can represent true and false values (or on and off states), 1 being true.

- Therefore, we say that one bit can store 2 different values (2^1).
- Two bits can store 4 different values (2^2) i.e., 00, 01, 10, and 11.
- Four bits can store 16 different values (2^4).
- Eight bits (1 byte) can store 256 different values (2^8).

A computer that has an 8-bit processor means that the CPU's internal data registers can only hold and process data/instructions in blocks of 8 bits at a time. It is also known as the word size (or word length) of a computer system.

Computers are always limited to the processor technology of the time that they are invented. From the early 1970s to late 1980s, they were limited to using an 8-bit processor. This was a hardware limitation or bottleneck that the corresponding software had to conform with, resulting in slow computation time (e.g., 16-bit processors can perform much larger number of operations per second compared to 8-bit processors) and pixelated graphics since graphics processors could only display a maximum of 256 colours (2^8).

Figure 1.1: Super Mario Bros. (NES)

Figure 1.2: Sonic the Hedgehog (Master System)

8-bit powered game consoles were the first of their kind and have laid the foundations of the game industry. It was decided that this project would have similar graphics to games that were developed for 8-bit consoles such as the Nintendo Entertainment System (NES) and Sega Master System. These

consoles were released in this time period, where the appearance of games was limited to processor hardware. However, many developers today still create games that are similar in appearance as a stylistic choice. This will be covered in further detail in the next section. It was determined that using a more simplistic style for this project would be a wise choice for a beginner trying to create their first game in the space of a few months.

1.3 **Objectives**

At the end of this project, one would hope to have gained valuable game development skills and experience that can be used in the future as a creative hobby or possibly as a full-time career. It would be wonderful to have a physical game to be able to show off and be proud of. However, there were many problems that stood in the way before we could get even close to that point.

This project will attempt to explore everything that goes into the creation of a game from scratch. This includes researching the different technologies that are involved, the architecture or how it is built, how it is developed, and the design and animation i.e., what the game looks like and how it is played. Throughout this educational process, newly acquired knowledge and skills will be used to develop an entirely new project in itself, with the hope that it will be complete and playable by others (without any bugs).

It is likely that many crossroads will be encountered where the implementation decisions on elements such as architecture and design will have to be made. The resulting outcome of these decisions will be discussed on completion of the project.

1.4 **Report Outline**

This report goes through each stage of the game development process.

In Chapter 2, an insight to the history of retro gaming will be given to the reader, while covering where the idea for the game originated. It will also go into the level of planning that was involved in the project.

In Chapter 3, the project architecture choices will be discussed and the impact it had on the project, covering the different ways you can build a game and the game engine that was chosen for development.

In Chapter 4, the game physics and code will be explained. It will also cover the details of how different features are implemented in the game.

In Chapter 5, the game design will be discussed. This includes the thought process that goes into deciding what the game looks like and how it is played.

Chapter 6 will conclude the report by analysing the results, providing any final thoughts, and if there is further ambition to continue the project in the future given a deeper understanding of what is required.

2 Background

In this chapter we will cover the early history of retro gaming and the 8-bit limitations that are responsible for the iconic graphics we know today. This will be followed by exploring where the idea for the game originated as well as planning for the work that ensued in the early stages of the developmental process.

2.1 History

The history of video games goes all the way back to the 1950s and 1960s, and there are now nine different generations of video game consoles. Only the details of generation one to three will be covered as that is what's relevant to this project. The complete history of video games up to the present day, can be read about on Wikipedia.

It all started when computer scientists began designing simple games and simulations on minicomputers and mainframes. Spacewar! was developed by MIT student hobbyists in 1962 as one of the first such games on a video display. It wasn't until 1972 that the first generation of video game consoles began with the arrival of the Magnavox Odyssey. This consumer console had no processor or memory, so it didn't have the ability to display colour, sound or scoring. It was this same year that the Intel 8008 became the world's first 8-bit microprocessor. The Odyssey had a table tennis game which inspired Atari to make the classic arcade game Pong. It was also the era of other popular arcade games Pacman and Space Invaders.

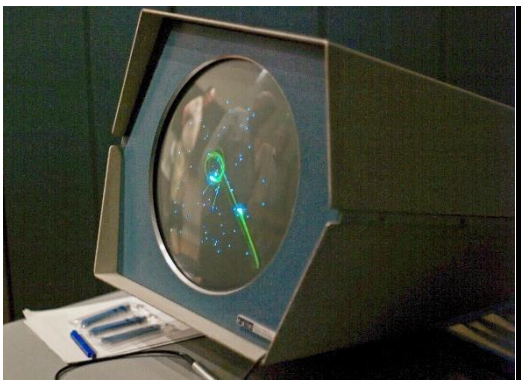


Figure 2.1: Spacewar!

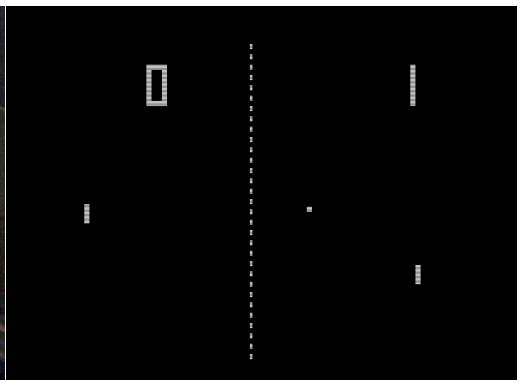


Figure 2.2: Atari's Pong

By 1975, the manufacturing costs of microprocessors had lowered enough to make them a considerable choice for programmable consoles that could load in game software through swappable media. The Fairchild Channel F, released in 1976, was the first home console to use programmable ROM cartridges (Read Only Memory chips encased in plastic). This enabled players to swap in and out different games that were programmed for the device. The microprocessor would then read the instructions from the inserted ROM cartridge. This sparked a new generation of consoles which would unfortunately lead to Fairchild's exit from the market entirely.

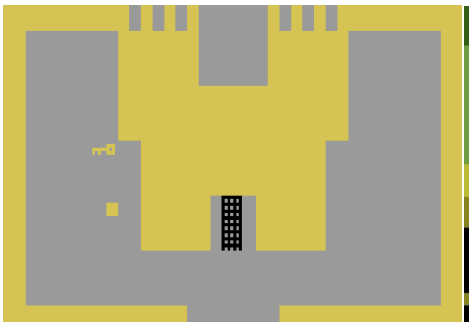


Figure 2.3: Adventure (Atari)

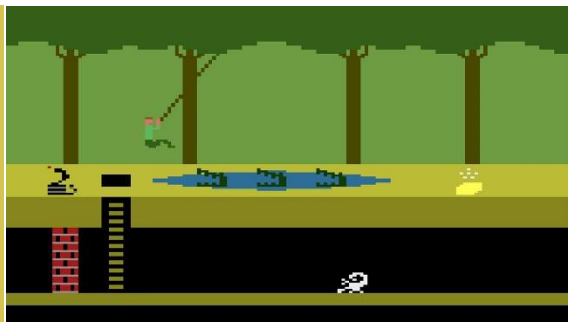


Figure 2.4: Pitfall! (Activision)

Atari and Magnavox came out with their own consoles in 1977. The Atari 2600 was a huge success and had many third-party development firms, including Activision, who would pay licence fees to create games for the console. At the end of 1983, the market was oversaturated with poor-quality games and Atari's loss of publishing control led to a huge video game crash which bankrupted many companies in the industry. The $3 billion U.S. market dropped to $100 million by 1985.

The Japan video game sector, however, was not affected. The third generation of video consoles began that same year when Nintendo released the NES, which entered the US market two years later along with the Sega Master System in 1985.

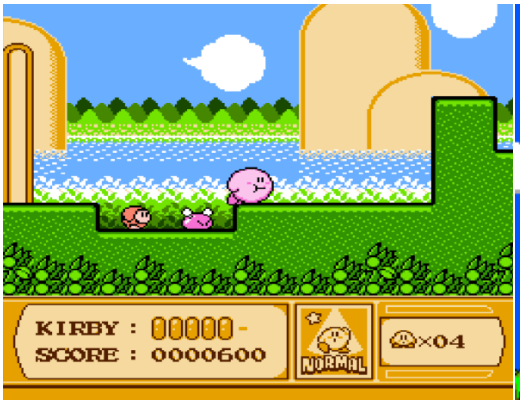


Figure 2.5: Kirby's Adventure (NES)

Figure 2.6: Psycho Fox (Master System)

These consoles and their successors (the SNES and Sega Mega Drive) are revered and regarded as icons of the retro gaming era. Still, in this day and age, 8-bit graphics and audio are being employed to create new games that operate on modern hardware. The only difference is that there are no longer any limits to the colours that can be displayed. Now, pixel art is used to mimic these 8-bit visuals and invoke a sense of nostalgia.

2.2 Hardware Limitation vs Stylistic Choice

The 8-bit consoles mentioned above had much less storage to work with than consoles being developed today (e.g., the PS5 has a 256-bit memory interface). There were some extreme hardware limitations to the graphics that could be displayed. There are many examples that can be found of 8-bit games that contain these restrictions.

Those skilled enough to make it to the 256th level of Pac-Man (known as the "spit-screen level" or "kill screen") will know that it is impossible to beat. The game would essentially crash and turn into a jumble of letters and ghosts. Link couldn't loot more than 255 rupees in the original Legend of Zelda because it is the maximum value an unsigned 8-bit value can hold.

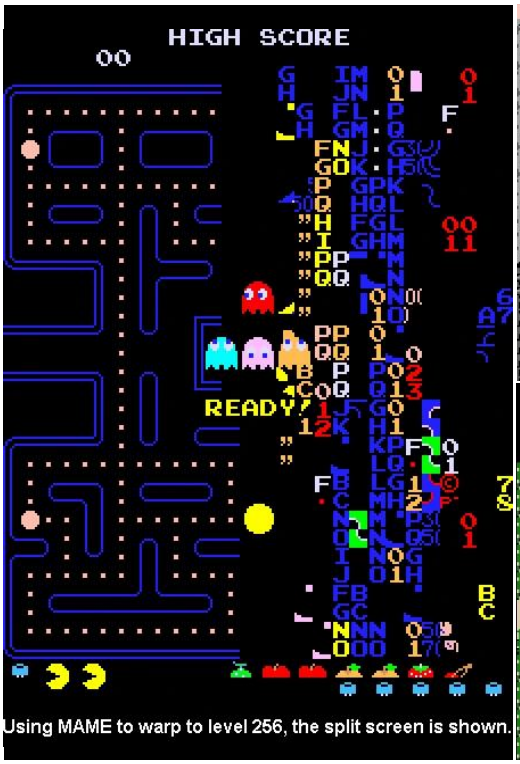


Figure 2.7: Pacman Level 256

Figure 2.8: The Legend of Zelda Rupee Limit

But these 8-bit limitations were also responsible for the iconic graphics we know today. The strict rules development rules required to run a game on the NES was like making an artist paint with one hand tied behind their back. The console's picture-processing unit (PPU) split what's displayed on screen into two layers, the background and sprite later.

The background layer is split into 968 8-by-8 tiles. Developers were only allowed 256 unique color tiles to be on the screen at once so repetition was key and it could only show a maximum of 64 colors but only 13 at one time. The sprite layer, or the layer where the protagonist and enemies would move around in, had its own complex set of rules as well.

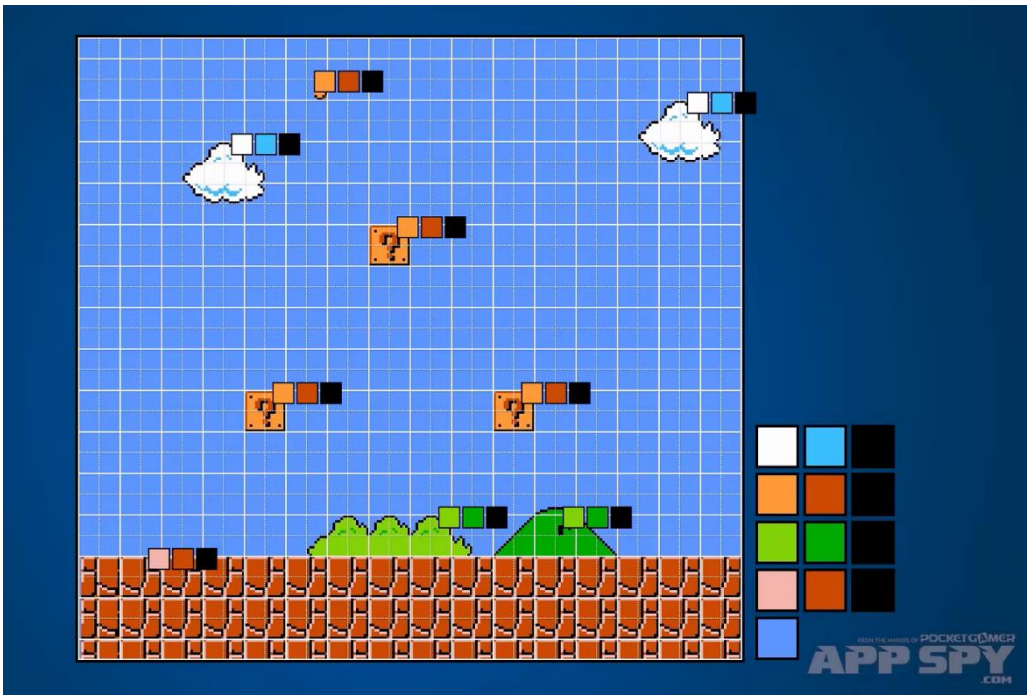


Figure 2.9: Mario Bros. (NES) Background Layer

The pixelated visual style invented out of necessity back in the 1980s is still relevant today. The NES was the definitive console of its generation, so much so that it inadvertently launched an "8-BIT" game style that modern developers still use despite having nothing to do with 8-bit consoles.

Indie games (independent video games) developed by smaller teams without the financial and technical support of large game publishers. Like Shovel Knight and Owlboy fall within the realm of "8-bit inspired" games because of their pixelated graphics and glitchy soundtracks. However, these tiles are far more advanced than anything the retro NES developers had the ability to create at the time.

Figure 2.10: Shovel Knight

With the PlayStation 5 and Xbox Series X, 4K graphics are now the home console standard, pixelated games still have a cult following. Titles like RPG Undertale are often described as "8-bit retro" but these games are only 8-bit in appearance, not in soul.

they weren't made with the 8-bit limitations that early developers operated under where the look of the game was a necessity more than a stylistic choice. For example, *Shovel Knight* might seem like it could run on an NES, but its developers Yacht Club Games had access to colors that weren't available on the retro console and Owlboy took ten years to make.

Figure 2.11: Owlboy

2.3 **Inception**

All games begin with an idea. To gather inspiration, a couple of emulators were downloaded and used to simulate retro consoles. An emulator is a computer program made to simulate a gaming console. Basically, it as an app that lets you play NES and Master System games on your computer.

A selection of games, called ROMs, were also downloaded for each device. A ROM file is just a copy of the original game that come in many different file formats, compatible to the emulator of their original console. Only a certain emulator can open it. Although technically ROM stands for Read Only Memory, these game files were given this term because it is an exact replica of the memory on the original cartridge.

It was a fun research assignment playing the games shown in previous sections as well as other side-scrolling platformers similar to the desired outcome. Another game called Contra gave inspiration to let the character sprites carry guns to shoot enemies with. It was found that games that had a two-player cooperative mode was more enjoyable.

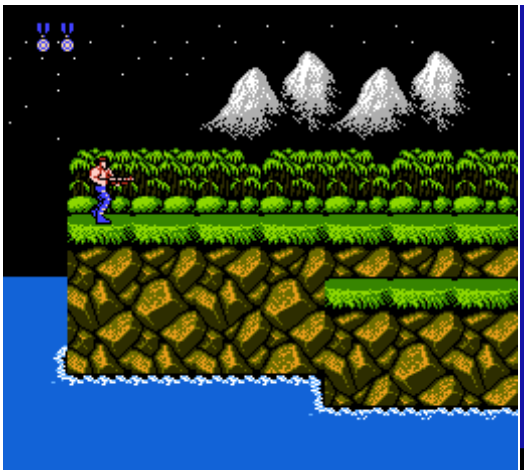


Figure 2.12: Contra (NES)

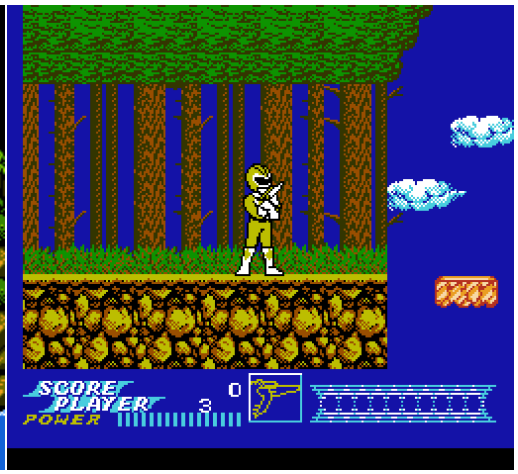


Figure 2.13: Power Rangers 2 (NES)

After quite some brainstorming and elaboration the following idea was settled for. The game would be called Man v. Martian. The player would be a men in black agent that has been kidnapped by extra-terrestrial beings and needs to escape from captivity. By good fortune, the agent is equipped with a high-tech plasma gun to fire at alien enemies. You may also partner up with a second agent in two-player mode which will make the game an even more enjoyable experience for all those who get a chance to play it.

It is important to elaborate like so in this phase because usually there would be a team of workers in different departments, e.g., the game programmers would start implementing the features and game designers would start creating the visuals. However, the main finding here was that it was valuable for this project to have a strong core concept to look back on as a constant reminder of the end goal.

2.4 **Planning**

Now that the seed had been sown, a plan had to be put in place to transform this idea into a reality. The project scope was defined to set out the main objectives and deliverables such as the different features that needed to be implemented. This included the number of stages/levels in the game that were desired, a two-player mode, and in game assets such as character sprites and music. Some tolerance was also given so that if the vision turned out to be too ambitious then there were options to scale the deliverables down. The number of levels could always be reduced, or free assets available online could be availed of instead of learning how to create your own.

The scope also highlighted the amount of time to complete it which was roughly 20 weeks of the academic year. With this time limit came some constraints and assumptions, e.g., it was determined that the game would only be deployed on computers with a desktop environment (i.e., Windows/Linux/macOS computers). The primary focus was on the fundamentals of game development, availability to a larger audience using a wide range of devices would not be considered. There were also some risks involved that were addressed that could take away some valuable time from the project. Unforeseeable events such as illness and other modules where the level of attention that is required fluctuates on a weekly basis.

It was found that there were many benefits to this process. Primarily, it began to clarify what the project entailed, or the number of tasks involved. A lot of research on the process behind the creation of a game was required before being able to define any kind of a scope. Writing out a plan to describe what the game will and will not have helps to prevent the project diverting from what was originally intended (scope creep). This tends to happen on longer projects like this one where you end up spending time on tasks that were not expected at the beginning. This usually leads to project delays or in this case, where there is a hard deadline, a disappointing low-quality game. Not to mention how undesirable it is to add unnecessary stress or burnout from overwork.

Coincidentally, this is actually very common in the video game industry where there is a crunch or compulsory overtime to compensate for failing to produce the deliverables needed to launch a game on time. It was found that this planning phase of the project served as a guideline to start scheduling tasks. From the scope, a Gantt chart was formulated to show a rough plan for the upcoming weeks. This helped to try keep the project on track throughout the project.

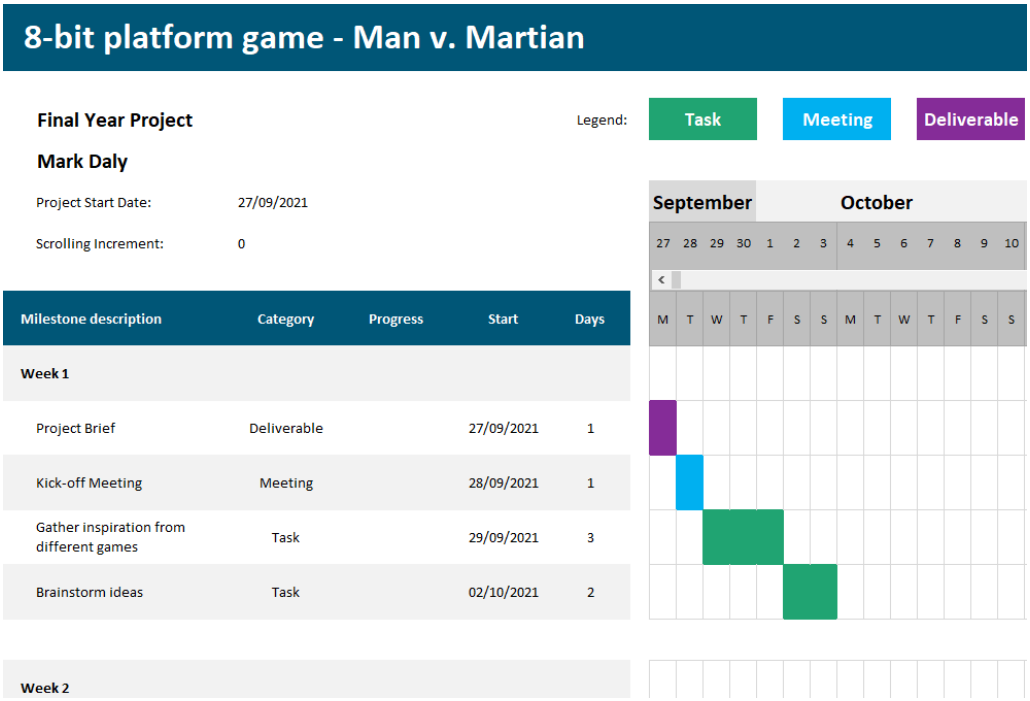


Figure 2.14: Gantt Chart

3 Architecture

In this section, the main architectural and implementation decisions of the project will be explored how they influenced other aspects of development. This will include an evaluation of the different frameworks that were considered, the software that was ultimately chosen and the vital role it had throughout the project.

3.1 Python Frameworks

Python is a great language to get started in game development. As mentioned previously, delving into any new skillset generally comes with a steep learning curve. Initially, this can be intimidating so it is beneficial to make the process as easy as possible for yourself (baby steps). Python, being known for its simplicity and ease of readability, is a fitting language for beginners in game development to use as their gateway into the industry. There are also lots of developers and forums available online to guide you through any difficulties.

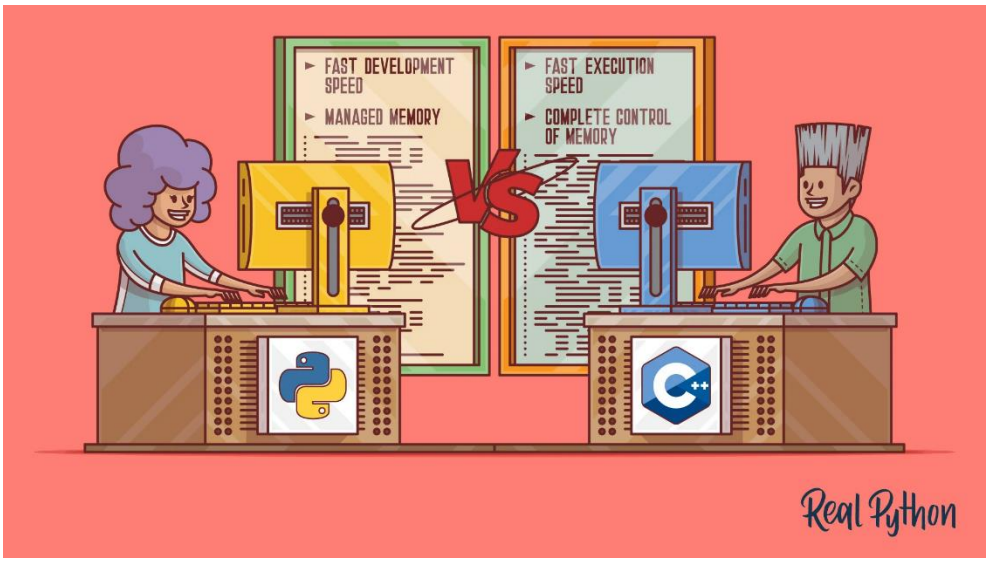


Figure 3.1: Python vs C++

In the gaming industry, Python is mostly used for rapid prototyping of games. Since it has some limits with performance, the industry standard is C# with

Unity or C++ with Unreal Engine (for games that are more resource intensive). These are robust development platforms with every tool you could possibly need and have been used to create many popular game titles e.g., Fortnite (Unreal) and Rust (Unity).

Figure 3.2: Unity

Figure 3.3: Unreal Engine

Since Python is better suited for hobby projects, prototyping ideas for a game, and understanding the concepts of game development, it is the better choice for this project. Contrary to C# or C++, there aren't as many games engines that use Python as their scripting language. I will compare some of the more popular ones below, all of which are open-source and free to use.

3.1.1 **Pygame**

Pygame is a cross-platform set of Python modules used to create 2D video games. It uses the Simple DirectMedia Layer (SDL) library. These modules provide access to specific hardware components, such as sound, video, mouse, keyboard, and joystick. Some of the main features of Pygame include sprite and scene graph management, collision detection, pixel-array manipulation, MIDI support (music), drawing and FreeType font support.

Figure 3.4: Pygame

It supports Linux, Windows, macOS and BSDs. Applications can also be run on Android devices with the use of the Pygame Subset for Android. All you will need is a code editor like Visual Studio Code or you can choose an IDE like JetBrains PyCharm.

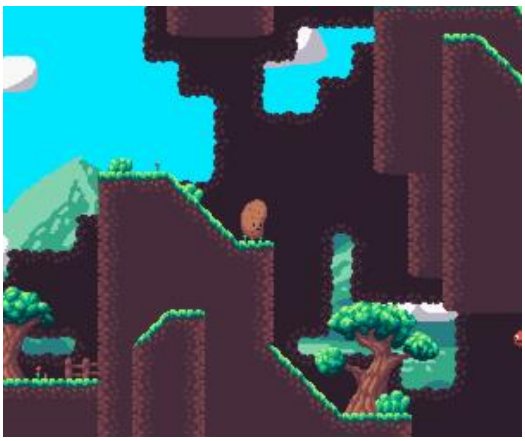


Figure 3.5: Super Potato Bruh (Pygame)

Figure 3.6: Rambit (Pygame)

Advantages

- It comes with a straightforward API that is easy to understand.
- There are a wealth of tutorials and examples available online, on top of the API documentation so you have all the support you need to get started.
- It is available to all users as it is portable to almost any OS.
- Its drawing system allows users to draw on multiple canvases.

Disadvantages

- The developer community is non-existent. There aren't really any forums available to reach other Pygame developers online.
- A lot of the modules in Pygame are outdated and inefficient. Since it uses the first version of SDL, it is missing some new and improved features developed for SDL2.
- Good for developing smaller games but has performance issues when it comes to larger projects.

- Since it is not a complete game engine, some rendering, physics and mathematical operations may not be supported. It is quite basic (just a Python wrapper for SDL).
- The documentation is messy and can be very confusing as to how some functions work.

3.1.2 **Pyglet**

Figure 3.7: Pyglet

Pyglet is another cross-platform multimedia library for Python used to create games and other visually rich applications. Some features that are supported include user interface event handling, OpenGL graphics, joystick, loading images/videos and playing sounds/music. It provides built-in support for standard formats such as PNG, WAV, BMP and DDS, offering optional use of FFmpeg to playback additional formats.

It supports Linux, Windows and macOS. There is no support for mobile devices because there is no integration of OpenGL. Again, all you need to do is install the Pyglet library and choose a code editor to start developing your game.

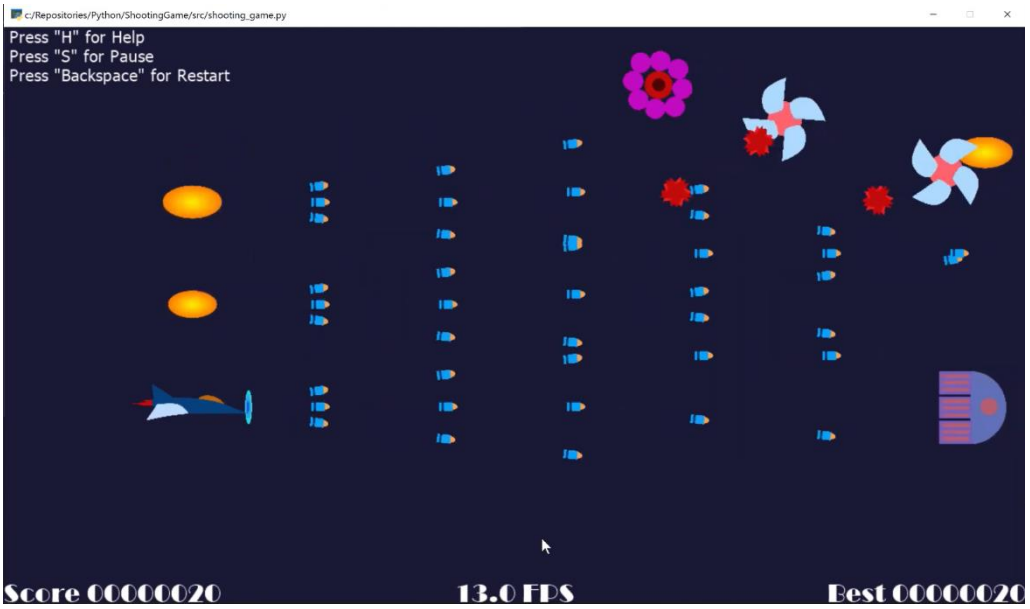


Figure 3.8: Pyglet Shooting Game

Advantages

- Very friendly Pythonic API makes it easier to learn.
- Pyglet is tightly woven with OpenGL, so 3D drawing is supported (not useful for this project).
- Good desktop platform support.

Disadvantages

- No integration of OpenGL exists so mobile platforms are not supported.
- Since it isn't very popular, there is little community support.

3.1.3 **Pyxel**

Pyxel is a retro game engine for Python that has the power to make old-fashioned pixel art games easily. It is a much newer game engine (released in 2018) with a growing community. Inspired by retro gaming consoles, it has extremely simple specifications (only 16 colors can be displayed and only 4 sounds can be played back at the same time). Pyxel comes with a special tool for creating pictures and music.

Figure 3.9: Pyxel

As a game engine for Python, Pygame and Pyglet are more popular. However, Pygame is older and slower, Pyglet is difficult to use and has fewer documents. By combining a simple API with a modern design, Pyxel offers a third choice that allows anyone to enjoy game development without compromising the ease and productivity of Python.

Figure 3.10: Pyxel Jump Game

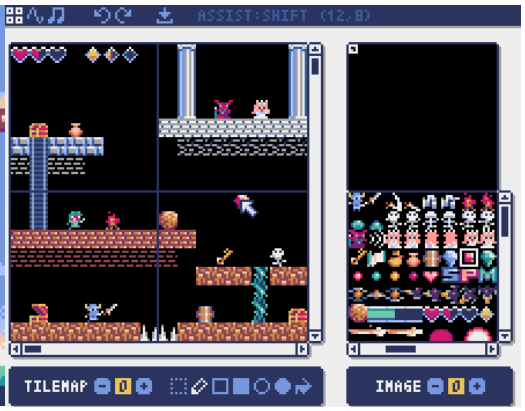


Figure 3.11: Pyxel Tilemap Editor

Advantages

- Modern game engine with simple API
- Specifically designed to create retro games
- Only need to remember a few simple instructions/functions to use
- Runs on Windows, macOS, and Linux

Disadvantages

- A bit too pixelated for this project
- Limitations with scalability

3.2 **Godot Engine**

Godot (pronounced "Go-dot" or "Guh-dough") is a cross-platform, extremely versatile, game engine (high-level framework). It is designed to create both 2D and 3D games targeting PC, mobile, and web platforms. It is free and open source released under the MIT license. Godot aims to offer a fully integrated game development environment. It allows developers to create a game, needing no other tools beyond those used for content creation (visual assets, music, etc.). Godot provides a huge set of common tools, so you can just focus on making your game without reinventing the wheel. Godot comes with a fully dedicated 2D engine. Work in pixels as your units, but scale to any screen size and aspect ratio.

Figure 3.12: Godot

Unlike Unity, Godot's support for 2D processing is far better because it was carefully designed from the start. That allows for greater performance, fewer bugs, and a cleaner overall experience. Unlike most engines which support both 2D and 3D, the 2D is better optimized and bug free, differently from Unity which uses a "fake 2D", as its 2D is just a 3D with 2 dimensions.

dynamically typed programming language which is syntactically like Python. Unlike Python, GDScript is optimized for Godot's scene-based architecture and can specify strict typing of variables. This custom language allows for superior optimization and editor integration to make games with zero hassle.

The development environment runs on multiple operating systems including Linux, BSDs, macOS, and Microsoft Windows. Applications can also be run on Android/iOS mobile platforms and can also be deployed online with HTML5 support.

Figure 3.13: Kingdoms of the Dump (Godot)

Figure 3.14: Gravity Ace (Godot)

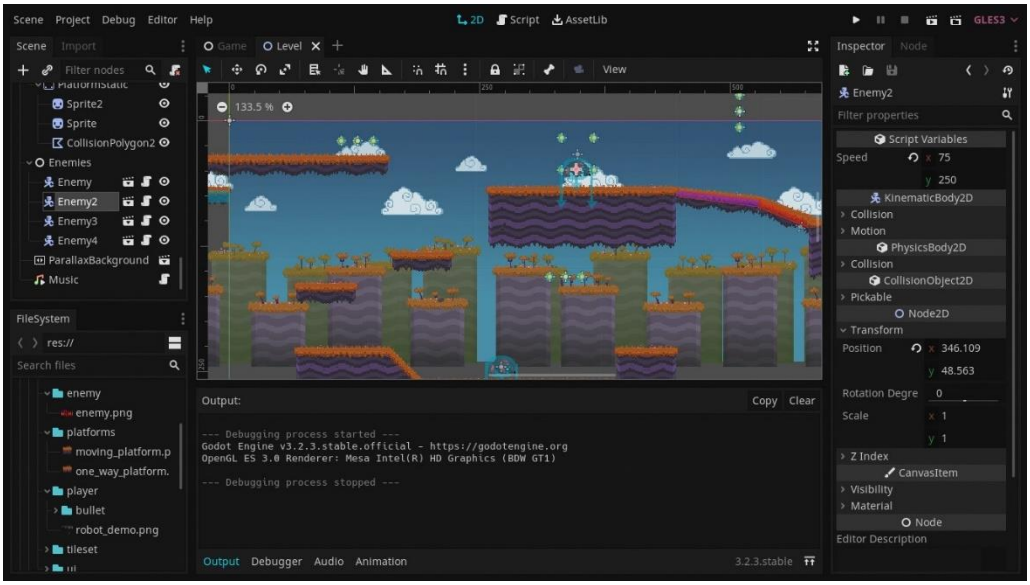


Figure 3.15: Godot User Interface

Advantages

- Live debugging. If you do live editing in any designed game, then after stopping the game, the changes will remain, which means it will not be disturbed.
- Well documented
- New updates every year
- Fast speeds of processing
- Programming language is easy to use
- Games can be deployed everywhere, even game consoles (via third party providers)

Disadvantages

- Learning a new language might be intimidating, but if you know Python you will pick up GDscript very easily
- Different to the other Python frameworks as with Godot you use the built-in editor. Software will take longer to get comfortable with.

It was concluded that the Godot game engine was most suitable for the project at hand. The only drawback was the thousands of features it has for the professional developers that use it. It was crucial to prepare well and get familiar with the software first, instead of diving right in, to avoid feeling out of depth at every obstacle. This would just lead to unwanted frustration and loss of morale and so the preparation process began.

Figure 3.16: Human Diaspora (Godot)

4 Development

In the previous chapter, the architecture that is core to the development of the project was decided on. In this chapter, the main work of the project will be covered, including familiarisation of the game engine and how it was used to implement different features. It will also discuss the main challenges that were encountered, and the methods used to overcome them.

4.1 Fundamentals

Every Godot game is made up of four foundational building blocks: scenes, nodes, the scene tree, and signals. The architecture of the game engine is built from the idea of a tree of nodes which are grouped together in different scenes. A scene could be any object in the game such as a character, their weapon, the user interface. Nodes at the lowest level are what formulate each scene in the game, e.g., a typical character could be formed from a physics body node at the root that has child nodes of an animated sprite, area, collision shape and camera.

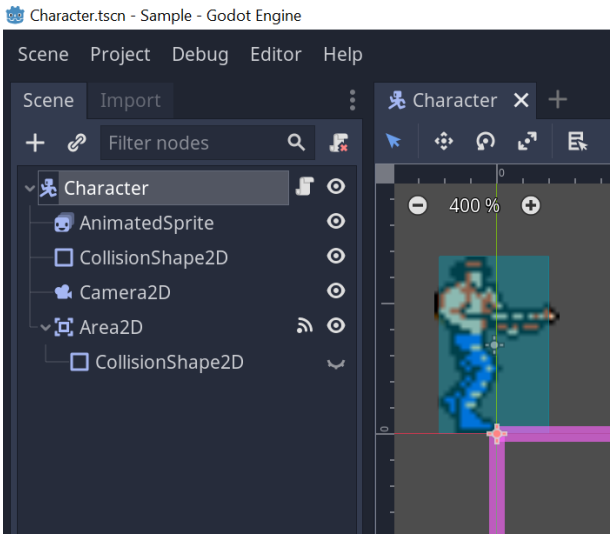


Figure 4.1: Character Scene

The first step was to get familiar with the editor. This was accomplished by learning how to create a scene, add nodes to the scene tree, and attach a script to start coding its functionality in GDScript. It started off with something simple like coding the movement and direction of a sprite. Then the use of signals was discovered to enable different scenes to communicate with each other. In the example below when an enemy enters the area of the player, a signal will be sent to the player to take health damage. This in turn will send a signal to the health bar to update its value. Functions with a green arrow to the left indicate that they are triggered by a signal.

```
1  extends TextureProgress
2
3  func _on_Player_health_changed(new_health):
4      value = new_health
```

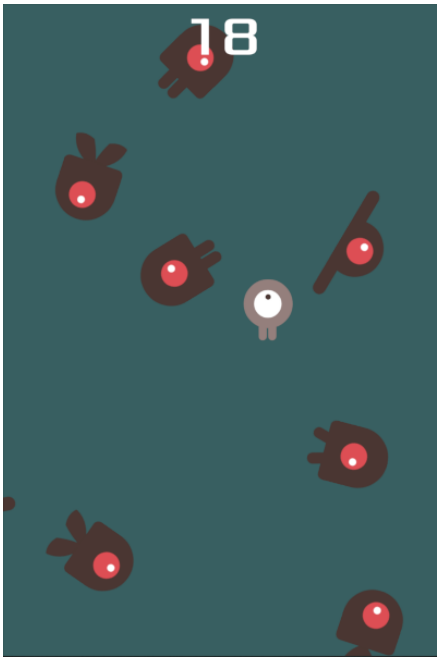
Figure 4.2: Healthbar Script

```
1  extends Area2D
2
3  signal health_changed(new_health)
4
5  var health = 10
6
7
8  func take_damage(amount):
9      health -= amount
10     if health < 0:
11         health = 0
12
13     get_node("AnimationPlayer").play("take_damage")
14     emit_signal("health_changed", health)
15
16
17 func _on_area_entered(area):
18     take_damage(2)
```

Figure 4.3: Player Script

4.2 Dodge the Creeps

Once the fundamentals had been learned, it was time to practice making a game from start to finish. There are some excellent tutorials on YouTube that go through this process. Even providing the in-game assets for you to download on GitHub so that you can follow along.



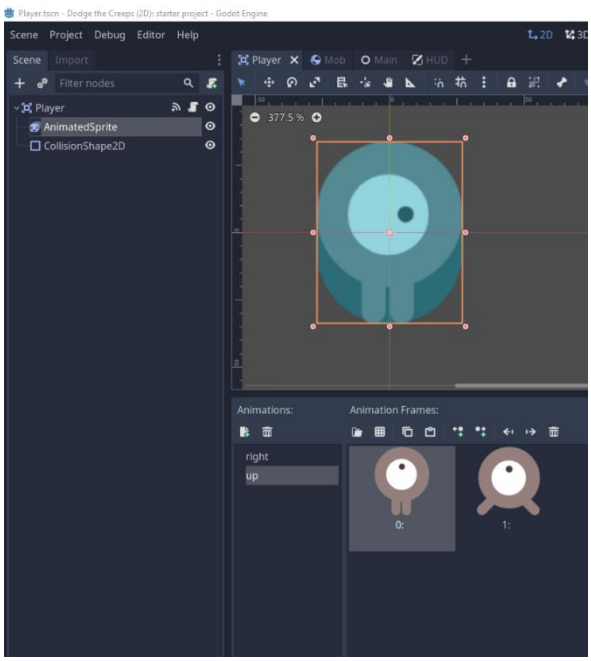


Figure 4.4: Dodge the Creeps Gameplay Figure 4.5: Player scene and AnimatedSprite

The first tutorial was a game called Dodge the Creeps. Many valuable skills of how to build a game from scratch (excluding game design) were learned. The initial step was learning how to move the player and animate the player sprite based on its direction of movement while preventing the player from leaving the screen.

The process or physics_process function (depending on the type of node) is the main function that defines what actions a character will take after input. This function is called every frame. In a game that is running at 60FPS (frames per second), input will be processed roughly every 1/60 of a second. The input to the function, "delta", is the time passed since the previous frame since it is not constant and dependent on a number of factors such as:

- System hardware, such as the graphics card, motherboard, CPU, and memory.
- Graphics and resolution settings within the game.
- How well the game code is optimized and developed for graphics performance.

```gdscript
func _process(delta):
    var direction = Vector2.ZERO # Vector2(0.0, 0.0)
    if Input.is_action_pressed("move_right"):
        direction.x += 1
    if Input.is_action_pressed("move_left"):
        direction.x -= 1
    if Input.is_action_pressed("move_down"):
        direction.y += 1
    if Input.is_action_pressed("move_up"):
        direction.y -= 1

    # normalize direction so player will
    # remain at constant speed if 2 keys are pressed
    if direction.length() > 0: # pressing at least one movement key
        direction = direction.normalized()
        # animate player on input
#       get_node("AnimatedSprite").play()
        $AnimatedSprite.play() # get_node shorthand
    else:
        $AnimatedSprite.stop()

    position += direction * speed * delta # player movement
    # prevent the player from leaving the screen
    position.x = clamp(position.x, 0, screen_size.x)
    position.y = clamp(position.y, 0, screen_size.y)

    if direction.x != 0:
        $AnimatedSprite.animation = "right"
        $AnimatedSprite.flip_h = direction.x < 0 # flip on move_left
        $AnimatedSprite.flip_v = false
    elif direction.y != 0:
        $AnimatedSprite.animation = "up"
        $AnimatedSprite.flip_v = direction.y > 0 # flip on move_down
    else:
        pass
```

Figure 4.6: Player process function

For the enemy, learned how to choose random animation on enemy spawn.

```gdscript
extends RigidBody2D

export var min_speed = 150.0 # px/s
export var max_speed = 250.0

# choose random animation every time an enemy is spawned in the Main scene
func _ready():
    $AnimatedSprite.playing = true
    var mob_types = $AnimatedSprite.frames.get_animation_names()
    # ["fly", "swim", "walk"]
    $AnimatedSprite.animation = mob_types[randi() % mob_types.size()]

func _on_VisibilityNotifier2D_screen_exited():
    queue_free() # delete enemy on screen exit
```

Figure 4.7: Enemy Script

Coding player-enemy interaction in this game was similar to how the health bar worked in the first example. This when it started to feel like less of a slope because the same concepts were being reused. It was also learned how to create the game interface and display different messages temporarily with a timer.

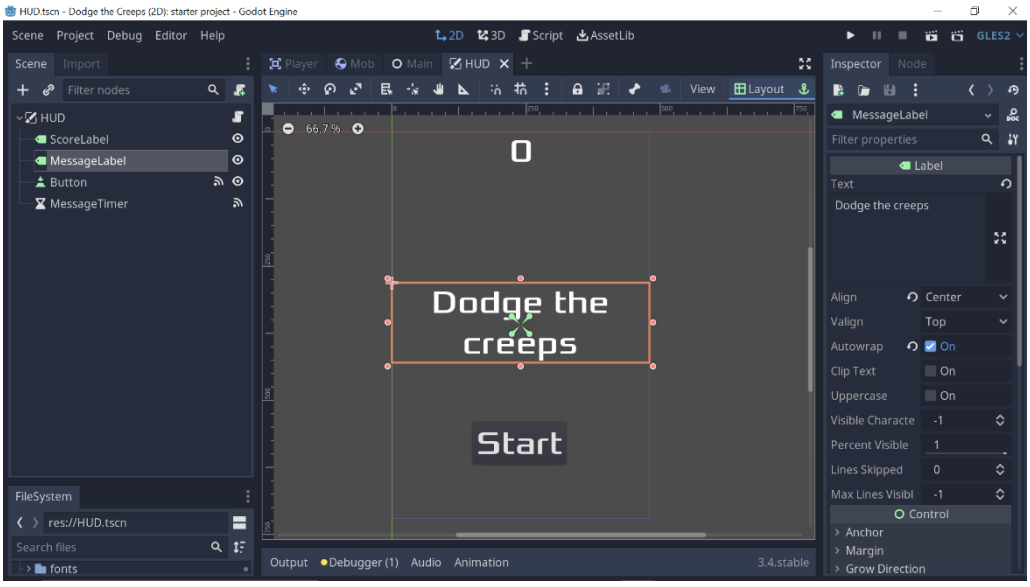


Figure 4.8: Game Interface

It took longer than expected to follow along with the tutorial fully and make sure every concept was understood. At the end there was still a lack of confidence to get started with Man v. Martian. Ultimately, additional practice was needed to get more comfortable with the software.

4.3 Platformer

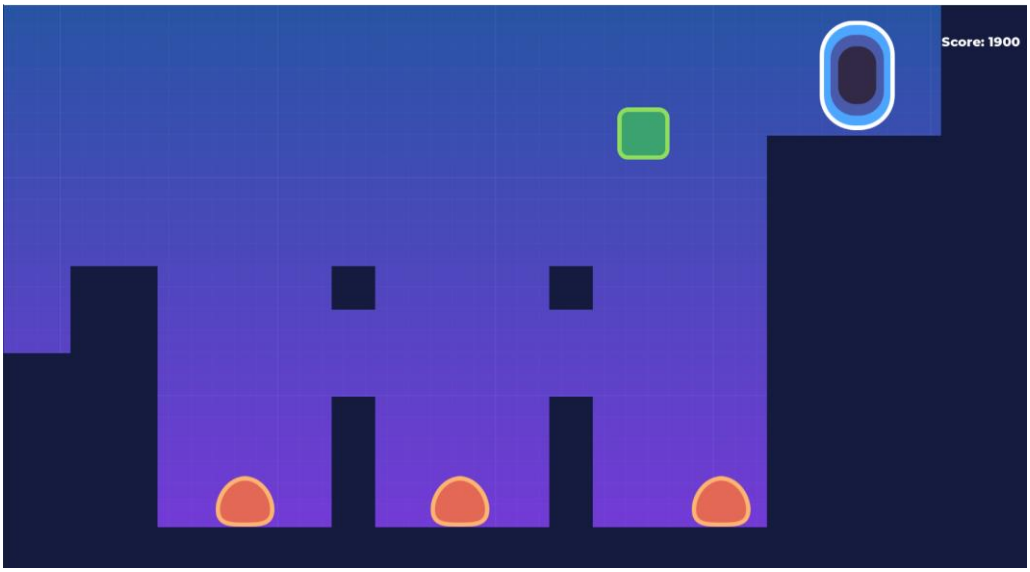


Figure 4.9: Platformer Gameplay

The second tutorial, Platformer (pictured above), was a lot closer in logic to what was trying to be achieved. One obvious example is that there is a floor, so the player needs to have gravity (a downward velocity) to collide with it. Then you must consider the calculation of jump input.

```gdscript
func _physics_process(delta: float) -> void:
    var is_jump_interrupted: = Input.is_action_just_released("jump") and _velocity.y < 0.0
    var direction: = get_direction()
    _velocity = calculate_move_velocity(_velocity, direction, speed, is_jump_interrupted)
    _velocity = move_and_slide(_velocity, FLOOR_NORMAL)

func get_direction() -> Vector2:
    return Vector2(
        Input.get_action_strength("move_right") - Input.get_action_strength("move_left"),
        -1.0 if Input.is_action_just_pressed("jump") and is_on_floor() else 1.0
        )

func calculate_move_velocity(
        linear_velocity: Vector2,
        direction: Vector2,
        speed: Vector2,
        is_jump_interrupted: bool
    ) -> Vector2:
    var out: = linear_velocity
    out.x = speed.x * direction.x # horizontal movement
    out.y += gravity * get_physics_process_delta_time() # vertical movement
    if direction.y == -1.0: # if jump is pressed
        out.y = speed.y * direction.y
    if is_jump_interrupted:
        out.y = 0.0
    return out
```

Figure 4.10: Player Script

Other things that were learned were how to stomp an enemy to death. This was similar to how an enemy would kill a player

and adding a camera that follows the player along the level.

portal

```gdscript
tool # enables display of warnings in the editor
extends Area2D

onready var anim_player: AnimationPlayer = $AnimationPlayer

export var next_scene: PackedScene # next level to teleport to

# teleport when player enters portal area
func _on_body_entered(body: Node) -> void:
    teleport()

# trigger warning if next level not set
func _get_configuration_warning() -> String:
    return "The next scene property can't be empty" if not next_scene else ""

func teleport() -> void:
    anim_player.play("fade_in")
    yield(anim_player, "animation_finished") # wait till teleport animation finished
    get_tree().change_scene_to(next_scene)
```

Figure 4.11: Portal script

It served as a great baseline for a generic platformer game. It was found that this practice without a doubt lessened the problems that were encountered programming the game for the project. Although it came with one major downfall… a great deal of time. In hindsight, the Platformer tutorial alone would have given enough confidence to get started on the game. It would have been wiser to just work off an example with similar gameplay because the project was not to learn how to be a master of the Godot game engine, but to focus on the development of your own game.

4.4 Man v. Martian

How to shoot

```
71 func shoot() -> void:
72     if Input.is_action_just_pressed("shoot"):
73         var bullet = bullet_scene.instance()
74         get_parent().add_child(bullet) # unbinds bullet speed from player speed
75         # by adding bullet as a child of the root node (instead of Player)
76         bullet.position = $BulletSpawnPosition.global_position
77         if $AnimatedSprite.flip_h:
78             bullet.set_direction(bullet.get_direction() * -1)
79             bullet.position.x = bullet.position.x - 138
```

Figure 4.12: Player Shoot function

Adding music

5 Design

When developing the general idea of the game, character sprites (and a tile set) were taken from online for the purpose of getting the major features implemented. These served as placeholders or test sprites to save time on asset creation at the beginning of development. The main priority was to program the core physics and game play. Now the project was in dire need of some game design which is basically what the game looks like and how it is played.

First, some research was done on 2D game art styles. Two of the most popular styles are Pixel Art and Vector Art. It was decided that pixel art would be used for this project as it was more consistent with the retro theme (usually associated with earlier video games). This is where many pixels or tiny coloured squares are combined to make an object in the game. The next thing to consider was what software should be used to create the character sprites in pixel art. There were many options available for this task. Some of the more popular tools were Photoshop, Krita, Gimp and Aseprite. The choice wasn't going to make much difference here. It was a matter of personal preference for either something you may be used to (Photoshop/Gimp) or unfamiliar software designed for creating pixel art. There were benefits and drawbacks to both.

In the end Aseprite was chosen so some time was spent getting used to the interface and tools, but honestly it was straightforward enough. A software designed for one purpose meant it was a lightweight but had all the desired functionality. However, some practice drawing was most certainly needed before any attempt could be made at creating character sprites. Luckily there was a pleasant man on YouTube that showed how one could make the ultimate doughnut (and turn it into a monster)

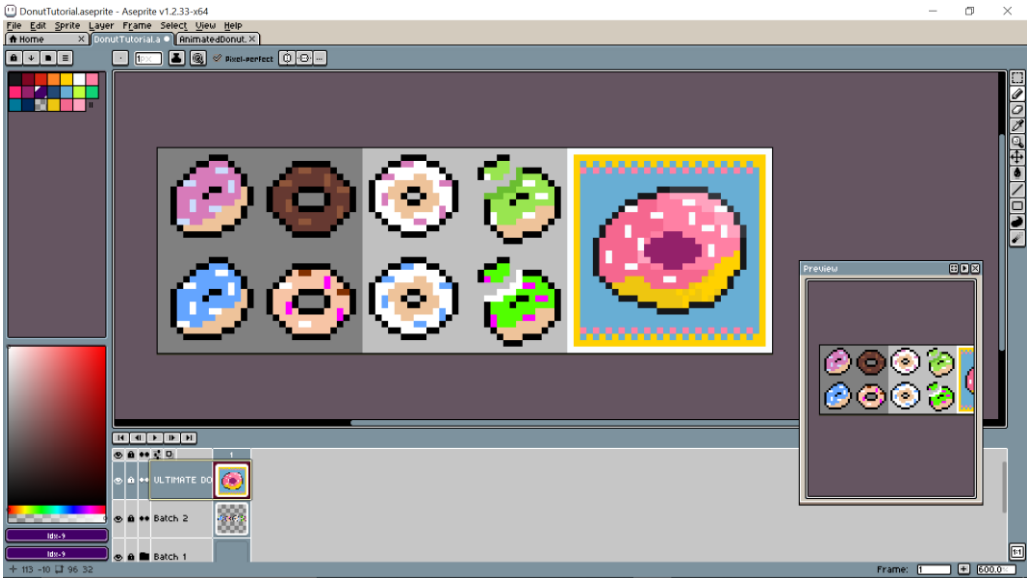


Figure 5.1: Pixel art batch of doughnuts

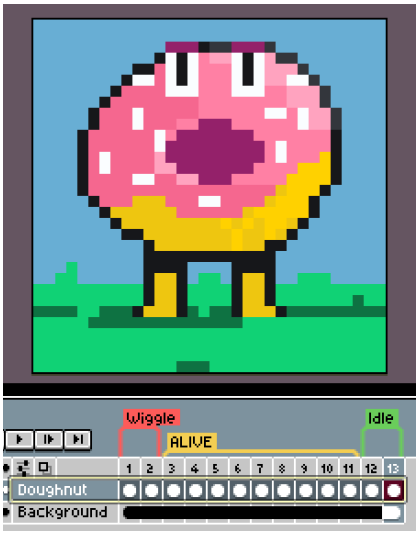


Figure 5.2: Doughnut Monster

Finally, it was learned how to animate the art by using a sequence of frames and applying the twelve principles of animation. It was discovered that a standard walk/run animation included eight different frames. The software streamlines this process by enabling the user to copy one frame to the next and make slight changes. After all this effort, it was exciting to get started on the main character/player sprite.

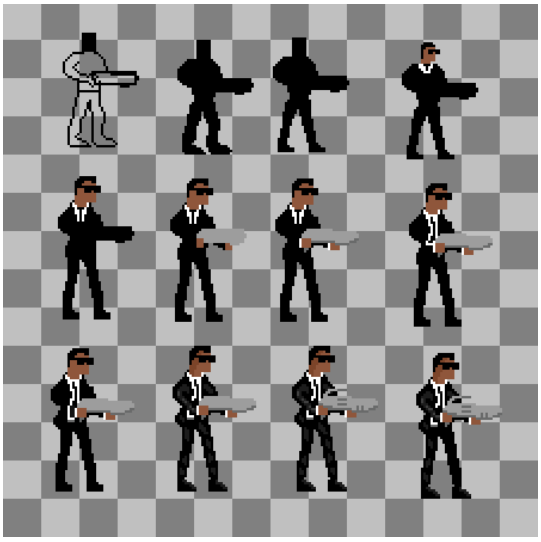


Figure 5.3: Player Roughwork

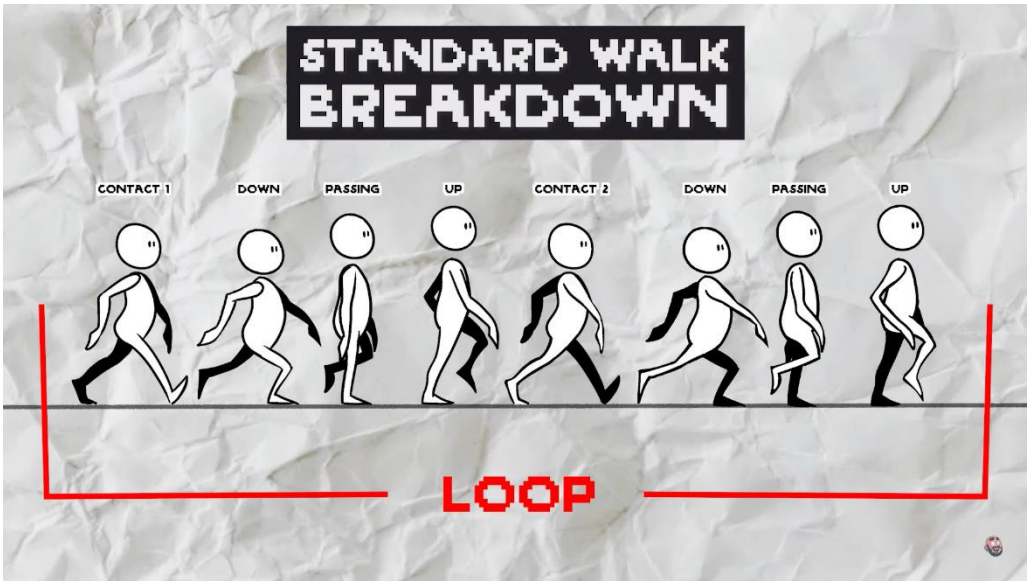


Figure 5.4: Walk cycle animation

Figure 5.5: Run Animation process

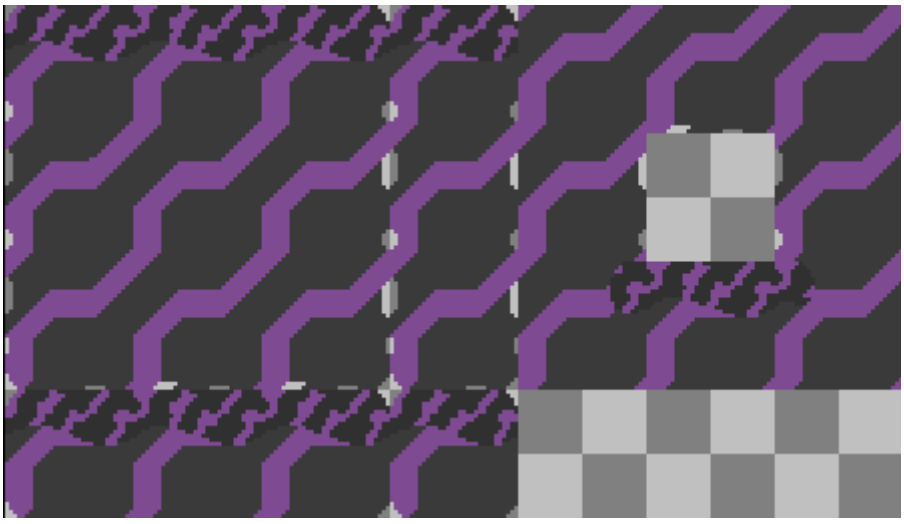


Figure 5.6: Tileset

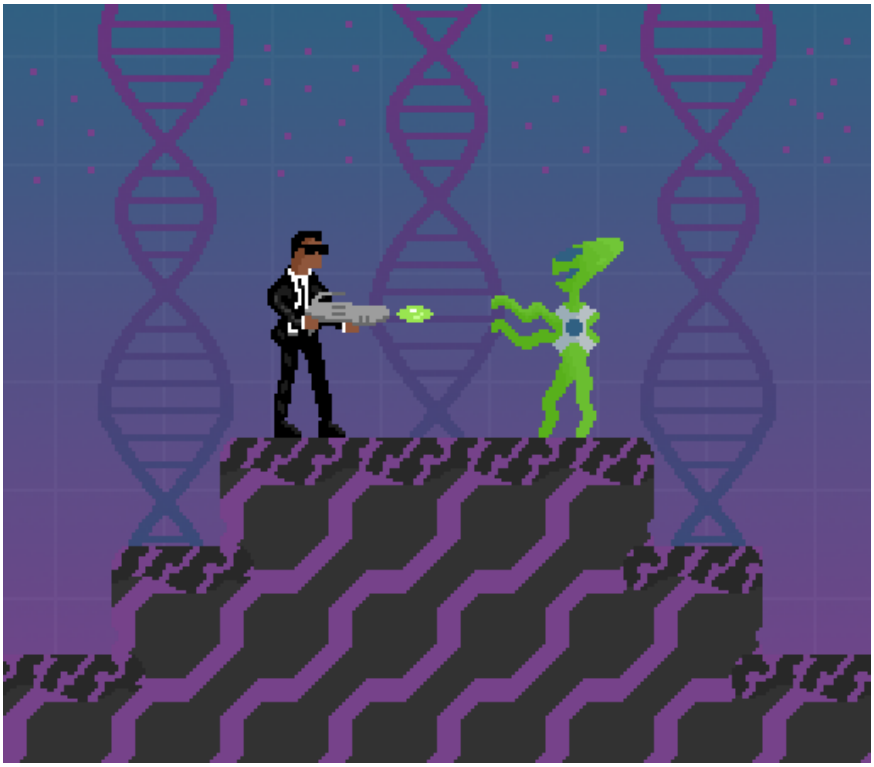


Figure 5.7: Result artwork

6 Conclusion

You may be able to spot a recurring theme throughout this project. Although it is almost at the point of a beta release, most of the time was spent on learning and practice. It was very easy to get lost in the fun of the project and get caught up in the education of each process because it was genuinely enjoyable and fascinating. It is so simple to put too much focus into one small element of the project. For example in the game design if you make a player sprite of a certain standard you must also make the enemy sprite to a similar standard or the difference may be off putting in the final gameplay.

At the beginning, there were too many ideas for all the possible features that could be incorporated into the game, while not knowing how to implement any of them. There was an impression that there would be number of character choices with different attire that the player(s) will be able to select e.g., men in black agent or commando trooper. Another assumption was that would be eight different stages with increasing difficulty. One idea would be to introduce jet packs on the second or third stage that can be used to shoot flying alien enemies. I could also have an alien boss halfway through and/or at the final stage. Another feature could be to introduce mini flying robots that you can shoot down to collect powerups or health.

The main sentiment is disappointment for inability to finish the game on time and write a better report. Sometimes you just run out of time…

Bibliography

[1] Wikipedia, Super Mario Bros. https://en.wikipedia.org/wiki/Super_Mario_Bros.

[2] Sega, Sonic the Hedgehog: https://www.sega.com/games/sonic-hedgehog

[3] Wikipedia, History of Video Games: https://en.wikipedia.org/wiki/History_of_video_games

[4] Wikipedia, Spacewar!: https://en.wikipedia.org/wiki/Spacewar!

[5] Wikipedia, Pong: https://en.wikipedia.org/wiki/Pong

[6] Classic Reload, Adventure: https://classicreload.com/atari-2600-adventure.html

[7] David Hodgson (Activision), Pitfall!: https://blog.activision.com/activision/2019-04/RetroVision-Feature-Part-1-Happy-37th-Birthday-Pitfall-Harry

[8] Retrobates, Kirby's Adventure: https://www.retrogamer.net/retro_games90/kirbys-adventure/

[9] Retro Gamer Team, Psycho Fox: https://www.retrogamer.net/retro_games80/psycho-fox/

[10] Map 256 Glitch: https://pacman.fandom.com/wiki/Map_256_Glitch

[11] Nintendo, The Legend of Zelda: https://www.retrogamer.net/retro_games80/psycho-fox/

[12] Zelda Dungeon, The Legend of Zelda: https://www.zeldadungeon.net/wiki/The_Legend_of_Zelda

[13] Pocket Gamer, What is 8-BIT? |What are 8-bit graphics, anyway? https://www.youtube.com/watch?v=QaIoW1aL9GE

[14] Jason, Schreier, The Seven-Year Saga Of Shovel Knight Is Finally Over: https://www.kotaku.com.au/2019/12/the-seven-year-saga-of-shovel-knight-is-finally-over/

[15] Danny Paez, How a beloved video game style was spawned out of necessity: https://www.inverse.com/gaming/8-bit-games-meaning-definition-origins-list

[16] Kate Fanthorpe, Owlboy Nintendo Switch review: https://www.indiegamewebsite.com/2018/02/13/owlboy-nintendo-switch-review/

[17] MistyMines, Contra: Level 1 Music (NES, 1987): https://www.youtube.com/watch?v=jWBIZ4Jj3Kg

[18] Power Rangers 2: https://www.romhacking.net/?page=translations&action=images&id=1395&imageid=4

[19] Jim Anderson, Python vs C++: Selecting the Right Tool for the Job: https://realpython.com/python-vs-cpp/

[20] Unity: https://unity.com/

[21] Unreal: https://www.unrealengine.com/en-US/

[22] Pygame: https://www.pygame.org/

[23] DaFluffyPotato, Super Potato Bruh: https://dafluffypotato.itch.io/super-potato-bruh

[24] Tiago Prado Oliveira, Rambit: https://www.pygame.org/project-Rambit-2141-.html

[25] Pyglet: http://pyglet.org/

[26] Nobody's Hero, Python: Shooting Game Programming by using Pyglet Library: https://www.youtube.com/watch?v=RWZfQLvtqrU

[27] Pyxel: https://github.com/kitao/pyxel

[28] Godot: https://godotengine.org/

[29] Gayan, Best gaming programming software 2020: https://techcastlk.com/best-gaming-programming-software-2020/

[30] Roach Games, Kingdoms of the Dump: https://godotengine.org/showcase/kingdoms-of-the-dump

[31] John Watson, Gravity Ace: https://godotengine.org/showcase/gravity-ace

[32] Wikipedia, Godot (game engine): https://en.wikipedia.org/wiki/Godot_(game_engine)

[33] Leocesar3D, Human Diaspora: https://godotengine.org/showcase/human-diaspora

[34] GDQuest: https://www.youtube.com/channel/UCxboW7x0jZqFdvMdCFKTMsQ

[35] GDQuest, Code Your First Complete Game with Godot: https://www.youtube.com/watch?v=WEt2JHEe-do

[36] GDQuest, Make Your First 2D Game with Godot: https://www.youtube.com/playlist?list=PLhqJJNjsQ7KH_z21S_XeXD3Ht3WnSqW97

[37] Aseprite: https://www.aseprite.org/

[38] Saultoons, Pixel Art Tutorial: https://www.youtube.com/playlist?list=PLp99DFIAFv-xM8l6Uzmg83yvGRu6jMfid