Video Game Development

A project by Felix Shepherd.

Rudolf Steiner–Skolen Aarhus, 2026.

Table of Contents

Intro

For the past year, I have worked with the topic “video game development,” where the practical part of my project has been to make a video game myself using the Unity game engine. In this essay, I hope to explain the journey I have gone through, the things I have learned, and other related topics that I have explored during my time working with game development, while emphasising programming in general.

Some of the original goals I had for game development were to understand what goes into creating the games I have spent a large part of my life playing. And to learn some of the skills that are needed.

But I also had some interests and questions that I thought could be answered well through the practical part of creating a game myself, such as:

- How has the game industry evolved from where it started?
- What is a game engine and how does it assist game creation?
- What are the different types of games and game styles?
- What styles of games are popular, and what are the trends in the game development industry?
- What is the difference between game studios and small team games, and what do the different groups focus on?
- What are the other aspects of game creation beyond coding, and how do they tie into the games, such as sound design and visual design?
- How can games tell a story or show a narrative through their design?
- How has artificial intelligence influenced game creation, and what does it mean for the future of the industry?
- How does coding a game differ from the general programming that I am used to?

These are all questions that I also plan to centre the written part of my project on, as well as of course, telling my experience with developing a game and passing on some of the useful things I have learned that I wish I could have known before I started making my own game.

Game development and why I chose it / Background

In late 2023 and early 2024, while on a trip in South Africa, I decided to learn the basics of coding. I had been interested in coding for the better part of a year at that point, and with the limited internet access and abundance of time, it felt like a natural choice to actually sit down and learn it.

I started with learning the coding language Python. It seemed like a logical place to start, given how it was described as a versatile and simple coding language that could be used for all sorts of projects. For the next month or two, when I would occasionally get internet connection, I would download documentation PDFs and tutorial videos with which I taught myself the basic fundamentals of programming.

After returning to Denmark, I didn’t spend much time on programming until the end of 2024 and beginning of 2025. Here, I began making my first game. I used ‘pygame’, which is a visual library for Python, meaning that you can use built-in functions in it to display things on the screen.

I spent about a month on this project, and it helped brush up my skills in coding, but also taught me a lot about the basic principles of game development. It also taught me about the other aspects needed to make a game.

After completing this, I continued to spend time working on various coding projects to keep myself busy.

The reason I originally decided to start making a game in Python was to prepare for the coming assignment. I originally decided that I was going to make a video game around 9 months before we were given the project. I chose game development since I found it was an exciting way to visualise the coding process and motivate myself to learn more about coding, as well as being a very creative process that had a lot of room for all sorts of storytelling or other forms of expression, and also a way to use the knowledge I had amassed from years of playing video games myself.

I also thought that this project, and what it would teach me, could be beneficial for my future career, as I want to get into the robotics engineering industry, where I assume being able to code could translate well into future possibilities.

History of Video Games

Start and early history

The earliest video games were not published commercially but rather were technological demonstrations on cathode ray tube (CRT) screens or oscilloscopes.

From this, several ‘video games’ arose, including the 1952 ‘OXO’, which was a tic-tac-toe game that could be played on a computer screen. Then followed ‘Tennis for Two’, which could show a simple tennis match on an oscilloscope and various other visual simulations.

The generally believed first video game by modern standards was ‘Spacewar!’, a game developed in 1961 at MIT, where two players would control different spaceships and battle each other while flying around the gravitational field. While originally controlled by the buttons on the computer, it was later replaced with a gamepad for easier use. This was also the first game to be spread across multiple computers.

These games and simulations, while being basic, provided the backbone for future games to be created.[1]

Arcade games and home consoles

During the 1970s, developers started making hardware specifically designed for games, and it was here that we saw the first coin-operated arcade machine, which ran ‘Pong’, a simple two-player game that gained massive popularity and started the growing interest in the industry.

This would be followed by different home consoles that brought games to the consumer’s house. From here came the next generation of consoles that could run multiple different games that came in the form of game cartridges.

Through all these new products, the video game market boomed, but then soon after, due to poor regulation and a massive number of low-quality games being pushed into the market and following

[1] Mark J. P. Wolf, "History of Video Games," EBSCO Research Starters, 2024, https://www.ebsco.com/research-starters/history/history-video-games , accessed November 13, 2025.

this, there was a subsequent crash of interest on videogames in North America.

This, however, was counteracted later in 1985 by the games companies Nintendo and Sega, who each released consoles that had strict regulations on the games they provided to avoid a similar crash.[2]

3D games

During the 1990s, with the release of more powerful consoles and computers, the first 3D games started to be released. These games used polygonal graphics to show simple shapes in a 3D space, which would then be coloured over with a texture to give the illusion of detail. With the new added dimension, game creators could explore whole new genres that were not possible before, like 3D shooter games or exploration–focused games.

Some of the pioneers of this new technology were ‘Doom’(1993), ‘Wolfenstein 3D’(1992), and ‘Virtua Fighter’(1993), each showcasing new possibilities or upgrades on previous genres.

At this point, these games still used blocky, basic shapes. This, however, changed during the late 1990s and early 2000s when more powerful computers were released (namely, the introduction of the graphics processing unit [GPU]), which enabled new and more complex modelling algorithms to display more detailed and stylised models. Following this, game creators had increasing freedom in how they could stylise the visuals for the games they made. They could choose to make them as realistic as possible or to choose a cartoon–like appearance, depending on what suited the narrative of their game.[3]

Online games

Until this point, all games had been made for either one player or two players using the same physical computer, however key innovations led to the possibility of players being able to play in the same game even if they were on separate machines and far from each other.

The first types of games to achieve this were games played on terminals that connected to a mainframe, where they could have multiple terminals connected to the same mainframe. The terminals themselves would not run the game but rather just be a

[2] Wolf, "History of Video Games."

[3] RocketBrush Studio, "The Evolution of 3D Graphics and Its Impact on Game Art Styles," RocketBrush Blog, March 4, 2024, https://rocketbrush.com/blog/the-evolution-of-3d-graphics-and-its-impact-on-game-art-styles , accessed November 15, 2025.

display for the mainframe. These systems were only used around research centres and were never widely adopted.

The next evolution was MUDs, which stood for ‘multi-user dungeons’. These were the first true multiplayer games. They worked the same way the previous ones did, but these allowed for multiple players to have a completely different experience and still interact with each other. These games, though, still had to rely on one central computer, so they were limited to being simple text-based games.

While simple, these games set up many of the systems that we still see in modern games.

The next evolution came with the spread of dial-up internet. It became possible to connect players over large distances. With this, you could either directly connect with another person’s computer or join public game servers with multiple people connected. But due to the limited efficiency of dial-up internet, these games would also be text-based or very low-bandwidth games.

The beginning of true internet games, as we know them now, came with the widespread adoption of home broadband internet. Now players could freely play fully visualised 2D and later 3D games in lobbies.[4]

Mobile games

The start of handheld games was in the form of purpose-built mobile game consoles starting in the 1980s, or preinstalled games on Personal Digital Assistant (PDA) systems. But the latter of the two was limited in what they could play, as their games were meant more as tech demos than anything else.

During the 1990s, cellphones started to become commonplace, many of which came installed with simple games like ‘Snake’. But this changed with the release of the i-mode mobile internet service; now users could download and play games that were not preinstalled. And while system storage and power were incredibly limited, it paved the way for app stores as we know them today.

[4] Datapath.io, "The History of Online Gaming," Medium, 2017, https://medium.com/@datapath_io/the-history-of-online-gaming-2e70d51ab437 , accessed November 16, 2025.

In 2007 and 2008, the first iPhone and App Store were released, which led directly to a massive boom in mobile gaming. This was thanks to the iPhone not having a built-in keyboard, therefore freeing screen real estate and allowing games to have their own controls customised to fit the game. This, combined with their upgraded processors, allowed for far more complex mobile games than ever before.[5]

Current era and future

Since then, computers and consoles have gotten magnitudes more powerful, which again allowed for more innovation in the industry and the creation of massive, multiple-thousand-employee game studios that can pump out new massive games each year.

The games industry also saw one of its biggest successes with the lockdown due to COVID-19, as while so many people were stuck inside, many naturally started to get invested in playing video games.[6]

But it is in my opinion that with the new audience and emergence of free-to-play online games, the presence of microtransactions has skyrocketed. Now, many games are simply made to capitalise on this industry rather than being a product of creativity.

Conclusion

While video games started as simple technology demonstrators, they have evolved into one of the biggest consumer industries in modern-day society.

[5] Andrea Knezovic, "The Complete History of Video Games 1952 – 2026," Udonis Blog, January 12, 2026, https://www.blog.udonis.co/mobile-marketing/mobile-games/history-video-games , accessed November 13, 2025.
[6] Knezovic, "The Complete History of Video Games 1952 – 2026."

What is a game engine?

When deciding to make a game, the first thing to consider is which game engine you want to use. A game engine is a software sort of like Photoshop or After Effects in the sense that it can assist you in setting up and creating a game, while it will handle most of the necessities of a game, such as rendering what you can see or hear. This helps since it lets the developer focus on making the game itself, rather than all the low-level backend of interacting with the different operating systems and the hardware to visually display the game. On top of this, game engines will give you a lot of tools that assist in the game making process.

What to consider

Some of the most popular game engines are: Unity, Unreal Engine, Godot, GameMaker, Construct, and Ren'Py.[7]

Each of these engines has its advantages and disadvantages, and these are important to know when deciding what engine will suit the type of game you want to make.

One thing to consider is the complexity of working with the engine. This is also based on what type of game you aspire to make. Based on my own experience, a more complex engine is harder to learn and use, but often allows for more complex games as a result. On the other hand, using a less complex and more abstracted engine can save a lot of time or might not require the developer to be as familiar with coding as some other game engines would, by using more visual-based scripting. But it would come with the drawback of less customisation and freedom for the developer.

Of the game engines mentioned, Unity and Unreal Engine are the most complex, while Construct and Ren'Py are easier to pick up and Godot and GameMaker are somewhere in the middle of that spectrum.

It's also important to consider that more abstracted engines run slower and therefore can limit what games can be made on them. For example, GameMaker, Ren'Py, and Construct are limited to creating 2D games only, with Ren'Py focusing on visual novel games, and Construct specialising in visual coding. advertised

[7] GamesFromScratch.com, "Game Engine Popularity in 2024," GameFromScratch.com, January 29, 2024, https://gamefromscratch.com/game-engine-popularity-in-2024/ , (accessed December 2, 2025).

to non-coders, and GameMaker offers extensive premade assets that speed up development time.[8] [9]

On the other hand, unity and unreal are made for 3D games and can run large games fine, Godot also supports 3D but not to the same scale as the previous two. [10]

A game engine's speed depends on a lot of things, a major one being what scripting language it uses. Engines like Unity and Unreal Engine use C# and C++ respectively, which are considered more low-level languages compared to Python, HTML, or JavaScript that the earlier-mentioned engines use.

The difference between a low-level and high-level language is based on how much or little the code you write is interpreted before directly interacting with the hardware (abstraction). A low-level language uses less or no abstraction, meaning it interacts more directly with the hardware and therefore has more restrictions and requirements for the code you write to follow. A high-level language will use higher levels of abstraction to translate the code you write into lower-level instructions and then feed it into the hardware. This makes the language slower but also easier to write as there are fewer syntax restrictions you must follow.

Using a game engine[11]

Engines are designed so you import the visual elements that you make in different software, this can either be 3D models or 2D sprites and animations, and then you can create scripts inside of the editor, which you can you have to edit externally (some game engines like Godot have a built-in text editor, but Unity doesn't). You can then combined these assets and add some components provided in the game engine to create functional game objects, for example: if you add an image or animation of a character and you want them to move, you would have to a script, which would contain the logic on how the object should behave i.e.; if a certain key is pressed it should then be moved in a

[8] Wikipedia contributors, "GameMaker," Wikipedia, The Free Encyclopedia,https://en.wikipedia.org/w/index.php?title=GameMaker&oldid=1336235506 (accessed February 13, 2026).

[9] Tom "PyTom" Rothamel, Ren'Py Visual Novel Engine, version 8.5.2 (https://www.renpy.org/ , accessed February 13, 2026).

[10] Wikipedia contributors, "Unreal Engine," Wikipedia, The Free Encyclopedia, https://en.wikipedia.org/w/index.php?title=Unreal_Engine&oldid=1338082598 (accessed February 13, 2026).

[11] I have come to realise that some of the following terminology might be exclusive to Unity Engine since that's the one I learned.

corresponding direction. Then you would add some components like a hitbox/collider and possibly a physics component/ body to make it properly behave as intended.

In this way, a game engine assists in the creation of games in the same way that a Lego set provides the individual bricks that you can build something out of and customise to your desired effect.

Of course, there is much more to talk about in how to use a game engine, but I will talk more about that when I talk about my process.

When going through this process, I decided on Unity as it is the most popular engine by far, but it also has good support for the type of game that I wanted to create.

Unity

Unity Engine was first released by David Helgason, Joachim Ante, and Nicholas Francis under the company name of ‘Over the Edge Entertainment’ and was unveiled in 2005, with the stated goal of ‘democratising game development’. Originally, it was designed for Mac OS but was later ported to other operating systems. Following the engine’s success, the company was rebranded to ‘Unity Technologies’ and was relocated from Copenhagen to San Francisco. Since then, it has become the most popular game engine, which is used by both hobbyists and large game companies.[12] [13]

[12] Wikipedia contributors, "Unity (game engine)," Wikipedia, The Free Encyclopedia, https://en.wikipedia.org/w/index.php?title=Unity_(game_engine)&oldid=1333900879 (accessed November 23, 2025).
[13] Wikipedia contributors, "Unity Technologies," Wikipedia, The Free Encyclopedia, https://en.wikipedia.org/w/index.php?title=Unity_Technologies&oldid=1337692308 (accessed February 13, 2026).

Game Styles (Genres & Artistic Approaches)

Games have evolved a lot from where they started, but this does not mean that the techniques and styles that were used in the past are no longer used today. Rather, all styles are equal, and it’s more of a personal preference as to which is most enjoyable to develop and play.

The main and most noticeable choice a game developer makes is whether to make the game 2D or 3D.

With 2D game development, visual elements are still drawn images, often animated by drawing each frame individually. While 3D games use models that are made and rigged so their look can be dynamically updated.

Both 2D and 3D have advantages and disadvantages in what games they are most suited for, but it doesn’t prevent you from any specific genre.

Beyond this, the genres of games you can choose from are vast. Some of the most popular ones include: First-person shooters, platformers, fighter games, adventure games, role-playing games, simulation games, strategy games, puzzle games, sports games, and so on.

Current Video Game Styles & Trends

There is a growing split between the two groups of game makers: solo or small team game studios and the large giants of the industry. The big studios have the overwhelming advantage in personnel, money, and experience and therefore mainly focus on large 3D projects, often utilising well-known IPs (like popular shows or characters) to ride on their past success and attract players.

On the other hand, small studios often choose to work with 2D games or smaller-scale 3D games as those are more manageable to make with less budget and people, but these games often end up being more impactful in their storytelling or by being more willing to try new types of gameplay since they have less on the line and therefore more freedom in expression.

In recent years, there has been an ever more desperate monetisation of the large studios’ games, while they are unable to innovate and seem to just remake the same type of games each year, betting on the fact that the players will continue to buy into it out of habit or nostalgia, which has pushed many people towards indie games again.

Indie Games Reemerging

Following the stagnation or degradation of titles being released by big game studios, and the ever-increasing number of tools and information that make it easier to make games, the presence of indie games in the mainstream has increased as well.

I have personally noticed that compared to just a few years ago, the amount of small dev team or indie games that become so popular that they outpace many of the well-known studios has also dramatically increased.

Coding

Note to the Reader: the following segment is more technical as it's hard to discuss these topics without using terminology that might not make sense if you do not have experience with coding

General rules for coding

The whole idea behind coding is to make the computer do things for you, so for me, the point is to minimise the amount of effort you must put in to get the computer to do what you want it to do, which leads us to the point of making reusable code.

When you make a script to do a specific task, i.e. for your player to pick up an item or weapon on the ground, you wouldn't want to remake the same script again and again for every different type of item or weapon that your player can come across, instead, you would want to make one script that can handle all the cases and scenarios it could face. For example, bad code could be something like:

```
      void OnTriggerEnter2D(Collider2D other)
  {
      if (other.gameObject.name == "shotGun")
      {
          equip(shotGun);
      }
```

```
    }
```

- This is an example of bad code, here when the player comes into contact with another item, it will compare the name and if the item is called shotGun It will equip the corresponding game object for the shotgun.
  This is bad because you would have to make cases for every item you come across.

```
void OnTriggerEnter2D(Collider2D other)
    {
        weaponItem item = other.GetComponent<weaponItem>();
        if (item != null)
        {
            equip(item);
        }
    }
```

- On the other hand, this is the system that I used, where instead I would create a reusable script called weaponItem and attach it to each of the weapons I wanted the player to be able to pick up, then when the player comes into contact with an item the script will check if that item has the 'weaponItem' script and if it does, will equip the weapon that you touched.

These are, of course, massive oversimplifications and in these cases, you would still need to program the 'equip' function, which honestly, was far harder to make.

Datatypes and terms for understanding this doc

Variables are like containers for data. They will have a name that you can call on, but you also must define what type of things it can hold, as it can only hold one type at a time. There are many datatypes, but to give an example, you could have an 'int' variable, which can contain numbers (integers), or you could have a string variable which contains strings of characters (words or sentences). Variables are therefore incredibly useful for saving data or rewriting data in the future.

In programming, there are two types of approaches to how a programming language handles variables: statically typed languages and dynamically typed languages. The difference being that for statically typed languages, you must declare the variable type on initialisation and then cannot be changed later. Whereas with dynamically typed languages, the same variable that you could have used to, for example, store a string, can later be reused to store an integer during runtime.

Datatypes. Define how the following data should be handled and stored, as well as how different operators should interact with the type.

Operators. Perform different actions based on what datatypes they are interacting with. The operator’s signs vary from one programming language to another, but some of the most common are: +, −, ∗, /, and =. An example could be that if the + operator was interacting with two integers, it would simply return the sum of the two integers, while if it was interacting with two strings, it would return the two strings concatenated into one.

Functions. Are like a list of instructions for the computer to execute. These could use variables and perform different actions with them, and they can take multiple different types of input and use them all based on how you set them up. They are therefore one of the most important parts of making reusable code.

General code structure of C# in Unity

C# is a coding language developed by Microsoft that has a similar syntax to other C-style languages but differs in that it is object-oriented and has more abstraction. Object-oriented means that all scripts you write will be wrapped inside of classes, which are a sort of container for functions and variables, which in Unity’s case are attached to objects.

The scripts then, by their nature, can be adapted to fit any purpose with few limitations.

Operation order

The scripts interact with the game using a few key functions. Where in these you can place statements, create logic checks, or call your own functions, and then these will all be called at a specific point during your game’s runtime based on which function they are placed in.

The two most common are:

```
void Start()
   {}
```

And

```
void Update()
   {}
```

In Start() the items it contains are run before the first update. And in Update() the items are repeatedly run every single time the frame updates.

There are also others, for example, Awake() which is run as soon as the object is initialised, and FixedUpdate() which is run at the set pace that physics calculations are run in the engine (normally 50 times per second). These are just the most common handful of the large list, but they demonstrate how the operation order of scripts' logic is decided.

Movement

In my opinion, movement in any game is by far one of, if not the most important mechanics to do well. In my experience there are multiple different ways to set up movement, and the choice ultimately should be based on what type of game you are making and how you plan to design the maps.

As movement is designed to traverse the maps the player is placed in, it should match that, i.e. if the player has to traverse large open worlds, it might be better to give them higher movement speed, or if the maps' objects are vertically tall, it could be worth increasing the jump height or giving them some alternative method to scale these obstacles.

How you execute this depends on whether you are making a 2D top-down, 2D side-on, or 3D game.

For 3D games, they differentiate themselves by the fact that you have to account for navigating a z axis with both the camera and character, as well as all the problems from side-on games.

For 2D games, the viewing method fundamentally affects which methods you can use. For a top-down game, it is far simpler, as you just need to directly manipulate the player's position based on which direction is pressed. While for a side-on game you have to account for physics, now you have to have a gravity force pushing the player down, and vertical movement can often feel lacking if you just directly manipulate the player’s position. If this method is used it greatly reduces the options on how you implement jumping, as if you are constantly rewriting the player’s position, it will overwrite whatever preceded it. Thus, one solution is to use vectors and a physics engine to ‘push’ the character when moving to simulate acceleration and deceleration and to account for simultaneous movement on both axes.

Mob AI

For me, programming the enemy mob’s logic scripts was the most fun part of programming.

When creating an enemy AI, you have to design it to have multiple states, the more complex the enemy behaviour should be, the more states it will have to have.

The simplest version is to have two states, one for when the player is out of the mob’s engagement range and one where they are in it. From there, you can make countless different states to match different situations based on whatever factors you think the mob will face.

Combat

The way you design combat should also be linked to the map’s design, the movement system, and the enemies that the player has to fight. But this is also one of the most expressive elements of a game and therefore has no limits or guidelines that you have to follow when designing it.

Other Aspects of Game Development (Beyond Coding)

During my process, I quickly came to find that game development is only about 40% coding, the rest is working on the assets you will use. However, the coding was the part of the process that I was most confident in, and I believe that was reflected in my work ethic when making my assets. Instead of making the best possible assets that I believed I was capable of, I just aimed to make something usable, thus allocating more time to the part of the process I was confident in. Due to this, my experience in the following areas is very limited.

Sound Design and music

Since I did not have access to equipment to record sound effects in real life, I had to make them electronically.

My main goal when making sound effects was to firstly make a consistent sound profile that all sound effects fall into, so that no sound felt out of place or disrupted the gameplay. And secondly That each sound should fit its intended purpose while being distinct enough that it could be easily distinguished from the others.

For music, while I didn’t have much time, I tried to make simple and repeatable music so that when you hear the same song repeated multiple times, you wouldn’t get sick of hearing it, and to aid this goal, I made multiple tracks that could chain together at random to hopefully give a unique sound experience each time.

Drawing and Animation

Art is debatably more important than code in a game, so it became a problem when I realised I was not all that good at it. Over time, I found I could make up for my lack of talent by using digital pixel art and aiming for a simplistic style, something that I explain in more detail during my process segment.

Animation was another challenge I faced. The main technique I used was to just draw a still image of the start and end frames of the animation I wanted to create, and from there add the frames in between to make it fluid.

Writing & Narrative Design

Even with good gameplay and art, if a game does not have a meaningful story, it will just end up confusing and lacking purpose.

Due to my time constraints, I realised I would not have time to make a fleshed-out story, so instead I chose to just focus on making a small but playable game.

My process

Getting started

Going into the practical part of my project, I knew that it would take a huge amount of time to complete, so I decided to be as ready as possible and start as quickly as I could. So as soon as the assignment officially started, I dedicated a few hours a day to working on the project. The first thing I decided to work on was learning the scripting language for Unity, C#.

This wasn't too hard a process since much of the coding knowledge and experience I had from learning Python carried over to learning C#, so it was mainly getting used to the syntax and new rules of the language.

Python is a general-purpose high-level language, while C# is an object-oriented language that is lower level than Python, meaning it has less abstraction. In practice, this means that when coding in C#, I had to tell the computer more directly what to do than I was used to in Python.

I spent about a month learning and practicing C#, where most of the time was spent translating terms and names of features I was used to in Python into C# and relearning how to use them in the new system. At the end of this month, I felt that I had a decent understanding of the basics, and even though there were still many things I had to learn, I felt it would be better to learn them in practice by actually starting to make a game.

By this point, I had spent a lot of time thinking about how I wanted my game to end up, and while I didn't have a fully polished-out version in my head, I had the basic ideas down so that I could begin the setup.

The style I decided on was a 2D pixel art side-scroller game, and at this point, I thought it would become a story-oriented game, but I would end up changing my mind later in favour of a rogue-like game. I chose to make 2D as I personally enjoy a lot of 2D games more than 3D ones, also because they are simpler to make. For the art style, I decided to use pixel art for the sprites since I felt that by limiting the number of pixels I had to worry about, I could skip over much of the detail while keeping a coherent visual style. And then I had to decide between a top-down 2D game or a side-on 2D game. This difference is simply which angle you see the game from (either from above or from the side), but it fundamentally changed how game logic would function, so it was important. I ended up settling on a side-on game simply because it's what I enjoy playing more.

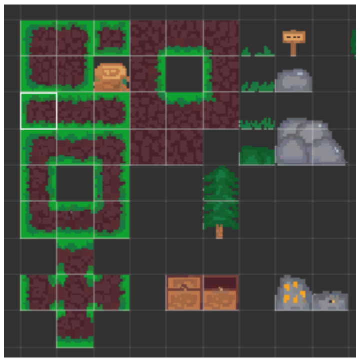
First test game

Now that all the basic decisions were made, I chose to make a test game, just a small project file that I could use to learn the basics of using the Unity engine while not worrying about making future-proof code.

Here, I tested out how to add image files and connect them with game objects and scripts. I made a simple testing script for player movement. This script, in hindsight, was of poor quality for many reasons, but at that point, it taught me a lot.

Besides making a simple player character, I also tested out the tile maps system in Unity and made a simple map with it using a free tile map image I had found online.

Tile maps are a built-in function of the Unity game engine where you can take an image that you would make, where each section based by grid size would be an individual 'block' you could place onto a grid on your active scene. The image is called the 'palette' and should contain each block in every possible rotation and combination so that they can be freely used in the editor.



Example of the tile map I later created.

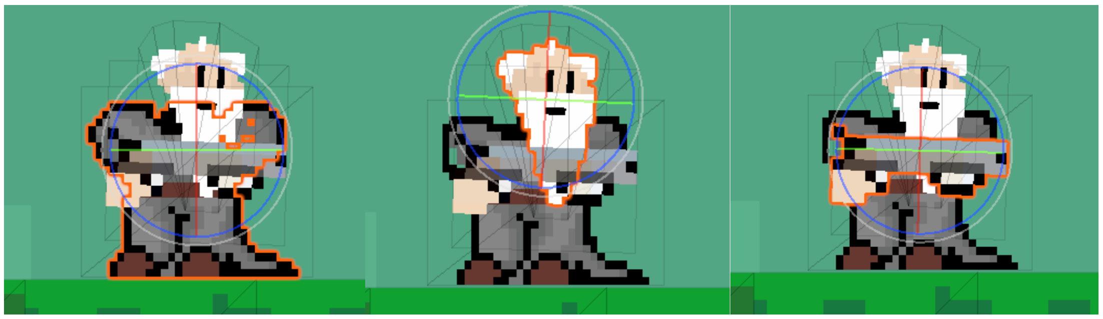
At this stage, by far the most time-consuming task was the research. It felt like for every single item that I wanted to add or program, it would take full days of research into which systems and compositions would best achieve the desired effect.

After this short test period, I decided to create a new project file and start the construction of my actual game.

Start of the main game

Since I was working on a 2D side scroller, the first thing I did was make a tile map and create a simple ground for the player to stand on, as without it, the player object would just be pulled down out of frame by gravity.

Having completed that, I began work on the player. The construction of how I would make the player is something that I had considered and thought about before, and while the first character I made was just a still image, I decided that I wanted to make the character's visuals modular, meaning that the head would be separate from the body, and the arms would be a replaceable sprite that could eventually be swapped out to add future weapons.


Example of the different components making up the player.

After finishing setting up the visuals of the player, I also set up a small movement script, one that just directly sets movement in a fixed vector direction. With these things complete, the next task was to add logic onto the camera object so that it would track the player.

The camera object controls what the player's display will see, so it is important that it follows the player correctly.

To start, I directly set the camera's position to the player, but this made it look far too fixed and rigid, so I added a small delay instead that helped the camera smoothly transition to the player's current position.

When I felt like I had a usable player movement setup, I started the arduous journey of setting up room generation.

Room generation was perhaps the most important element for my game to work correctly, but it was also incredibly hard to set up at the stage I was at with learning Unity. Since I was making a rogue-like game with random generation for the rooms, this meant that I would create several room prefabs (like an object you can make and then save to reuse later) each with a few exits where, when coming into contact with an exit object there, it would record the exit direction and then destroy the current active room object and spawn a new one with an entrance opposite the direction the player exited the last room, then it would teleport the player to that position.

The hardest part about setting up all of this was not the code but trying to figure out how to reference components and objects in the Unity editor. After learning for a bit, I started to understand how to reference components attached to the object on which the script itself was located, but getting components attached to other objects was a completely different story. This became the main issue I faced for much of my time scripting.

And to add to the already hard situation, I knew from my previous experience coding how important it would be to make the code and setup of the world correctly in the beginning, so I could continue to build up on it later without having to go back and rewrite massive parts of the scripts.

This pressure made it hard to start and locked me in a spiral of trying to figure out the most optimal way to build my scene, as now not only how I composed my scripts mattered, but also where they were placed and what they were attached to in the scene mattered too.

Looking back, I think I tried to approach this far too early in my project, but the hard part is that I don’t think there would have been an easy way to avoid this since it was such a central point to the game’s logic.

Due to all this, I ended up wasting multiple months trying to set it up correctly and only managed to finish it due to spending most of my summer holiday reworking it.

This was by far the worst part of working on this project, as in my opinion, coding is fun when you are working with concepts you are familiar with or gradually implementing new things on a small scale. So when you are forced to work with a completely new system in a programming language you are not yet familiar with, it’s easy to feel drowned in it and lose the will to work further on it.

And having spent so much time already on just setting up the basic room generation, which at that point still had a few errors I had to sort out, I started to worry about how I would get everything done in time since I also hadn’t even started on the written part of my project. It was here that I decided to set myself a deadline: by Christmas, I would have a basic playable version of the game ready, so I could work on the written part.

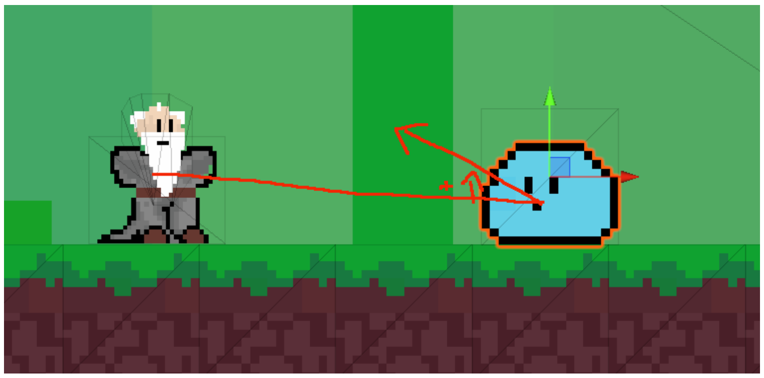
Having finally finished the room generation, I could start working on the parts I knew I would enjoy, namely combat and enemy development.

Mobs, combat and health

To do these, I created a separate clear scene and again created a simple enclosed map as a testing ground. Then, I got to work on the first enemy, which was a simple slime.

For the slime, I created a simple still image for the sprite and began on the logic, which was honestly really simple. I would make a 2D vector between the slime’s position and the player’s position. I would then check the distance and once the slime was within a set range of the player, it would begin to attack. For the attacks, I simply normalised the existing variable, then added an upward component, so the vector was pointing above the player and then force-pushed the slime object in that vector’s direction times by a set multiplier.

This would make the slime appear to jump in an arc towards the player.



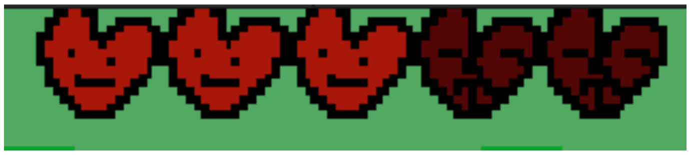
Demonstration of the vector in which the slime would be launched.

The natural next step was to implement damage to the player, and in order to do this, I had to create a system for player health and a way to display it onscreen.

The first part was easy enough. I just created a script that contained the player's maximum and current health and a public function 'takeDamage' which can be given an integer value, and it would subtract that from the player's current health.

It was the second part of this problem that took a much longer time, as I needed to learn how to make UI elements, which are the things you can directly see overlaid onto the screen. I started by creating two sprites, one that was a full heart and one that was empty. I then created a UI element with five child objects[14], each with a sprite renderer component. I then attached a manager script onto the parent object with a function that would iterate through the child objects and set a number of them to either the full heart or empty heart depending on the player's current health relative to their max health. Then I would call this function in the player's 'takeDamage' function.

So now I could combine these elements and make it so that whenever the slime came into contact with the player, it would call the 'takeDamage' function on the player and give it the preset damage integer variable, which would then update the hearts on screen to represent the player's current health.



The UI element that showed the current health.

Now that mobs could damage the player, this naturally led me to the next point of the player damaging the mobs. I first started by creating a shotgun object with a sprite renderer and a 'shotgunLogic' script. I then created a separate object that should represent the bullet. At this point, I couldn't think of a way to create multiple pellets like how a shotgun is supposed to be, so I created a single slug round to start off. I gave it a sprite renderer so it would be visible, a hitbox, and its own script that when it comes into contact with an opponent, damages them. To make this universal, I made a reusable script 'mobInfo' which I attached to all enemies I created. It contained the variables for health, current health, and other details that needed to be easily accessible. This included a public

[14] An object that is attached to another object, while being lower in the hierarchy.

‘takeDamage’ function which operated the same as the player one. With this, when the slug came into contact with another object, it would compare their tag, and if I had set that object to ‘enemy’, it would look for the ‘MobInfo’ script and make the mob take a fixed amount of damage.

To make the shotgun actually shoot the slug, I programmed the shotgun to detect when the left mouse button is pressed and then spawn an instance of the slug prefab and give it a force push in the vector from the shotgun’s position to the mouse position.

At this point, I ran into an issue regarding the player’s movement, namely that until this point, I had been checking for the player’s input and if it corresponded to the keys for left or right, I would then manually set the velocity vector of the player to that direction. This, however, clashed with my wish to add the shotgun’s recoil, since any force applied while the directional keys were held down would instantly be overwritten by said manually set velocity vector. To fix this, I did a complete overhaul of the player movement so that when a directional key was pressed, it would check the player’s current velocity and if it was under the set max speed, I would add a small force push in the desired direction. This would quickly stack up so that the player would maintain the max speed velocity. This system also allowed force pushes in different directions that still could be counteracted to an extent, but not fully overwritten as before. This system also had the added bonus of making the player’s movement more interesting, as you would now have to hold down a direction to slowly get your player up to the max movement speed.

It was also here that I started implementing some animations for the player.

With these new systems, I returned to the original scene, which was set up with random room generation, and began work on a more polished version of it. The first thing I worked on was remaking some of the scripts related to the containment of room prefabs, namely the script that was in charge of sorting the rooms into different lists based on which entrances or exits the room had.

After completing this, I decided to overhaul the whole weapon pick-up and dropping scripts. At the current time, they were buggy and a pain to fix, but more importantly, they didn’t match the type of game I was going for. In most rogue-likes, you often choose your loadout before you start the adventure, as to tailor the experience to your preferences. But the current designed system would mean that the player could pick up and drop weapons while mid-adventure. In order to steer the game back onto course with what I had envisioned, I reworked the scripts and began creating a new system.

My goal was to make it so you would select a weapon type at the start of a run and then pick up buffs/power-ups along the way that would enhance the weapon's play style in different ways, rather than letting them completely change weapon mid-way through a run.

To set about this, I made it so when the player first spawns in, they will have to select a weapon before they can continue. This would function by there being an object in the scene that the player could interact with, and when they did, it would pop up a UI element which contained multiple buttons that, when pressed, would call a function that would use a modified version of the 'equip' function on the weapon object specified by the buttons' text.

In order to further build on the start of the game, and to make use of my learned knowledge on how to use buttons, I created a start scene, which contained buttons you would expect to see on a game's start screen, for example: 'play', 'settings', and 'quit'. I then linked the 'quit' button to a function that closed the application when called, and set the 'play' button to change the active scene to the main one with the room generation set up.

From here, it was time to work on gameplay. The first thing that had to be changed was room objects themselves. At the current time, they had no way to spawn mobs. The method in which I added this functionality was to modify an existing 'mobSpawner' object that I had created for testing purposes. I had programmed the 'mobSpawner' object to create a specified mob, and when said mob was no longer active, spawn another copy. The modifications I did were to set the mob spawned to a random mob object inside of a list, and to track the number of mobs it could spawn and limit itself to a specified number. Here, I had a reusable, customisable object that I then set in the room prefabs. The cool thing about this object is that by setting multiple in the same room, you would have multiple spawns of random mobs to fight.

The problem then became that I only had one mob.

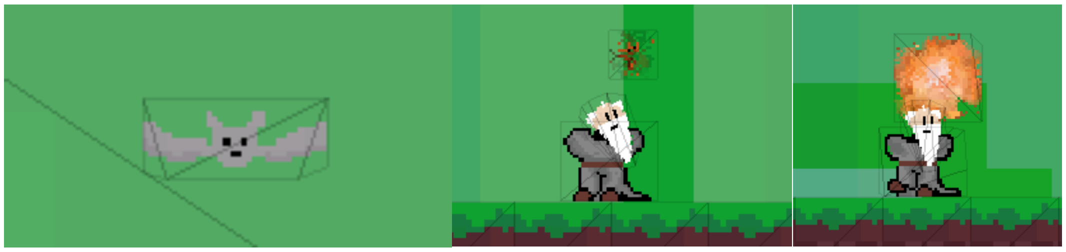
Then, the process of creating mobs started again. This time, however, I decided to make something more complex to match the new skills and knowledge I had acquired in the over four months since creating the first. The final product was a flying Bat that could shoot exploding fireballs. The logic for this one was relatively simple. The Bat was dependent on three game objects. The bat itself, the fireball projectile, and the explosion object. For the Bat, I made an object with a rigid body and a collider and created the logic script.

In this logic script, I wanted to make the bat bob up and down to simulate it flapping its wings while flying. To do this, I made a repeating function inside the script which would give the bat object a forward push in an upwards vector every 1 second. Then, to get it to hover at a desired height, I made it so the power of the push scaled to the y-axis positions of the bat object in relation to the player plus a set offset. In this way, the bat would naturally be pulled down by gravity and then pushed up again by its logic. It would then, when the distance vector between the player and itself was under a set distance, begin to lower or raise itself until it bobbed just a bit above the player.

At this point, I just needed to add horizontal movement so the bat could track the player. To do this, I just manually set the horizontal velocity towards the player when they entered the set range.

Then came the process of adding attacks. I decided that I would make it shoot projectiles as it wouldn’t make sense for it to use physical attacks.

I then made logic on the bat that would spawn the fireball projectile object with a velocity pointed towards the player. I then added logic onto the fireball itself so that when hit by something, it would destroy itself and spawn an instance of the explosion game object, which when called, would play an explosion animation and then destroy itself once complete, while at the same time for the duration of the animation would damage anything that was in its radius. However, since the animation lasted multiple frames, it could therefore damage the same entity multiple times. To fix this, I changed two things. The first was to give a force push to any entity inside the range of the explosion when they were damaged to push them out of the explosion range. The second was to add a whitelist which contained all of the entities that had already been hit by this specific instance of the explosion, and then would check that before damaging an entity, making it so it could not damage the same entity more than once.



Demonstration of the bat mob and its attack.

While testing, I had just been using a still image for all of these components' sprites, but now having completed the logic, I returned to Aseprite and began animating. I made the bat flaps, its wings, and a flying animation but then also made a separate animation for when it falls. Then I created logic to tie these together, so when the bat was in the air, it would flap its wings, and when it died, it would enter a falling animation until it hit the ground and then would be destroyed.

I then put this into the list of mobs that the mob spawner could choose from and added the spawners into the (currently two) room prefabs. Then I moved on to creating blocks for the entrance and exit areas that would spawn once the player entered a room object and would be destroyed once all mobs in the room had been killed. I set this by making the objects constantly scan for items in the room with the tag "mob"(something which I had set all the mobs I created) and then if there were none, destroyed itself.

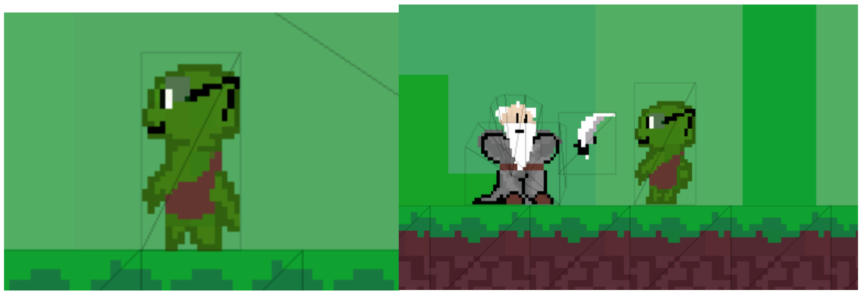
Following this, I started to think about what my game needed to become playable for the presentation that was inching closer by the day. I decided that a lot of the logic had to be updated to make gameplay better but that those updates should come after I finished adding more features and playability.

The first goal I had was to make more rooms, which I constructed from the tile map I had made previously and then incorporated them into the room logic managers. Each room had to make sense, which mainly involved making sure they could be traversed by the player movement systems I had in place. This meant a lot of testing for each room that I created, but it was also here that I became really grateful for the modular systems that I had previously set up, since they sped up the process tremendously.

Once I had completed what I considered to be a playable amount of rooms, being that it didn't feel like I was traversing through the same rooms again and again (which was also helped by now having enough rooms to access rooms from the different directions as I had set up before), I then moved on to adding the features I wanted to make the gameplay more enjoyable. First was to make a new mob.

This time I decided to go for a land-based humanoid design to cover the gap between the bat and the slime. I ended choosing to make a ‘goblin’. I designed this enemy to walk towards the player once they were within a certain distance and once they were close enough, it would attack. The attack was that I made it spawn an instance on a game object that I called ‘slash’ which had a hitbox and if it collided with the player, it would damage them, much like the explosion object I had made earlier. Using a separate object was the simplest method I could think of at the time, but it also gave some advantages that I had not thought of at that point. The main being that if a projectile was fired at the goblin at the same time that it attacked, the projectile would be stopped and not damage the enemy, since it technically was a separate object. This was good as it added an extra dimension when fighting this mob as you would have to time your shots to hit it. I then gave it the ability to jump, which was a simple comparison of the mob’s position and the player’s current position, where if the player was higher, it would jump. This meant that it could chase the player over objects to a certain extent.

Setting up the logic for the mob was relatively simple, but the part I always found hardest was to make the art for both the mob and also an animation for its attack and walking. I think somewhere here I realised that I wasn’t all that good at making sprites or animations, so I just decided to make them silly, and in a way that while still being recognisable, was fun to make for me as well. I’m glad I made this decision as even though the sprites and animations are not perfect, it reflects a style that I find joy in, and I believe that counts for something.



Demonstration of the goblin mob and its attack.

Then I turned to the issue of the actual gameplay. At that point I still had one issue that had been on my mind, that being, because of the way I had made the weapon only shoot one projectile. It was not only dishonest to its title of a shotgun but also made gameplay quite boring. I therefore set out to figure out how to make it function more like a shotgun. The solution I settled on was to simply shrink the projectile, then, when the gun was fired iterate positions within an arc of the vector the gun was pointing in, then spawn multiple instances of the original slug projectile each with one of the vectors applied to them as an impulse push.

Sound effects, background music and UI

Here is where I decided I was done with the gameplay side of the game, and while it wasn't perfect I believed that if you took the time to look at it closely enough, you could see the effort put in and the time spent learning this new skill.

So, I started work on the final things before I was done, this being the UI, final touches on the art and music/ sound effects.

I first started work on some more UI systems, where now the game can be paused, and a menu is pulled up which lets the player return to the menu or open a settings menu, that I planned to implement later, since I wanted a way to adjust the volume of the sounds while in game.

When I had set up the basic menu, I started work on a background image, as up until now I had just had an empty void behind the objects on the scene. For this I decided to draw a background image and then place it on a lower render level, so it appeared behind the other objects. But doing just this looked very stiff. So, I decided I wanted to add a motion parallax effect. At first, I tried to do this by manually scripting the background image to follow the player with a slight offset, but this looked very unnatural. I kept playing around with how to set up the script until I remembered that even if a Unity project is 2D, there is still an unused 3D z axis. And by instead changing how the camera displayed distance on the z axis and moving the image further back, it naturally obtained the effect I was after. This, of course, caused another day or two of bug fixing but it looked good.

I then turned towards the sound effects. Originally, I wanted to make the sounds from recordings I would take myself and then manipulate the audio to sound how I wanted it. But by the time I came to making sound, I didn't have enough time to do this, so instead I used the open-source software ChipTone, which gives you the tools to directly change frequency and wave forms of sounds. using this software, I created audio clips for all the most important actions that were performed in my game, i.e. jumping, shooting, reloading, taking damage, and same for the enemies too. Due to the simple nature of the sound program I was using all of the sounds ended up being somewhat similar, which helped create uniformity.

To implement these sounds that I had created, I made a ‘soundFxManager’ object which had a public singleton instance of a script, meaning that I could access the public functions in that script from anywhere without having to create a reference to the scripts object. And inside this script I created a function that when called would take three parameters: the sound clip, the volume and the position, then would create an instance of a blank game object with a sound component and give it the entered parameters. This meant that from any script when certain conditions were met, I could just call this function and play the sound clip I gave it. This way the only scripts that would have to reference to the sound clips, were the scripts that had to use them.

Once I was satisfied with this, I moved on to the background music. At first, I wanted to use REAPER which is a full DAW (digital audio workspace) but after some time trying to get it to work, I realised that it simply was too complex for the goals I had. As since I had never created music before I just wanted to make something simple that could blend into the background. So I decided to find something simpler, and here I found two options BeepBox and Bosca Ceoil, both were simple but still had a diverse set of instruments you could use. With these I managed to create a handful of songs, each only around one minute, that could be played in the background while playing.

From here, I no longer aimed to add new features and simply refined the ones that were already in place.

Artificial Intelligence in Game Development

While working on this project, I came to learn of how AI was threatening so many jobs related to programming, and therefore I thought it could be relevant to mention my thoughts on the matter in relation to game development.

Programming in general is one of the industries that has been impacted the most by AI, almost all AI nowadays are trained on immense amounts of programming training data to the point where they can create whole complex scripts by themselves, their affect has already been seen in the job markets for programming positions where massive layoffs in favour of AI assistants have been the norm for the past while.

However, the game industry has not been as significantly impacted by AI. I believe this is due to the inherent nature of game development itself. Making a game is almost like an art form where it can be used to express emotions. I thus believe that an AI which is only trained to perfectly replicate the most statistically correct answers will never be able to truly make an enjoyable game to play, as the point of a game is not to be as efficiently programmed to the standard as possible, but rather to have an interesting playing experience.

What I’ve Learned/ what others should know

Math

During my time programming specifically for game development, I’ve become far more familiar with working in a coordinate system. And although the coordinate system used in game development and computer systems in general is different from the traditional one (in computer systems, the top left corner is 0,0 and the y axis is inverted, so the bottom left corner would be the max on both axes), it has still taught me a lot about working with 2 and 3-dimensional vectors.

I have also come to notice that, when I find myself stuck on a math problem nowadays, I will readily know how I could solve it with programming, and then I just need to find out how I translate that into an equation or function. With this new approach I notice that I become stuck on problems far less than before.

Consistency

Being able to just sit down and work on something productive for a large sum of time was something I struggled with, but having this project where I ended up spending hours a day researching for or working on this project, has helped me develop that ability.

Expectations

When starting I always tended to set my expectations far too high, this would leave me disappointed when what I planned would take me a week, ended up taking months. And while this is still something I struggle with even now, I have come to find that I can far more accurately predict how long it will take to complete tasks and therefor plan accordingly.

Tips

The following is just some small things that I personally wish I had known beforehand/ knew that I should look into.

Unity referencing

Referencing in Unity was by far the hardest part of the learning experience for me. Referencing is when you want to perform an action on a different object or script, where in Unity, you need a reference to that object in order to be able to make these actions. There are multiple different ways to reference objects and scripts, but the most optimal one to use will depend on a few factors:

1. If you want to reference a component of an object that the current script is on:
   GetComponent<T>() where you enter the object type of object you are after where the T is.
   There is also GetComponentInChildren<T>() and GetComponentInParent<T>() when working in a hierarchy.
2. If you want to get an object that is in the current scene:
   GameObject.Find() which searches the scene by name
   GameObject.FindWithTag() which searches the scene by type.
3. If you want the simplest, drag and drop method:
   Creating a public or [SerializeField] private variable will create a viewable field in the unity object inspector menu, where you can drag in an object or component and then use that in the script.

4. And finally, if you want to make the contents of a script to be universally accessible:

    Setting a public singleton instance inside of the script. This means that just by referencing the script name you can access all of this public variables and functions without having to use any of the previous measures, but this should only be used on scripts that you are sure there will only be a single instance of on the scene.

Debugging

My biggest tip to debugging when there is no clear error message is to use the debug.log function at each step of the process where something goes wrong. This of course is only possible if you have a good understanding of how each part of your code works, so try to remember the general layout of your scripts, and how they are supposed to interact with other scripts or objects in the scene.

By using logs, you can narrow down an error from multiple scripts to a single function and then down the individual condition or statement that is failing.

The simplest way to do this is to just put the log statement under whatever other statement is not behaving as it should, and this way, depending on if you see the log message appear in the terminal, will tell you immediately if the error lies with the statement itself, or if there is no message then you know there is a problem with the logic that precedes it.

What I plan to do in the future

Originally I wanted to continue work on this game for a while after this project had ended, and while it is still my intention to add more features to this game, I have come to find that the biggest bottleneck in my development is that allot of my game is built on logic from scripts I made when I had first started, and therefore from a time when I was not yet competent at making modular scripts and objects that would be easy to work with in the future.

So, while I do definitely plan on continuing work on game development, it will most likely be in a new game with a fresh start, where I can use all the knowledge I have gained to build a better foundation for a new game.

But also, I believe that the practical problem solving that I have learned from this project will be useful regardless of what my future endeavours may be.

Final thoughts

Looking back, I have learned so much from this experience both in the fields that I hoped I would learn new things in, but also in areas that I never thought I would have to learn about when I started this project.

And when I think of the process that I have gone through, I always feel there is something I could have done better, but now, I have come to believe this is just a sign that I have grown.

Sources list

Andrea Knezovic, "The Complete History of Video Games 1952 – 2026," Udonis Blog, January 12, 2026, https://www.blog.udonis.co/mobile-marketing/mobile-games/history-video-games , accessed November 13, 2025.

Datapath.io, "The History of Online Gaming," Medium, 2017, https://medium.com/@datapath_io/the-history-of-online-gaming-2e70d51ab437 , accessed November 16, 2025.

GamesFromScratch.com, "Game Engine Popularity in 2024," GameFromScratch.com, January 29, 2024, https://gamefromscratch.com/game-engine-popularity-in-2024/ , accessed December 2, 2025.

Jane Wakefield, "How the Computer Games Industry Is Embracing AI," BBC News, May 2, 2024, https://www.bbc.com/news/business-68844761 , (accessed February 13, 2026).

Mark J. P. Wolf, "History of Video Games," EBSCO Research Starters, 2024, https://www.ebsco.com/research-starters/history/history-video-games , accessed November 13, 2025.

RocketBrush Studio, "The Evolution of 3D Graphics and Its Impact on Game Art Styles," RocketBrush Blog, March 4, 2024, https://rocketbrush.com/blog/the-evolution-of-3d-graphics-and-its-impact-on-game-art-styles , accessed November 15, 2025.

Tom "PyTom" Rothamel, Ren'Py Visual Novel Engine, version 8.5.2 (https://www.renpy.org/ , accessed February 13, 2026).

Wikipedia contributors, "Construct (game engine)," Wikipedia, The Free Encyclopedia, https://en.wikipedia.org/w/index.php?title=Construct_(game_engine)&oldid=1320007825 (accessed February 13, 2026).

Wikipedia contributors, "GameMaker," Wikipedia, The Free Encyclopedia,https://en.wikipedia.org/w/index.php?title=GameMaker&oldid=1336235506 (accessed February 13, 2026).

Wikipedia contributors, "Unity (game engine)," Wikipedia, The Free Encyclopedia, https://en.wikipedia.org/w/index.php?title=Unity_(game_engine)&oldid=1333900879(accessed November 23, 2025).

Wikipedia contributors, "Unity Technologies," Wikipedia, The Free Encyclopedia, https://en.wikipedia.org/w/index.php?title=Unity_Technologies&oldid=1337692308(accessed February 13, 2026).

Wikipedia contributors, "Unreal Engine," Wikipedia, The Free Encyclopedia, https://en.wikipedia.org/w/index.php?title=Unreal_Engine&oldid=1338082598 (accessed February 13, 2026).