Abstract

This project investigated the task of creating a science fiction video game using game development studio Godot Engine v.4.0.3 (Linietsky et al. 2023), following current video game industry practices and an agile development methodology. There was a focus on creating a video game world with a focus on diversity and cultural inclusivity through its story and characters, and this dissertation seeks to explore current issues in diversity and inclusivity in gaming.

The end product is evaluated in this dissertation, alongside the development methodologies and software used. This dissertation will add to the small (but growing) collection of academic work on the topic of diversity and inclusivity in video gaming, further pressing and exploring an issue that I believe is becoming more and more salient in today's video gaming world.

On the whole, this project was a success, with the majority of the aims covered in this dissertation implemented into the game and said game is a prototype of something much larger and so is a solid foundation for further development.

Contents

Acknowledgements

I would like to thank my amazing boyfriend, Luke, for putting up with me, my weird sleep schedules, and strange work hours during this project.

I would also like to thank my project supervisor, James Osborne, for his endlessly useful advice, input, and reassurance.

I am also very grateful to have been a student at Cardiff University for a combined four years – the institution has tremendously helped develop my skills and person, and I am eternally grateful to all the friendly staff in the schools of ENCAP and COMSC for their amazing courses in BA English Language and Linguistics, and MSc Computing, respectively. Gwirionedd, Undod a Chytgord!

Introduction

In this dissertation project, I undertook the task of creating a science-fiction themed video game with a focus on diversity and cultural-inclusivity, using an agile development methodology. This dissertation evaluates the key issues in diversity and representation in video games; discusses standard industry practices in software, and more specifically, video game development; and thoroughly examines and reflects upon my own game development journey, including my methodologies and experiences.

I believe this dissertation to be important and its themes salient due to current issues with diversity in the gaming industry today, the lack of academic literature surrounding diversity in gaming, by how it furthers my own programming skills, adds to my programming portfolio, and contributes to the small but growing library of inclusive games out in the world, especially those developed by diverse developers.

To begin with, I outline the aims of this project, before exploring its background, including the current state of diversity in gaming, current trends in game development methodology, and a discussion of available game engines. I then examine and analyse my game through its various stages and its development, talking through the steps and decisions I made, the problems and bugs I faced, and how I overcame them. Lastly, I evaluate the project, critically analysing my development process and the end product, before concluding with a round-up of everything this dissertation found, and a more personal reflection on this dissertation, the choices I made, and my own development as a person.

Aim

The aim of this project is to create the foundations of a sci-fi themed video game with a focus on inclusivity and cultural diversity; to add to the small, but growing library of video games with such a focus (see page 55). The objectives I set myself were chosen to complete the project aim as closely as possible, and I consider the project to be successful if they are completed, and are as follows:

- The game must be developed in increments, following the industry-standard Agile development methodology, with the path of development changing based on requirements and suggestions from my supervisor.
- The game must include a cast of culturally diverse NPCs (non-player characters), with whom the player can talk, using a branching, waterfall approach to dialogue.
- The game must include industry-standard gameplay features, from simple, staple systems like player movement, health, and death, to more advanced, genre-specific features, like randomised map generation and branching dialogue.
- The game must include sci-fi themes and tropes, portrayed through dialogue, gameplay, characters, and art.

Background

The State of Diversity in Gaming

Being both a gay man and a gamer, I often find that the two traits do not meld together well. From the absence of same-sex romance options in mainstream games, rampant homophobia, sexism and racism in multiplayer chats and gaming forums, to a lack of representation of minorities in games, it is not an unfair judgement that gaming has a reputation for being a predominantly straight, white male hobby.

Video games published by large, well-known game development companies producing high-budget video games, also known as 'triple A' or 'AAA' companies, have a history of both breaking new ground in inclusivity and diversity in gaming, and also falling for harmful stereotypes or being outright racist, homophobic, sexist, and transphobic. It was not until 2022 - 10 years after its release – that Grand Theft Auto V (Rockstar Games 2013), the third-most sold game of all time (Sirani 2023), removed overtly transphobic content from its game despite years of complaints from the trans community (Bailey 2022). Take, as another example, Hogwarts Legacy (Avalanche Software 2023), a video game whose release was mired in contention due to controversy surrounding its franchise's creator, J.K Rowling, and her outspoken views about the trans community (Ward and Ross 2023), which caused many to boycott the title. This is not to say that this is not the case with all video games produced by AAA companies; many popular titles today are filled with inclusivity and diversity, which for the most part is popularly received by consumers. Take for example Blizzard Activision's Overwatch (2016), which features a highly diverse cast of playable characters that avoids the use of harmful stereotypes: 'of the 24 playable heroes, half are white, half are not, about half are women, about half are men, and there's even non-gendered options' (TheGamer 2017). This level of diversity in AAA games is important as it gives the majority of players the chance to feel represented and connected to a character and shows a world where the hero does not have to be a straight, white, cisgender male every time. In the case of my own game, it is therefore important to include a diverse cast of characters, to ensure that as many players as possible feel represented and connected.

Many indie game companies have been trying to break the reputation of gaming being a straight, white, male hobby with the creation of games with a focus on diversity and inclusivity, or with the creation of minority-founded-and-led game studios, such as Lumi Interactive, an Australian queer-female-founded indie studio, or AipoDigital, a Brazilian game studio who focus on LGBT+ narratives (Rogers 2022). There is a push in the video game industry to hire minority game developers, with websites like BlackGameDevs.com (Agloo et al. 2023) existing to promote black game developers worldwide, or charities like GaymerX (2013) who are pushing to get ‘more LGBTQ+ individuals into the industry’.

It is important for video games to include diversity and cultural inclusivity for two important reasons: it is a means to inform and influence a large audience; and said audience is constantly diversifying and reflecting the global population as a whole, rather than a subset of it.

For the first reason, it is important to note that it is estimated that there are over 3 billion active video game players globally (Howarth 2023; Stojanovic 2023); games reach almost half of the global population, and as such, are key in relaying ideas and narratives to a wide audience that can make players think differently about topics such as diversity and inclusivity. Studies have shown time and time again that media consumption affects how people think (Entman 1989), be it through the formation of new ideologies, or changes and challenges to existing ones. Entman (1989, pgs. 349-351) describes how media can be used to help consumers ‘accommodate new information’, and that a consumer ‘may stimulate new beliefs or change old beliefs’ when introduced to or challenged with new ideas. Entman also states that adoption of new ideas can occur even ‘when information is contained in the form of a subtle slant’, which is salient to this dissertation project as a video game does not even need to be explicitly about tackling racism, sexism, or LGBTQ+ issues – just including a diverse cast of characters is enough to promote diversity and inclusivity.

For the second reason, as time has gone on and gaming as an entertainment medium has become more widespread and diverse, so has its audience: in 2022, the Entertainment Software Association (ESA) stated that 48% of gamers in the US are female-identifying, up from 45% in 2020 (ESA 2020), and that non-white gamers are on the rise, with 29% of American gamers being Hispanic, African-American, Asian, Pacific Islanders and other ethnicities, up from 27% in 2021 (ESA 2021). Players naturally want to feel represented in the games they are playing; Cicchirillo and Appiah (2014) found that players identified with characters in video games that were most similar to them in terms of identity, and highlight the prevalence of negative stereotypes and the harm they can cause, and with video games players clearly becoming more diverse, it is important for games to reflect that, and become more diverse and inclusive while also beating harmful stereotypes.

(For a list of diverse and culturally inclusive sci-fi games, see page 55)

Standard Industry Practice in Game Development

Game development is a complicated, unpredictable process that differs drastically per project. Numerous factors contribute to the methods and ideas game development is undertaken using in any given project.

The literature overwhelmingly agrees that game development and its requirements are very different to standard software engineering and its requirements (Murphy-Hill et al. 2014, pg. 3; Marklund et al. 2019, pg. 189; Aleem et al. 2016, pg. 5); ‘games have specific characteristics, which the conventional software development process cannot completely address’ (Aleem et al. 2016, pg. 17). Aleem et al. write that requirements of a video game include unique factors, separate from traditional software development, such as ‘emotion, game play, aesthetics, and immersive factors’, and that their understanding is key so they can be incorporated while developing games.

The game development process can be based on the requirements of a game or its studio according to Marklund et al. (2019, pg. 188-9). Murphy-Hill et al. (2014, pg. 3) write that ‘in essence, games generally have one and only one requirement – that they are “fun”’, although the idea of what constitutes ‘fun’ is difficult to define; Marklund et al. write that ‘fun’ is something defined by the developers themselves in smaller studios, but in larger studios, is defined by using ‘identified subjective preferences and usability concerns of their target audiences as a guiding requirement’. It is important to note that, in game development, what a designer imagines would be fun, when implemented by artists and programmers, may not actually be fun in practice (Murphy-Hill et al. 2014). Aleem et al. (2016) argue that the idea of ‘fun’ is subjective and largely dependent on player or target audience. A great difficulty arises as the idea of ‘fun’ is difficult to define, yet creating a fun video game is paramount to its success; Buday et al. (2012) write that fun is essential in creating a successful video game as ‘making a boring videogame is one way to alienate players’, and that ‘fun is an entertainment game’s payload’, even stating that the overall technical quality of a game is largely irrelevant if, ultimately, it is still fun to play, as ‘fun but ‘buggy’ videogames … can do well in the marketplace’, clearly emphasising that commercial success is directly tied to how fun a game is perceived to be. With that in mind, in the case of my own game, to create something that would be successful if put to market, I needed to create something fun, and ‘fun’ is something I had to define myself as a solo developer, based on my own preferences in video games, and what I feel is ‘fun’, while keeping within the realms of my technical knowledge, limited available resources, and the limited timespan of the project.

Aleem et al. write that, in Osborne-O’Hagan et al. (2014)’s study on software development processes for games, it was found that agile practices were more common than hybrid approaches in game development studios. Aleem et al. state that 'lightweight agile practices such as Scrum, XP, and Kanban – are suitable where innovation and time to market is important'. This is especially relevant in the case of my own project; I worked using an agile methodology, and with only a short timespan to work on it, one could argue that my 'time to market' was quite rapid. I talk more about my use of the agile methodology later in this dissertation, on page 11.

Phases/stages of development are commonly used in the game development industry (Murphy-Hill et al. 2014, pg. 2), as a way to separate tasks logically across a timeline, to set goals based on time requirements, and to 'keep studios running efficiently' (Pickell 2019). Generally, the literature describes anywhere from five to seven phases, which I will condense to the five common phases that are universal across the literature I have read: a 'pre-production' phase, a 'production' phase, a 'pre-launch' phase, a 'launch' phase, and a 'post-production' phase (Pickell 2019). See the below figure for a diagram of each phase:

Pre-production	Production	Pre-launch	Launch	Post-launch
• Meetings between departments • Research of subject matter • Planning, ideation, idea refining • Formation of requirements • Creation of a game design document • Concept art • Hiring and sourcing artists, designers, writers, developers, team leaders, etc.	• Continuous meetings, standups, scrums • Creation of assets (e.g. 3d models, textures, sprites) • Writers and designers create the game's story and world • Functionality is programmed by developers • Continuous testing	• Alpha testing (internal tests) • Beta testing (external tests, 'closed beta', 'early access') • Feedback is compiled • Game changes based on feedback • Debugging game - breaking bugs caught by alpha and beta tests	• Public release, either global, or staggered in date/time by regions and platforms • Game becomes available for sale by vendors online and in - person • Multiplayer servers activated	• Hotfixes, patches • Continuous server maintenance • Community feedback is continuously gathered • Development of extra game features in the form of monetizable DLCs (downloadable content), microtransactions and free content updates

Figure 1. *The five stages of game development, beginning on the left, and progressing to the right.*

The pre-production phase consists of the formation of initial plans for a video game, including ideation, storyboarding, creation of concept art and a game design document (GDD), and the holding of meetings between different departments within a game development studio to work out requirements, game content, and what is feasible or realistic to the studio's available resources. Writers may have ideas that are too grandiose or outside the realms of technical ability for the programming team; designers have to work with artists to prototype assets according to the designers' specifications; and programmers have to work with artists to discuss how their assets and UI will be implemented and programmed in (Pickell 2019). A game design document is created at this stage (Aleem et al. 2016) which is a document that details the overall plans for the game, acting as a means to express core ideas to all departments, and is used and developed upon throughout the whole game design process. I talk more about my use of a GDD later in this dissertation, on page 13. I spent around three days in this phase at the very beginning of my project, ideating and then creating the GDD.

Pre-production is followed by the production phase, wherein development of the game itself begins, and where ‘most of the time, effort and resources [are] spent’: character models are designed to look exactly how they should, as per the designers’ specifications; developers write the game’s code; audio is sourced and programmed in; level designers create levels to ensure difficulty and theming is appropriate; etc. (Pickell 2019). It is not uncommon for projects to spend years in this phase. In the case of my own game, I spent a little over a month in this phase, which was enough time for me to create sprites and assets including the UI, design a simple level, and program numerous systems such as player movement, hitboxes, weapon-use, death, multiple-choice dialogue, and score/money handling.

The pre-launch phase then follows, which includes thorough testing of the game’s systems through internal testing (alpha testing), and then usually external testing (beta testing) which is sometimes advertised as ‘closed betas’ or ‘early access’ which can in itself generate revenue and attention. This phase is crucial to ensure the game releases in the best quality it can be, and to ensure there are no obvious, game-breaking bugs or missing assets or features.

The launch phase then follows, which sees the game’s release to the public. Generally, this phase is short, as it is immediately followed by the post-launch phase, where hotfixes are implemented to quash important post-release bugs, and patches are developed to quash minor bugs, and sometimes to include new or late content. In the months and years that follow, the game should receive regular software updates to ensure compatibility with new devices and operating systems, and maintenance to servers for online games should be undertook. Sometimes, new content is released in updates or even DLC (downloadable content), where the studio can generate more revenue by selling additional content for their game, post-release (Pickell 2019).

Game Engines

(Disclaimer: the first two paragraphs of this section are quoted from a prior piece of work I completed: the proposal for this dissertation (Fowler 2023)).

> According to Andrade (2015), in game development, the choice of game engine can make or break a project. It can influence (or is influenced by) the choice of game being created, what language a project is coded in, and its graphical, logical, and processing capabilities. He states that there is no such thing as 'the best game engine', and rather, the developer should ask themselves what their requirements are. He explains that newcomers should 'take the time to experiment with a couple of different options and get a feel of different editor interfaces and game abstractions.'
>
> Andrade lists 13 game engines, including popular, industry standards like Unity (Unity Technologies 2005), CryEngine (Crytek 2002), and Unreal Engine (Epic Games, originally Sweeney 1998). He recommends for indie developers to use Unity due to it being free for smaller developers. However, being almost a decade old, it is to be expected that some of the engines mentioned are depreciated, or newer engines are not listed. Vohera et al. (2021) also compares Unity, CryEngine, and Unreal Engine, but recommends Unity for novice developers due to its 'very well-written documentation, several courses, and ready-made templates'. While not surprising in the case of Andrade's paper due to its age, I was surprised to find no mention of the increasingly popular Godot game engine (Linietsky et al. 2023) in Vohera et al.'s paper, which is popular among novice developers due to its speed and ease-of-use, or Roblox Studio (2006), which is becoming a popular game development engine with potential for making profitable games. There is clearly no real 'winner' – the literature agrees that each engine has its perks, but as a novice indie developer, Unity, Roblox Studio and Godot are what I believe to be my best options.

Unity game engine is used by professional game developers around the world, and according to Toftedahl, writing for Game Developer (2019), Unity is the second most used development engine in the industry, with Unreal Engine being the most used. However, Unity is a more viable option for me as it is a lot more beginner friendly; the scope of this dissertation project would not require me to learn more advanced programs, or C++, the programming language Unreal Engine uses. While Unity is beginner-friendly, there is a lot of room for more advanced development, too.

While I would not have considered Roblox Studio for this project a few years ago, before Roblox 'hit the mainstream', Roblox, as of time of writing, boasts an average overall monthly player count of 202 million (ActivePlayer.io, 2022), and the top games created using Roblox Studio earn millions of dollars a year in microtransactions, as claimed by Roblox (2021). Roblox Studio is a tool with which I have years of experience using, and the scripting language it uses, Lua, is a language I am very familiar with. However, creating a game that strays from Roblox's default 3D, Newtonian physics, third-person, multiplayer setup requires a lot of boilerplate code, and even then, the end-product would be a 2D game system that runs in a 3D world; performance-wise, creating this game in Roblox would result in a poorly optimized experience. Additionally, as Roblox games are hosted on an external server, the user would need to be constantly connected to the internet to be able to play a game which I plan to be single player, would have to download the game from the Roblox servers on each play, and would also need to create an account with Roblox, imposing numerous limitations and hurdles which would harm the gaming experience and render it inaccessible to some players, especially those from countries where Roblox is unavailable. As I have a lot of experience with Roblox Studio and wanted to challenge my programming and game development skills, and because of reasons stated above, I decided against using Roblox Studio, in favour of Godot.

I chose to use Godot due to its advertised accessibility for beginners, modernity, and ease of use when creating 2D games specifically. My choice of Godot as a development engine came with advantages and disadvantages, as I expected to face with any game engine, but I believe it to have been the right choice overall. In terms of advantages, Godot stayed true to its reputation: I did find it to be very simple to learn. The program organised the structure of my game into 'nodes', giving each element an easily navigable hierarchy (parent/child structure). This is like Roblox Studio's object hierarchy system, so I felt familiar with this way of organizing the project. Godot offered its own IDE (integrated development environment) for writing scripts, which was endlessly useful as it tied itself to the game editor seamlessly and saved me from having to write my code in a separate program, only to have to import it across. It also offered its own programming language, called GDScript, which was designed specifically for use in Godot (I talk more about GDScript on (**page**)). Godot also came with a multitude of node 'types', which took some time for me to fully understand as they differed from Roblox Studio's types. I had to learn to understand that, just because a node may visually appear in the game world, it does not necessarily come equipped with a prebuilt physics system, unlike most of Roblox Studio's built-in objects. 'Animated2DSprites', for example, must be children of ‘KinematicBodies’ to be able to move and collide with other bodies, which differs from Roblox Studio where, in most cases, if an object appears in the game world, it will be affected by a prebuilt system of gravity, collision, etc. unless specified otherwise.

An issue I faced a couple weeks into development was the Godot Engine website being down for a month. This caused issues as the Godot Engine website hosted the official support forum, wherein developers can create or look at past questions about the engine. I was able to somewhat fix the issue by viewing Google’s cached versions of the website, but not every post had been cached, rendering some completely inaccessible, and due to it being a cached version of the website, I was unable to ask questions myself. This made searching for queries relating to the engine difficult, as at times I was forced to find workarounds for certain issues. I was unable to find out why the website was down until late July, when the website suddenly became available again, and a message at the top of every page had appeared informing me that they are in the process of migrating the forum to a new platform, and that the forum was read-only. (As of submission (11 September 2023) the forum is still read-only).

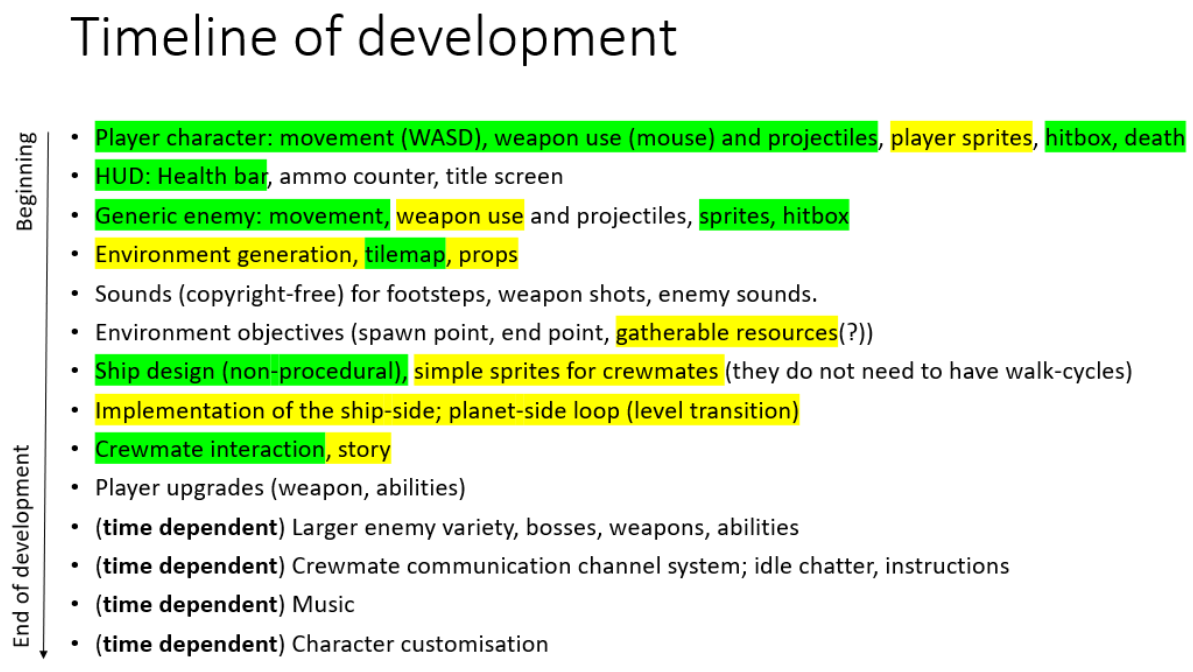
Academic literature on programs to use for specifically creating art for video games is scarce, and so for this reason, I chose to use what I was most familiar with, and this was the digital art studio, Krita (2005). Krita is a free, open-source digital art studio which allows for high-level creation of art. Krita is not made specifically for pixel art, lacking a concrete ‘spritesheet’ tool, but still allows the user to create art at smaller resolutions and export them as .png files, to preserve transparency layers. All artwork I created for the game was designed in Krita.

Approach and Implementation

Agile Development

The agile development methodology is a development strategy used in game development that 'involves breaking the project into phases and emphasizes continuous collaboration and improvement', with teams following 'a cycle of planning, executing, and evaluating' (Atlassian 2023). It emphasises the importance of regular meetings between team members, continuously iterating, reviewing and evaluating work. 'Standups' are usually performed, in the form of regular meetings where team members (generally the developers) discuss their recent work, and outline what they are going to do next; 'scrums' are also performed, where team members meet at the end of a development cycle for one iteration of the project, where the project is evaluated and next steps are formed, which may be returning to previous work done, or beginning work on a new feature. The time between scrums is called a 'sprint', where team members work towards a time-sensitive goal, be it developing new features or reworking old ones.

This dissertation project followed the agile development methodology, to ensure the direction of the project remained flexible based on changing requirements. My project supervisor was effectively my client. He helped set requirements for the project; we met often, and in those meetings, I would show progress on the game and discuss issues I had with its development, and his feedback was paramount in deciding what direction to take next. I used a project timeline (see figure 2 below), to mark progress, completed features, and features that were actively being worked on, which are strategies used in Kanban, a sub-strategy of agile where work items are represented on a kanban board (or in my case, a timeline), which is used to visualize project progress, backlogs, and planned developments.


Beginning
End of development

Timeline of development

- Player character: movement (WASD), weapon use (mouse) and projectiles, player sprites, hitbox, death
- HUD: Health bar, ammo counter, title screen
- Generic enemy: movement, weapon use and projectiles, sprites, hitbox
- Environment generation, tilemap, props
- Sounds (copyright-free) for footsteps, weapon shots, enemy sounds.
- Environment objectives (spawn point, end point, gatherable resources(?))
- Ship design (non-procedural), simple sprites for crewmates (they do not need to have walk-cycles)
- Implementation of the ship-side; planet-side loop (level transition)
- Crewmate interaction, story
- Player upgrades (weapon, abilities)
- (**time dependent**) Larger enemy variety, bosses, weapons, abilities
- (**time dependent**) Crewmate communication channel system; idle chatter, instructions
- (**time dependent**) Music
- (**time dependent**) Character customisation

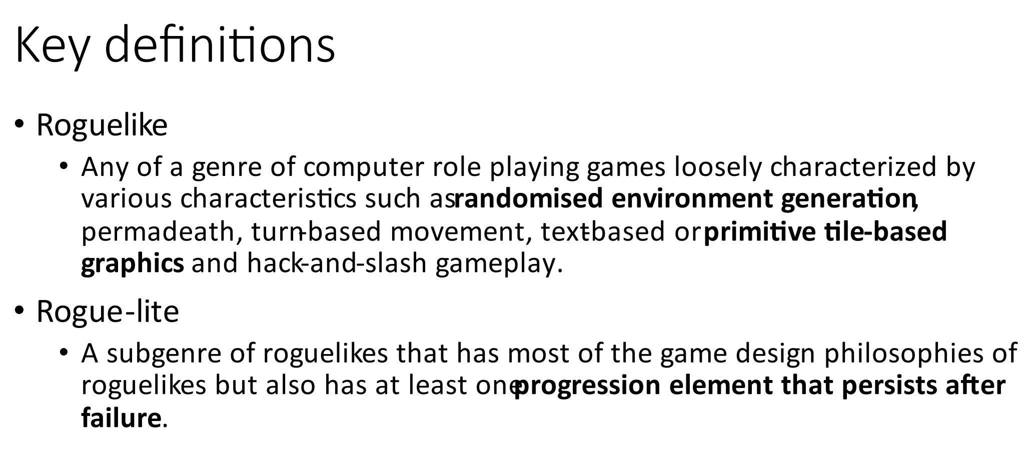
Figure 2. *The timeline of development, which I used to visualise my progress. Green highlighting indicates a completed feature; yellow indicates a feature in process; unhighlighted indicates a feature that is yet to be started.*

Agile was essential in the management of the project as reacting to feedback given in meetings ensured I was developing features that were deemed as necessary and was steered away from developing features not viewed as a requirement by my supervisor, which I believe helped create an overall better product. For example, early on in the project, I was encouraged to stop focusing my development efforts on creating more sprites for the player character as it was already satisfactory, and to instead begin the level design; later on, I was encouraged to develop a player character death system, alongside a currency system. This strategy allowed the game to develop in smaller, regular increments, and stopped overdevelopment of unnecessary features. Tracking my progress via the timeline helped visualise progress for both me and my supervisor, and helped with defining what each iteration of development would focus on.

The GDD

Before development of the game could begin, the project began following industry-standard practice, by the creation of a game design documentation (GDD), wherein guides for the setting, characters, themes, and gameplay were developed. Typically, the GDD acts as a way to 'communicate the designer's vision to the development team' (Almeida and da Silva 2013), though with myself being both designer and developer, I utilised the GDD to showcase my ideas to my supervisor instead. Almeida and da Silva go on to say that the GDD also acts as a guide to the whole development process, which was certainly the case during development of my game as I tried to implement all the ideas in my GDD that I had written before production of the game began. The expectation was that the document would also evolve alongside the project in a reciprocal manner (Colby and Colby 2019) as designs changed during development, be it gameplay changes or artistic changes.

In the GDD, I included numerous informative sections that helped form the plans of both the overall project and also each individual functionality/feature. The first page contains two key definitions:

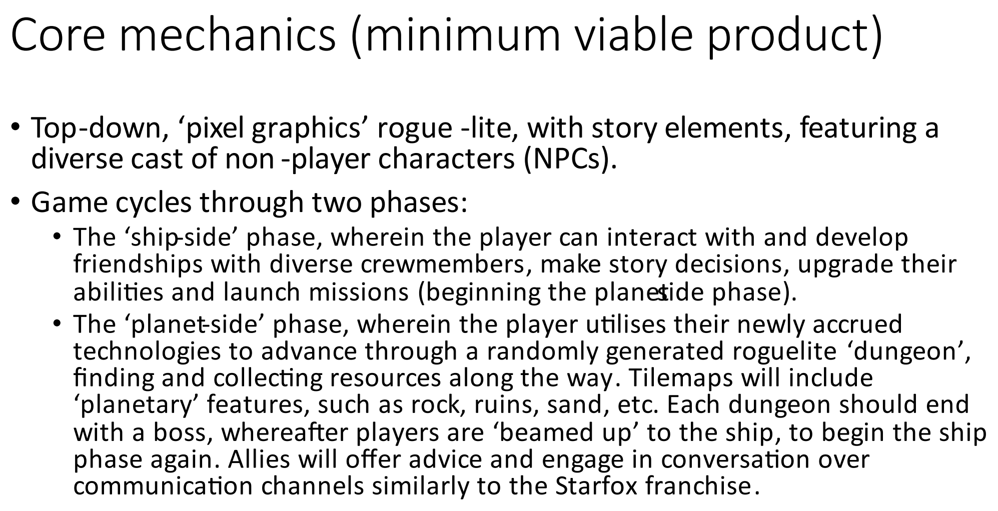


Key definitions

- Roguelike
  - Any of a genre of computer role playing games loosely characterized by various characteristics such as **randomised environment generation**, permadeath, turn-based movement, text-based or **primitive tile-based graphics** and hack-and-slash gameplay.
- Rogue-lite
  - A subgenre of roguelikes that has most of the game design philosophies of roguelikes but also has at least one **progression element that persists after failure**.

Figure 3. *Key definition of the Roguelike genre of video game, and a sub-genre of it called a Rogue-lite. (Game Design Document)*

As I was creating a Rogue-lite video game, these definitions were imperative to include in order to explain the genre to my supervisor, and also as a reminder to myself as designer, artist, and programmer of the game, as to what functionality the game needed to fit the genre. The next page went into more detail of what the requirements of the game were, in the form of a minimum viable product:



Core mechanics (minimum viable product)

- Top-down, ‘pixel graphics’ rogue -lite, with story elements, featuring a diverse cast of non -player characters (NPCs).
- Game cycles through two phases:
  - The ‘ship-side’ phase, wherein the player can interact with and develop friendships with diverse crewmembers, make story decisions, upgrade their abilities and launch missions (beginning the planet-side phase).
  - The ‘planet-side’ phase, wherein the player utilises their newly accrued technologies to advance through a randomly generated roguelite ‘dungeon’, finding and collecting resources along the way. Tilemaps will include ‘planetary’ features, such as rock, ruins, sand, etc. Each dungeon should end with a boss, whereafter players are ‘beamed up’ to the ship, to begin the ship phase again. Allies will offer advice and engage in conversation over communication channels similarly to the Starfox franchise.

Figure 4. *The game's core mechanics and minimum viable product. (Game Design Document)*

This section was, again, important to explain the planned product to my supervisor before development officially began. It outlines the core gameplay loop and how I intended the finished product to function. It outlines the two phases of the game: the ‘ship-side’, where the player interacts with NPCs on a spaceship that is considered the game’s ‘hub’ or ‘home’, wherein the player is safe from danger; and the ‘planet-side’, where the player completes missions in a randomly generated environment, slay creatures, earn money, and beat the boss. The next page gave a more visual diagram of the gameplay loop:

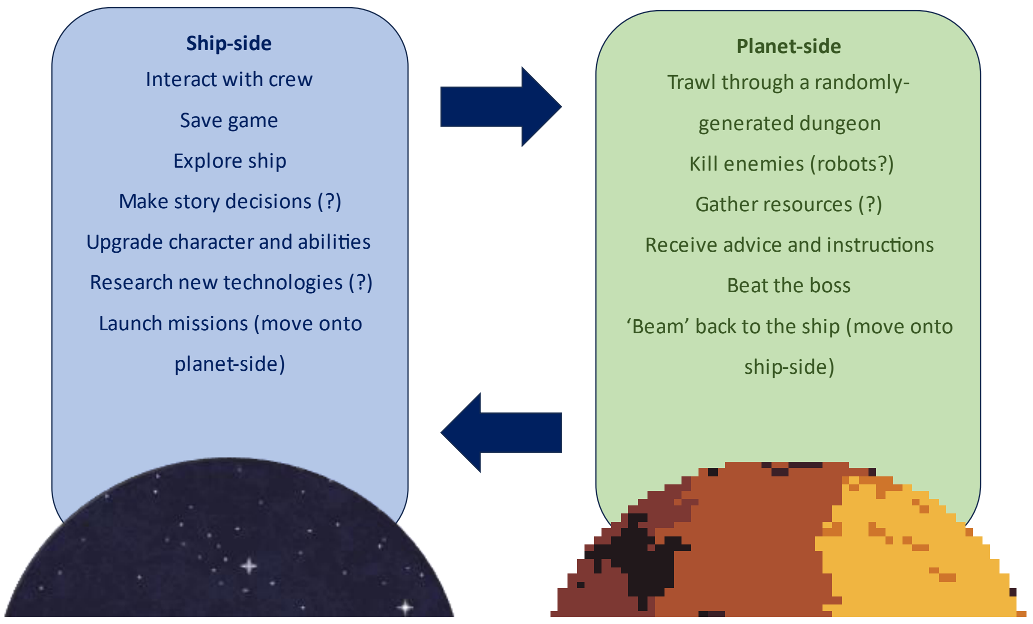
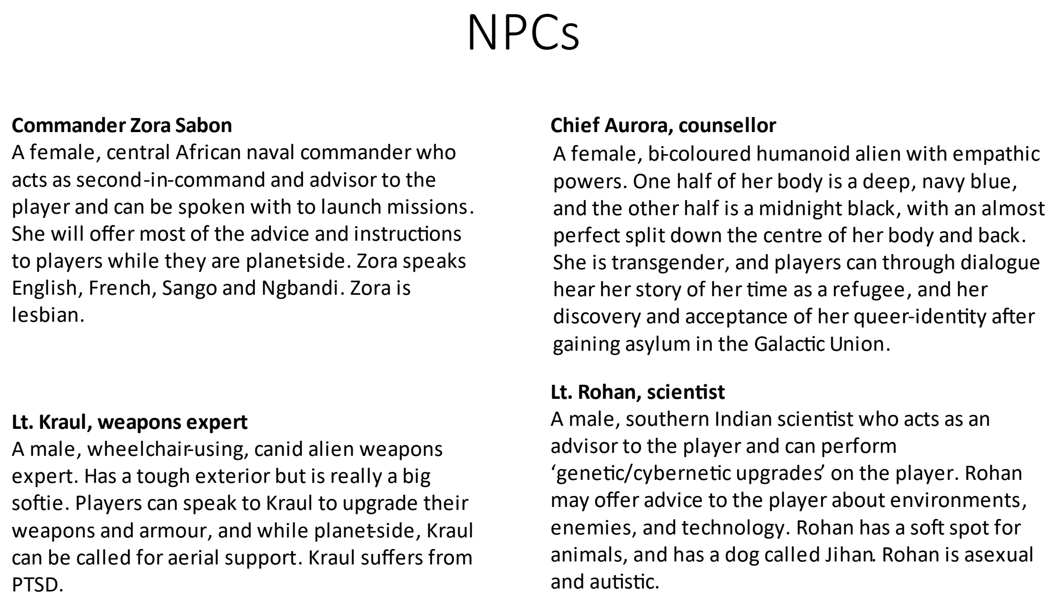

Ship-side
Interact with crew
Save game
Explore ship
Make story decisions (?)
Upgrade character and abilities
Research new technologies (?)
Launch missions (move onto planet-side)
Planet-side
Trawl through a randomly-generated dungeon
Kill enemies (robots?)
Gather resources (?)
Receive advice and instructions
Beat the boss
‘Beam’ back to the ship (move onto ship-side)

Figure 5. *A visualisation of the core gameplay loop. (Game Design Document)*

I separated the two game phases into two visually distinct sections to assist in presenting my plans and give the GDD theming and interesting visuals to help 'set the mood'. Rogue-lites are circular in nature, in that the gameplay loop (and part of their selling point!) is that you constantly repeat sections of the game, regardless of whether you win or lose, but each time it is repeated you are faced with different challenges in the form of a different map layout, a different boss, different loot and powerups, and different enemies. My game is designed around this circular pattern, encouraging the player to keep returning to the planet-side phase to face new, randomly generated missions, and to return to the ship-side phase to catch up with crew, ensuring there is a constant stream of fun things for the player to do. The next page outlines my plans for the NPCs of the game, who take the roles of the player's ship's crew:



NPCs

Commander Zora Sabon
A female, central African naval commander who acts as second-in-command and advisor to the player and can be spoken with to launch missions. She will offer most of the advice and instructions to players while they are planet-side. Zora speaks English, French, Sango and Ngbandi. Zora is lesbian.

Chief Aurora, counsellor
A female, bi-coloured humanoid alien with empathic powers. One half of her body is a deep, navy blue, and the other half is a midnight black, with an almost perfect split down the centre of her body and back. She is transgender, and players can through dialogue hear her story of her time as a refugee, and her discovery and acceptance of her queer-identity after gaining asylum in the Galactic Union.

Lt. Kraul, weapons expert
A male, wheelchair-using, canid alien weapons expert. Has a tough exterior but is really a big softie. Players can speak to Kraul to upgrade their weapons and armour, and while planet-side, Kraul can be called for aerial support. Kraul suffers from PTSD.

Lt. Rohan, scientist
A male, southern Indian scientist who acts as an advisor to the player and can perform 'genetic/cybernetic upgrades' on the player. Rohan may offer advice to the player about environments, enemies, and technology. Rohan has a soft spot for animals, and has a dog called Jihan. Rohan is asexual and autistic.

Figure 6. *Outlines of four NPCs. (Game Design Document)*

This section simply describes the key characteristics and relevance of four NPCs, whom I tried to create as diverse and interesting as possible. It also outlines what 'services' they offer; Cmd. Sabon allows the player to start planet-side missions; Lt. Kraul offers weapon and armour upgrades to the player in exchange for money the player earns on planet-side missions; Lt. Rohan offers 'genetic/cybernetic' upgrades to the player, in the form of new abilities.

Environment generation

A core theme of the roguelike and rogue-lite genres is environment generation, so that each time a player enters a 'dungeon', the map is different each time and helps keep the game fresh and interesting. Good examples of games with simple graphics that have environment generation include Nuclear Throne (2015), Rimworld (2013), Factorio (2016) and many others. For this to be successful in my game, the environments that are generated must be **natural-looking**, **navigable**, and **populated**. 'Props' can include rocks, trees/cacti, logs, bushes, ruins (including old vehicles, machinery), and must be generated using appropriate, unique algorithms so trees may spawn together in batches to create the impression of forests, but rarer props like old vehicles do not. Additionally, the map must not generate in such a way that the player cannot advance through it. A pathfinding algorithm must be used to check navigability and force regeneration if the map is untraversable.

Nuclear Throne

Rimworld

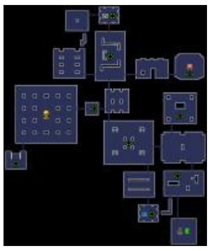
Factorio

Figure 7. *The first part of a discussion on environment generation, a core feature of Roguelite games. (Game Design Document)*

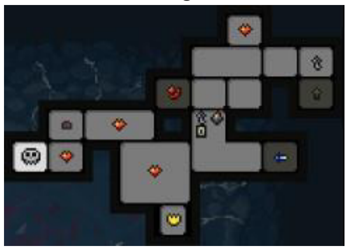
Environment generation (cont'd)

Due to time pressure and my own experience with creating environment generation, the game could feature a simpler 'room' generation system, rather than a 'terrain' generation system. This is, rooms of various sizes connected in sequence, with branching paths, with one path leading to the boss, in the style of Enter theGungeon (2016), The Binding of Isaac (2011) and Pokémon Mystery Dungeon (2005). This will also allow greater control over the design of the individual rooms as their props, spawns, etc. do not need to be randomly generated, and can be designed by hand. Either way, this system will be easier to implement.

Enter the Gungeon



The Binding of Isaac



Pokémon Mystery Dungeon

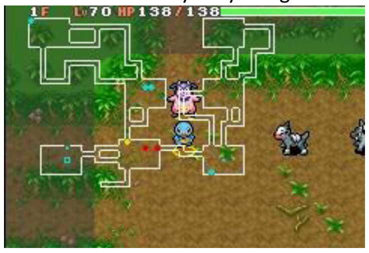
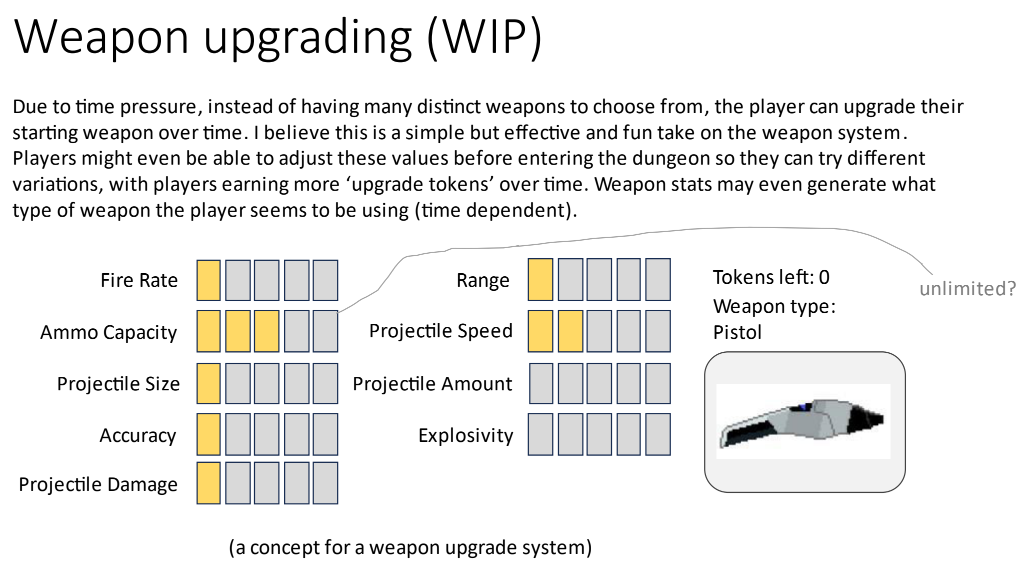


Figure 8. *The second part of the discussion. (Game Design Document)*

The above two sections discuss the environment generation of the game, with the first (figure 7) describing potential options based on existing roguelite/sci-fi games, and the second (figure 8) discussing the system I chose specifically. I talk more about environment generation on page 35. The next section moves onto plans for a proposed weapon upgrading system (the service offered by Lt. Kraul):



Weapon upgrading (WIP)

Due to time pressure, instead of having many distinct weapons to choose from, the player can upgrade their starting weapon over time. I believe this is a simple but effective and fun take on the weapon system. Players might even be able to adjust these values before entering the dungeon so they can try different variations, with players earning more 'upgrade tokens' over time. Weapon stats may even generate what type of weapon the player seems to be using (time dependent).

Fire Rate	🟨⬜⬜⬜⬜	Range	🟨⬜⬜⬜⬜
Ammo Capacity	🟨🟨🟨⬜⬜	Projectile Speed	🟨🟨⬜⬜⬜
Projectile Size	🟨⬜⬜⬜⬜	Projectile Amount	⬜⬜⬜⬜⬜
Accuracy	🟨⬜⬜⬜⬜	Explosivity	⬜⬜⬜⬜⬜
Projectile Damage	🟨⬜⬜⬜⬜		

Tokens left: 0

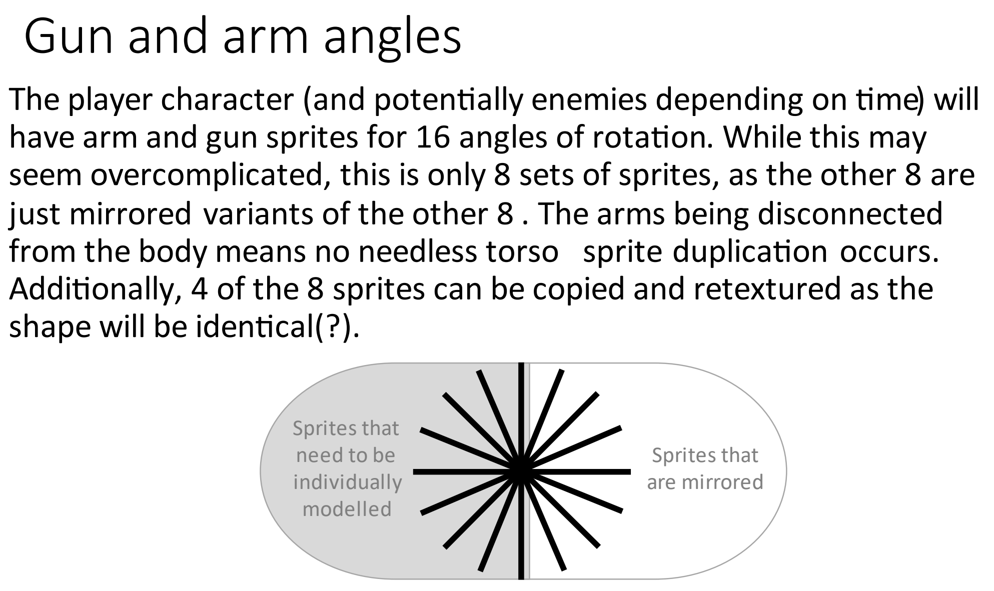
unlimited?

Weapon type:
Pistol

(a concept for a weapon upgrade system)

Figure 9. *Plans for a weapon upgrade system. (Game Design Document)*

This section outlines what a potential weapon upgrade system might have looked like, where each bar next to a statistic such as fire rate or ammo capacity indicates a 'level', with a yellow bar indicating that the level has been 'bought'. One yellow bar for the fire rate might indicate somewhat slow shooting gun, whereas a full five bars might indicate a rapidly firing weapon. Finally, I included plans for a system I found initially to be a little bit more complicated:



Gun and arm angles

The player character (and potentially enemies depending on time) will have arm and gun sprites for 16 angles of rotation. While this may seem overcomplicated, this is only 8 sets of sprites, as the other 8 are just mirrored variants of the other 8 . The arms being disconnected from the body means no needless torso sprite duplication occurs. Additionally, 4 of the 8 sprites can be copied and retextured as the shape will be identical(?).

Sprites that need to be individually modelled
Sprites that are mirrored

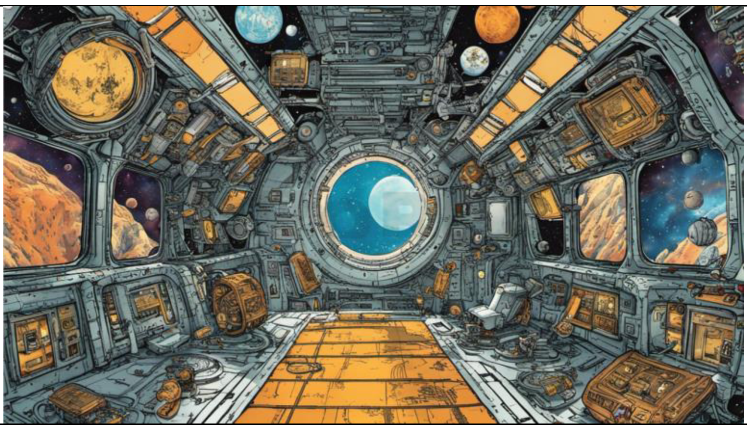
Figure 10. *A rough explanation on how I would go about setting up sprites for the player character's arms. (Game Design Document)*

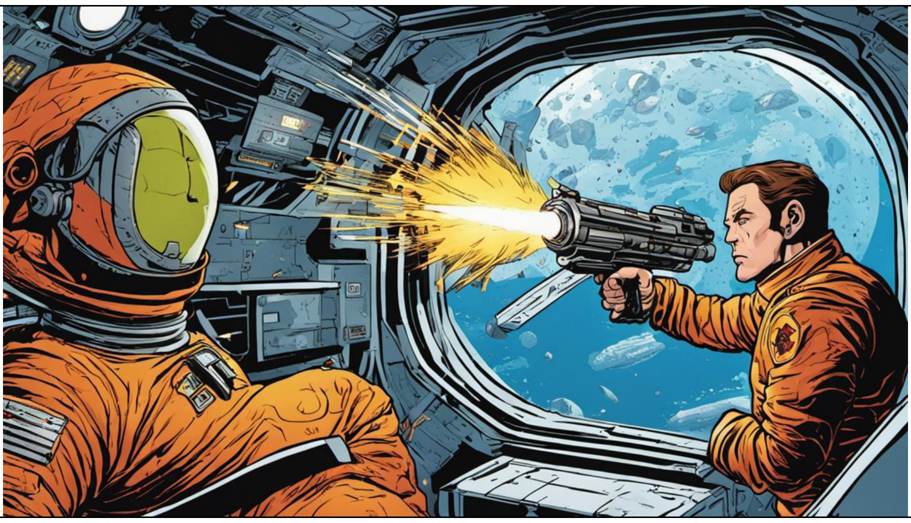
In the game, the player character's arms move independently of their body – as in, a player character's torso may be facing downwards, but they may be aiming their weapon to the east. As I found the arm rotation system to be quite complicated, I had to create a plan on how I would go about creating it. Plans such as these that outline key gameplay features are common in GDDs, as they help all members of a game development team to understand what the designer wants exactly. I talk more about the arm rotation system on page 39.

Storyboarding

Bond and Beale (2009) found that good storytelling was a point of 'high significance' when it comes to how video games are reviewed/received. They state that, while games aren't scorned much for bad storytelling if other features are received positively, good storytelling still makes a bigger impact on success. It is difficult to tell a story in a game I did not have a lot of time to work on, so I created a storyboard to outline what the story of my game is, including both lore and important events that the player is exposed to in the game. During the development of a game, it is common for a game's story and lore to change as new ideas may come about, or technical/artistic requirements demand changes.

I initially created the storyboard using Boords (2023), a website advertising a storyboarding creation service. Though only outlining the first hour of gameplay, the storyboard I created is useful to 'get the ball rolling' and helps set the themes and lore of the game's world. Boords offered a free service that allowed me to create twelve storyboard frames with A.I generated images (I was allowed to generate only five for free, although only three generated correctly for me) to go with them, though I was unfortunately unable to export them using their free service without upgrading to a premium plan. I had to instead copy and paste the storyboard content below, to help give some insight into the beginning of the game's story:


The Galactic Union faces a crisis as a breakaway faction surfaces: the Separatists. Branded by the Union as violent terrorists and criminals, the Separatists say they are unhappy with levels of Union bureaucracy and corruption, and their perceived ambivalence of less populous alien races wanting to join the union. The Separatists want a new union with open borders, and a more direct and fair democratic system. The Union says they are cautious and rigorous with checks of new species as they seek to keep peace in the Union. Battered by wars with other, alien empires in the past, the Union is distrustful, and especially cautious of spies and proxies of enemy empires.



Right before the events of the game, the Galactic Union's Starship Stalwart's decorated captain is killed in a firefight with Separatists on a busy capital space station. Weeks later, the mourning crew of the Starship Stalwart are forced to welcome aboard a new captain - the player - as animosity between the Separatists and the Union continue to heat up to astronomical levels.

The Stalwart's weapons expert, a wheelchair-using canine-esque humanoid called Lieutenant Kraul, is the first to meet the onboarding player. Through conversation, the player learns that Kraul expresses disinterest in the chain of command. The only reason they followed the Stalwart's previous captain was because they were a hero, not because they were the captain. He seems to hate the Separatists for killing such a hero. In his eyes, the player has some big boots to fill. The player is directed to commander Zora Sabon, the ship's second-in-command. She's very by-the-book, and elusive about her feelings, preferring to keep to formalities. She tells the player that the ship is ready to leave orbit at their command. She seems to agree with Union sentiments, but the player is left to wonder if she is merely following orders.

There are many missions and bounties set by the Galactic Union Council, from pest-control to diplomatic excursions. The player is given free rein on where the Stalwart heads, as they begin their first randomly generated mission. As the ship leaves orbit, the player is given time to prepare for their mission. Should the player explore the ship more, they overhear a conversation between the ship's counsellor, Chief Aurora, and lead scientist Lt. Rohan, who are in the laboratory. It seems like Rohan is worried about his childhood friends, who he knows to be Separatist leaning. Aurora is very understanding, telling Rohan that he could consider taking shore leave to visit them. The player is given the option to join their conversation, and is given the option to offer Rohan support, or reprimand Rohan for associating with Separatists. He accepts the reprimand, but both he and Aurora will resent the captain for it.

After some time in warp, the player is transported to its location. During the excursion, the player is exposed to chatter on comms between his crewmates. Kraul and Rohan seem to butt heads a lot, but the moment anything technological or scientific is brought up, they seem to forget about their animosity and work together like clockwork (the underlying theme here being that, despite their differing ideologies, they have a lot in common, and are really not all that dissimilar to each other).

Upon completion of their mission, the player is transported back to the ship with newfound platinum, intel, weapons, etc. Kraul offers their services to the player, allowing them to upgrade and modify their weapons and equipment. Depending on how the player spoke to Rohan earlier on, Rohan will either seek the player out to talk to them, and offer their own

services, or the player will have to find them in their lab and ask about his services. The player is encouraged to choose their next mission. The game progresses like this until a few missions in, where the Stalwart receives a distress signal on the borders of Separatist Space. A civilian ship claims they have been boarded by Separatists. However, Commander Sabon notes that scans indicate only one ship in the area.
This starts the first 'story' mission, which is not randomly generated like previous ones. The player beams aboard a ship of unknown origin, and it is in chaos. A group of heavily armed humans are holed up in the mess hall of the ship, having barricaded the doors. They claim the Separatists appeared out of nowhere and have already killed multiple members of the crew. However, the player receives a communique from Chief Aurora, who senses that the Separatists on the other side of the barricade are more frightened than they are angry and urges caution from the player. The player may opt to help the civilians ward off their claimed invaders, or can try and talk to the Separatists, who claim that actually, the civilians are slavers, and that they initiated a 'prison riot' to try and escape. Lt. Kraul will argue that they are lying, but the decision of who to help falls to the player.

Godot and GDScript

Upon completion of my GDD, with all necessary planning completed, I began work on my game. I started a new project on Godot and was given some initial options – whether I wanted the root node (the parent object that would constitute the main scene, or the object within which the majority of the game's nodes and scripts exist) to be a 2D scene, a 3D scene, a User Interface, or something else; what programming language I wanted the game to run on; and what renderer I wished to use. For the first setting, I chose 2D, as I planned to create a 2D top-down game, and creation of 2D assets to use was more realistic and within my technical know-how than 3D ones. For the second setting, I chose to use GDScript, Godot's own language, though also available to use were C#, and also offered are C and C++ which have 'experimental' support. The reasoning behind this was that GDScript contains elements of Lua and Python, two languages I am very familiar with, while also containing elements from the C, C++, and C# due to the fact it is object-oriented and imperative (Godot 2023a), which offered opportunity for me to practice programming in a C-like fashion, while allowing me to use features from Python and Lua that are familiar to me, like whitespace. GDScript is also gradually typed, meaning I can define types (statically type) when I am programming, but I have the option of omitting types, which makes the compiler regard the object being defined as dynamic, allowing it to define a type automatically at runtime. Finally, for the third setting, things get more complex; I was given a choice of 'forward+' rendering, 'mobile' rendering, or 'compatibility' rendering, the key difference between the three being their levels of compatibility with different platforms (Godot 2023b). Forward+ rendering supports desktop platforms only, but offers the most in terms of power, capable of rendering large, complex scenes, and offers the most features in terms of 3D graphics. Mobile is a slightly less powerful version, offering mobile support as well as desktop, but is faster at rendering simple scenes. Compatibility offers the most in terms of platform support, supporting desktop, mobile, and browser support, and is the fastest at rendering smaller scenes, and is 'intended for low-end/older devices'. For my game, I decided on mobile rendering, seeing as I did not expect my game to be overly complex, and the extra platform support is nice, but I didn't want to be . In the case of my game becoming much more complex than I expected, with increasingly larger scenes requiring more efficient processing by the engine, then I also had the option to migrate to a different rendering system.

Now the project had been created and the project directory had been set up, I had access to the development environment, which offered an impressive suite of tools to help me develop my game, from a large collection of 2D and 3D 'nodes' of differing types, which are essentially the building blocks of the game, to an in-app integrated development environment (IDE). My plan was to develop the game in Godot, importing sprites I would create in Krita. All my code was to be written using GDScript, in Godot's IDE.

Krita, Spritework and Pixel Art

The artwork required for my project was completed using Krita v4.4.0 (2020). Newer versions of Krita are available, however v4.4.0 is the version I have weeks of experience using, and I did not want to waste too much time learning a new art suite as well as a game development studio.

Krita is an open-source art creation environment that allows for users to utilise and customize numerous editing tools, brushes, and canvases to create art. Out of the large collection of art creation suites available for free on the internet, Krita was perfect for my project due to the large amount of experience I have with the software, and also because I could reduce the canvas size to levels where individual pixels are visible and can be finely manipulated, giving the art the retro feel I desired.

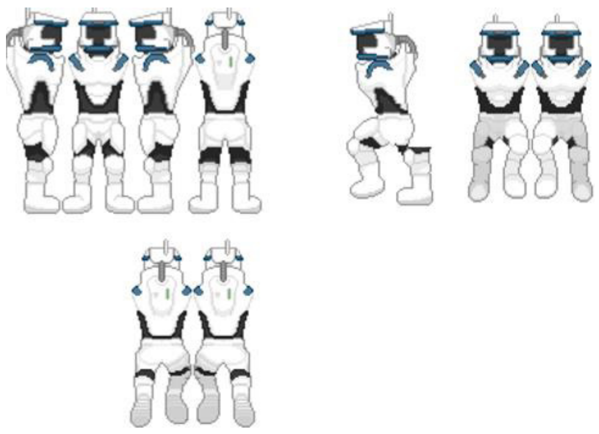


Figure 11. The beginnings of the player sprite sheet.

From a small canvas size, I created a guide – essentially a grid, stored on a layer that would be hidden when the art was ready to export – to ensure the sizes of my sprites remained within multiples of 32 by 32 pixels. I did this as my initial group of sprites, pictured in figure 11, were created using guides of 64 by 64 pixels, which turned out to be too big; after two days of work designing the player character's spritesheet, I felt as though each sprite was taking too long, There is a balance to the resolution of pixel art for games; in theory, the lower the resolution, the better, as lower resolutions become stylised (see figure 12), and worktime on a piece of art is reduced, enabling the artist to produce more sprites per hour. On the other hand, lower resolutions, particularly below 16 by 16,

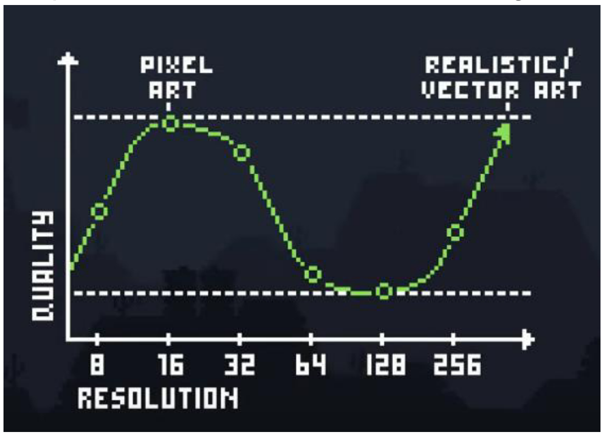


Figure 11. A graph showing the relationship between resolution and deemed artistic quality. (Pixel Overload 2020)

become difficult to work with due a lack of available detail, so sprites become vague and overly simplified. With larger resolutions, according to Pixel Overload (2020), sprites go through an ‘uncanny valley’ where they aren’t a low enough resolution to have a retro feel, but they’re not a high enough resolution to look realistic or to achieve any particular style. Additionally, worktime increases as the resolution increases. For me, the sweet spot was using a grid of 32 by 32 pixels; it was large enough to capture the details I needed, without being so detailed as to lose its retro feel, and without requiring a large amount of time to create.

Shading on more complicated sprites was achieved by creating a new layer to work on, setting it to 30% opacity, selecting the bounds of the image I am shading (effectively selecting the precise area of the sprite and ensuring I couldn’t shade ‘over the lines’) and then blocking out areas I wanted to be shaded with black. This then creates a shading effect that was quick to achieve.

Static sprites, such as unmoving objects like rocks or console segments (see figure 13), only need a single sprite to represent them. However, animated sprites require multiple individual sprites: each frame of the animation. In the case of the player sprite, it needed walk and run animations and the arms needed different sprites depending on which direction they are facing.

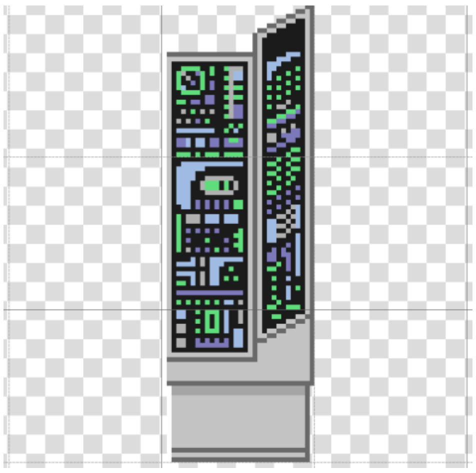


Figure 12. A console. The middle square can be removed/duplicated to create shorter or longer consoles.

Creating UI for the game took some careful thought in terms of its design. I decided to base its design on Star Trek: The Next Generation’s (Roddenberry 1987) ‘LCARS’ system, which is how information is displayed on computer screens in that universe.

Exporting the artwork was where things got a little bit complicated: I needed to resize the canvas to match the size of the sprite I had just created, turn off any guides I had used, and then save the image as a .png (as .png images retain transparency; saving as for example a .jpeg will save any transparent pixels as being white, meaning sprites will render with a white square/rectangle background in the game).

Once arms, head, and torso sprites were completed for the player character, all the player character’s parts were imported to Godot. In Godot, I had to set the ‘import filter’ to ‘closed’, otherwise the sprites would import blurry. In figure 14, you can see two versions of the player character – an idle, front-facing version, and a sideways, mid-sprint version.

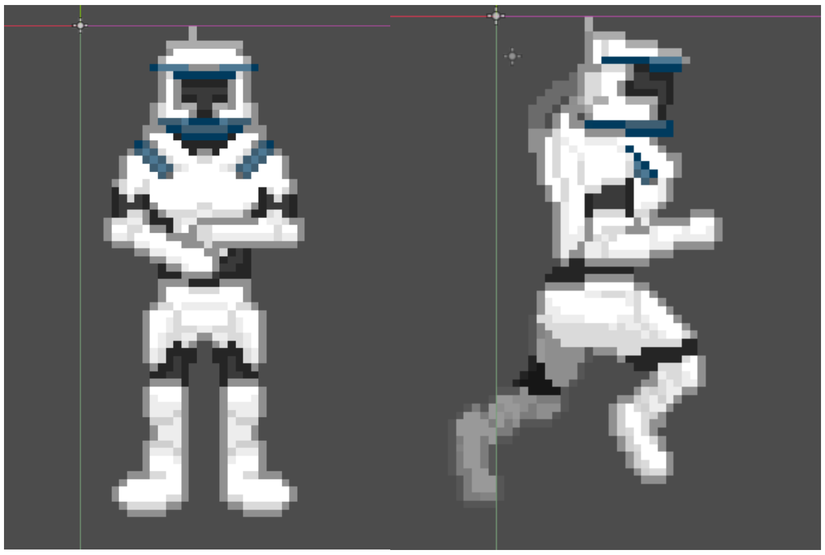


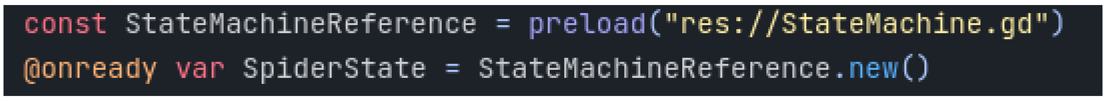
Figure 13. Two versions of the player sprite.

Scenes, Nodes, and the Hierarchy

In creating the game, it was important to understand Godot's way of organising projects and the building blocks used to create said projects. I explain how it works in this section before examining how my own project was organised.

A node is described by Godot as a 'building block'; a 'base class for all scene objects'; 'an ingredient in a recipe'. Nodes can represent anything from a sprite to a sound; a 'collision shape' to a UI element; a button to a camera position. Nodes can be given names, contain editable properties unique to the node type, and are sometimes tangible in the scene editor, in that they can be dragged around, rotated, and resized. Nodes can also be organised in a hierarchical manner by adding them, as a child, to another node, and a node can have as many children as it needs. This creates a node tree, and is identical to GameObject hierarchies in Unity, and instance hierarchies in Roblox Studio, in that nodes can be the parent of many child nodes but may not have more than one parent.

Node trees are always children to a scene, which acts as a grouping mechanism, holding a large tree together into one object. A project can have multiple scenes with different trees within them, and scripts assigned to nodes in one scene can reference other scenes or nodes within other scenes by first using the 'preload' method, defining the path to the node, scene, or script, then by making references to it as normal using the name of the variable the preloaded object was assigned to.



```
const StateMachineReference = preload("res://StateMachine.gd")
@onready var SpiderState = StateMachineReference.new()
```

Figure 14. *A class, 'StateMachine', I defined in another script is being preloaded into this script.*

In my own game, I used nodes extensively (as any game in Godot would). Looking at the spider enemy I created, it alone contained 15 separate nodes, all performing different functions to make the enemy work. Figure 16 displays its hierarchy, from the parent node 'SpiderEnemy' to its various collision shapes (I talk more about collision shapes and what they are on page 41). For the enemy to function, it required a 'CharacterBody2D' for scripts to be able to move the object, as CharacterBody2D nodes contain the required properties and a library of methods relating to movement and physics. I named this node 'SpiderKinematic'. This node then contained the

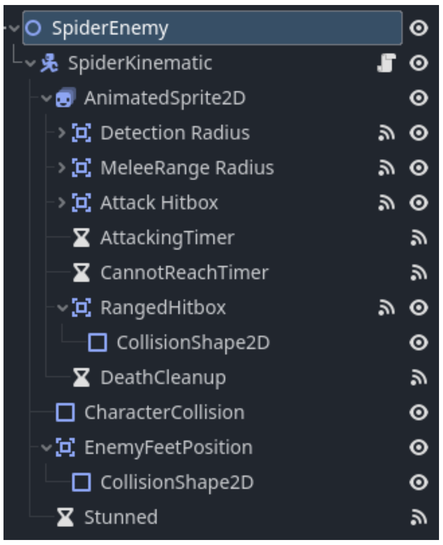


Figure 15. The Spider enemy's node hierarchy.

‘AnimatedSprite2D’, which is the animated sprite visually representing the spider. AnimatedSprite2Ds contain properties and methods used for handling sprite animations, though it has no physics, hence why it is a child of a node that does handle physics. The two nodes therefore work in tandem as any node contained within some kind of kinematic sprite, such as a CharacterBody2D, gets moved along with it; the only reason the CharacterBody2D is able to visibly move is because it has an AnimatedSprite2D to represent it; without one, the node would be invisible. Contained within the AnimatedSprite2D is a collection of collision shapes (again explained on page 41), and also some timers. The timers are used in scripting, as they contain properties and methods that allow the node to detect when a certain amount of time has passed since it was activated. I used timers to give ‘cooldowns’ on certain actions the spider might take. For example, when the spider reached 0 hit points from being shot at by the player, the ‘DeathCleanup’ timer is activated. It then signals its completion a couple of seconds later, so the Spider object can be deleted. It does this to give the AnimatedSprite2D it is the child of enough time to complete its death animation, and also enough time for in-game currency to drop as a reward for slaying the spider. The ‘SpiderEnemy’ sprite can be collapsed in the hierarchy window, as we may not need to be able to see every single node contained within and may only need the node that represents the entire object.

With my own game, I had planned to include scenes for different game states: a scene for the main menu, which would include all the UI and buttons required to start or load the game; a scene for the ‘shipside’ state of the game, wherein the player is in the ‘hub’ and can talk to their fellow crew before setting off on missions; a scene for the ‘planetside’ state of the game, which would contain all the necessary sprites and nodes required to randomly generate a level each time the scene switches to it; and a ‘storage’ scene, wherein enemies, props and general objects are stored, until a script from one of the other scenes duplicates any of its items and places them into their own scene. In practice, however, due to the short timespan I had to develop the product, I only had time to develop on two scenes: the ‘main’ scene, which included a map, the player object, enemies, and props already loaded into the game, and also a ‘storage’ scene, where any objects I didn’t need in the main scene were kept. This included the in-game currency, as while it didn’t need to be loaded into the main scene, it would drop upon an enemy dying, and was so duplicated and moved from the storage scene to the main scene. Similarly, the laser object that is shot from the player’s phaser is kept in the storage scene, and then duplicated and moved to the main scene when the phaser is fired.

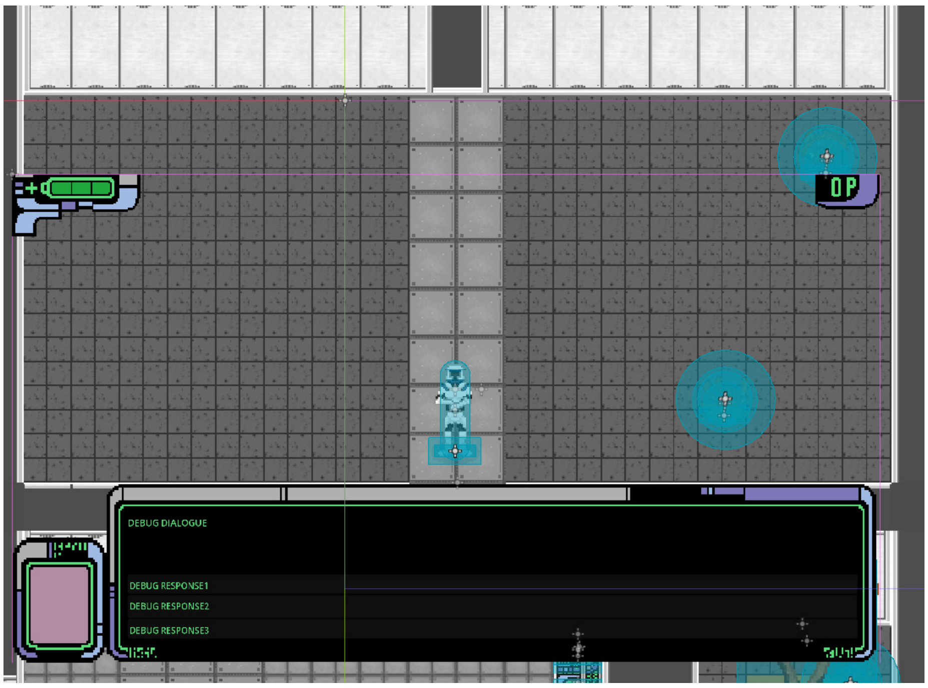


Figure 16. *A view of the scene editor, set to the 'base' scene. All UI elements and the player character are stored here.*

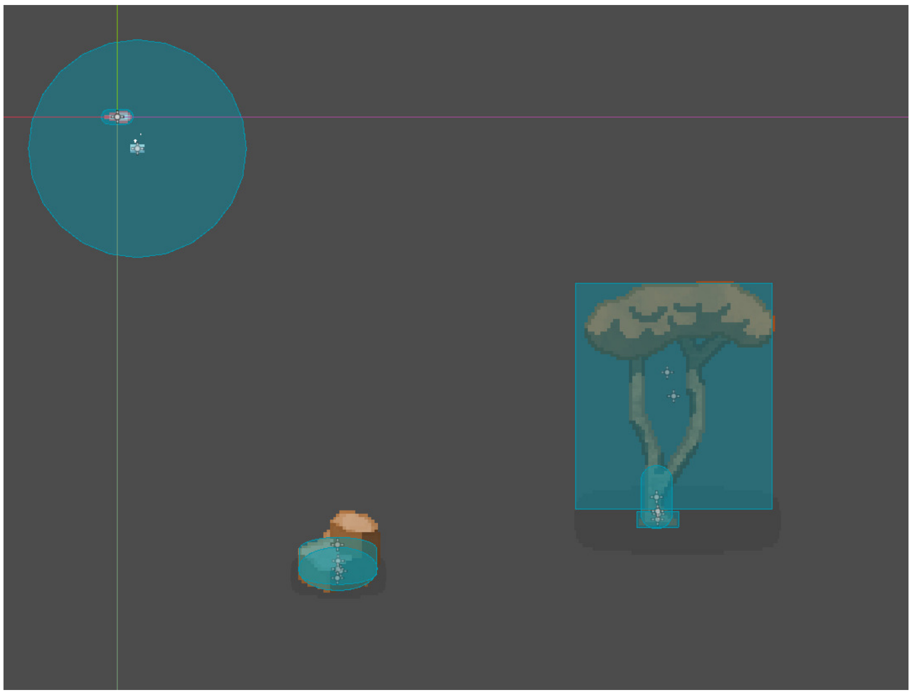


Figure 17. Another view of the scene editor, this time set to the storage scene, wherein props, the player's phaser laser, and currency is stored.

Creating Animated Sprites

I found creating animated sprites to be easy in Godot, as its keyframe-based SpriteFrames editor was intuitive and simple to use. In more advanced projects, its simplicity could become a problem – but for the small game I was creating, it was perfectly adequate, albeit with a few minor issues. I was able to create 'AnimatedSprite2D' nodes in my project, which are hierarchical objects that store 'SpriteFrames', a blank library of configurable animations. I had to ensure the AnimatedSprite2D nodes were nested in a 'KinematicBody2D' (AnimatedSprite2D nodes cannot move on their own; they need to be a child of a node that can move), attached to them a 'SpriteFrames' (essentially a blank canvas for a library of animations). With the node set up, to create an animation that would appear in the game, it wasn't any more complicated than giving the animation an appropriate name, and then inserting each frame individually. The length at which each frame would stay onscreen was adjustable, alongside the fps (frames-per-second) of the whole animation, and also whether the animation would repeat or not. Sprites could change which animation they had playing through scripting.

The only issues I had with the SpriteFrames editor was down to its minimalistic nature. There was no way to edit frames within Godot; if a frame was slightly off-centre or the wrong size, Godot seemed to offer no way to resize or move the frame's position, which meant I had to make the adjustments on Krita, and go through the process of exporting, then importing the new frame, before adding it back to the animation. Godot also offered support for importing entire sprite sheets (one image file containing all the frames needed), but I struggled to get this feature to work due to how unforgiving and rigid the 'capture grid' was – e.g if a row of sprites is one pixel too far down, the row of grid cells meant to capture them would be one pixel too far up. A fix for this would be to simply allow the user to 'drag select' over each individual sprite – this would also allow the user to manipulate the amount of whitespace around each sprite, potentially solving issues with centring from within Godot too.

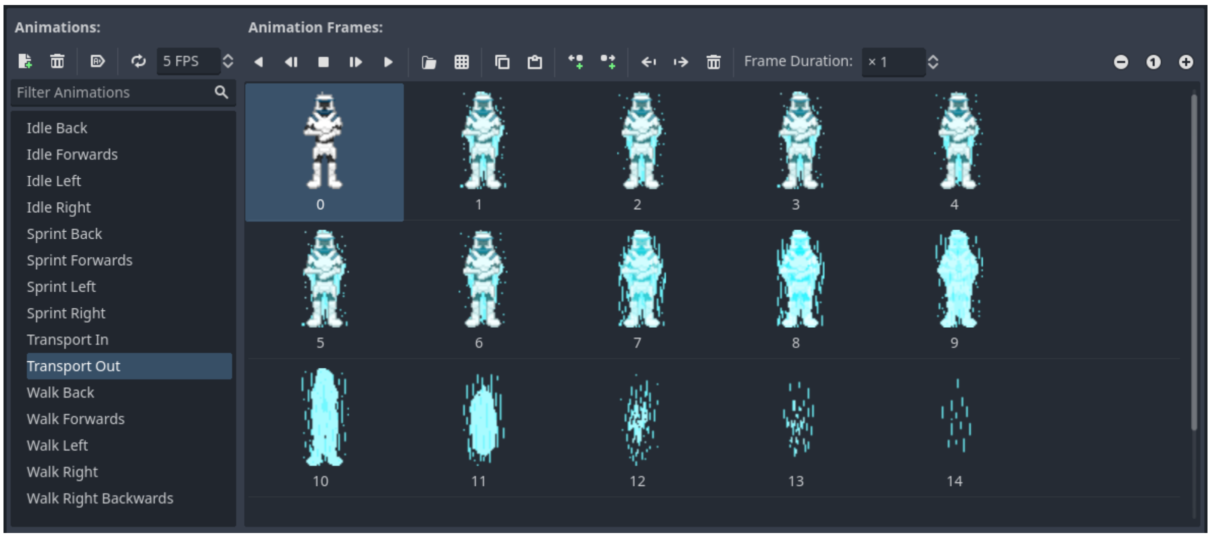


Figure 19. The AnimationFrames for the player character's teleport animation.

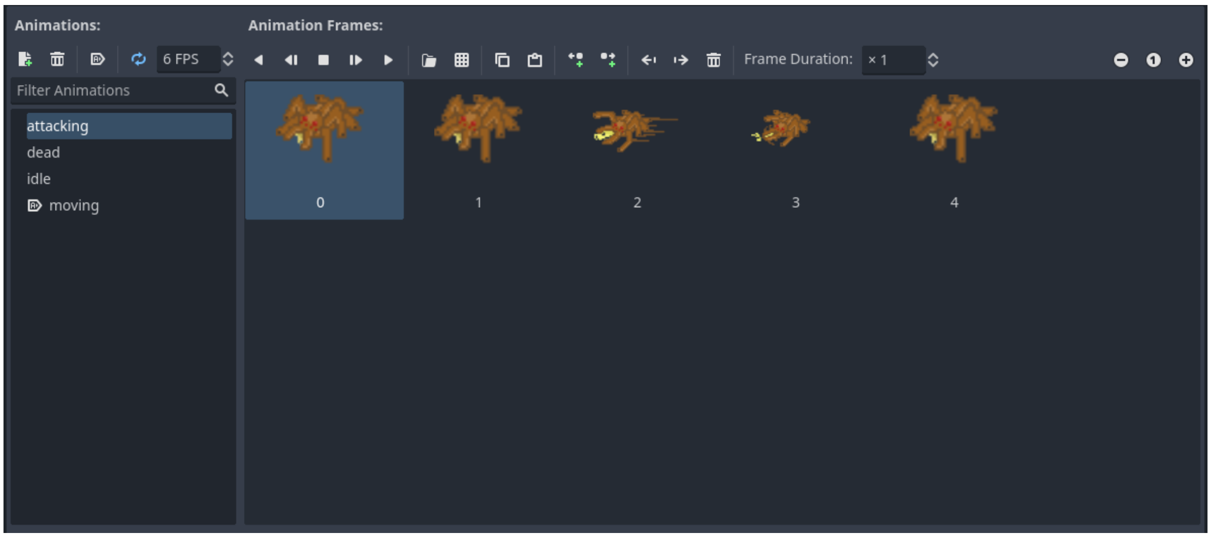


Figure 20. The AnimationFrames for an enemy spider's attack.

Tilemaps, Environment Design, and Level Generation

Lots of aspects go into creating a compelling world – characters, lore and sounds for example – but perhaps most import for immersion's sake is the physical environment in which the game takes place in. Environment design and aesthetics is important in video game development as it 'defines most aspects of the context in which the game takes place', and it renders 'an atmosphere capable of drawing and maintaining players' attention on an emotive basis, making them feel part of an entailing virtual world' (Fabricatore 2007). Generally, the environment and world created in a game (or any piece of media) should match the genre, although science-fiction exists in a unique situation where a character can be in any environment or time period and there will always can always be an explanation for it – take for example season 6, episode 13 of Star Trek: Deep Space Nine (Roddenberry 1992), in which the entirety of the episode takes place in 1950s US. Typical science-fiction environments, such as spaceships, space-stations and alien planets are highly appropriate for sci-fi video games, as such environments allow for interesting gameplay, such as encountering/fighting alien fauna, exploring a space-station, and talking to diverse aliens with interesting, if not out-of-this-world, stories. For my own game, I hence created tilemaps and sprites that could be used to develop these environments: high-tech, panelled walls and floors for a spaceship or space station interior; and strangely coloured grass, rocks, and trees for an arid alien planet (see figures 21 and 22 respectively).

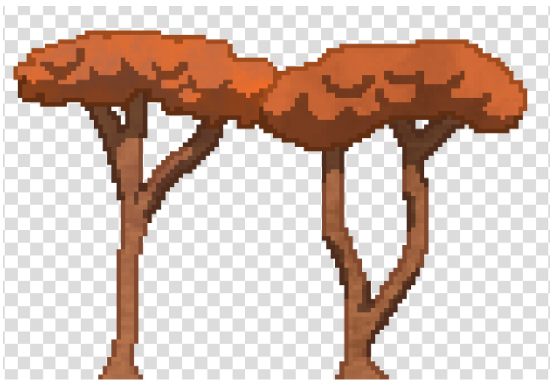


Figure 21. A pair of alien tree sprites.

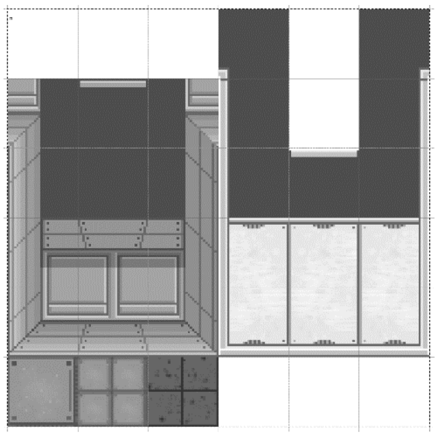
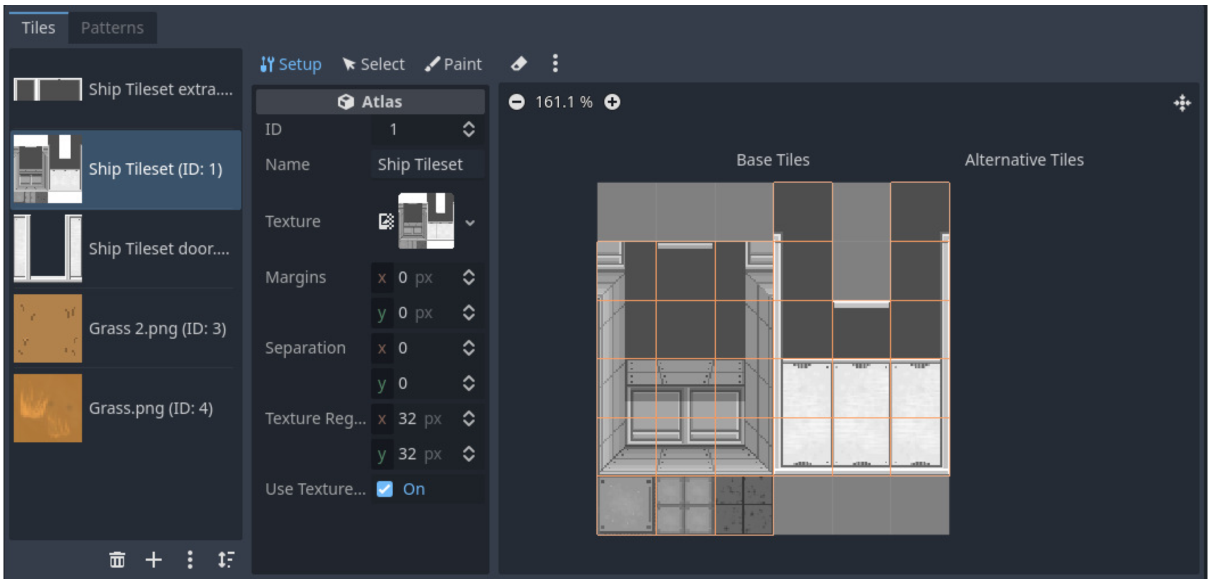


Figure 22. A tileset for a spaceship.

A tilemap is defined by Mozilla (2023) as a way to build 'the game world or level map out of small, regular-shaped images called tiles.' They state that using tilemaps, rather than creating 'big image files containing entire level maps', is a great way to improve game performance and memory usage. The idea is that the game world can be 'painted' within the game editor using these tiles as your pallet. A tileset (also known as a tile atlas) must first be created, which is an image file containing all the tiles that comprises the tilemap. I used Krita to create my tilesets, using a grid of tiles with a size of 32 by 32 pixels, that included different floor and wall tiles. It was important for tiles of a similar type to ‘flow’ – that is, when placed together in the editor, it should be difficult to see where one tile ends and another begins, as they should look connected. Walls needed corner tiles and ‘top’ tiles so as to make rooms of different shapes and sizes. Once the tileset had been finished, it was time to import it to Godot studio.

Godot allows the user to import images to use as a tileset and then allows the user to resize the grid, define any empty tiles that are not used, and more. First, I needed to create a ‘tilemap’ node, which I placed into the main scene. I then had to define its tileset, which I did by selecting the tileset image I had imported to my game folder. I then set the texture region size to 32, which set the size of each tile to be 32 by 32 pixels (ensuring the tiles weren’t split into smaller tiles or formatted incorrectly). Margins and separation were set to 0 pixels as I had created the tileset to have no empty space between tiles, so it was easier for me to see how each tile linked together. Setting which tiles I did not want to be used was as simple as right clicking the tile, and then selecting ‘delete’. This ensured that I didn’t accidentally start painting using an empty tile, as while empty tiles render as invisible and hence have no aesthetic impact on the game environment, I would still be writing unnecessary data to the game files, which would increase file size, and effect game performance, which is undesirable.



Ship Tileset extra....
Ship Tileset (ID: 1)
Ship Tileset door....
Grass 2.png (ID: 3)
Grass.png (ID: 4)

ID 1
Name Ship Tileset
Texture
Margins x 0 px
y 0 px
Separation x 0
y 0
Texture Reg... x 32 px
y 32 px
Use Texture... ✓ On

Figure 23. *The tileset editor.*

With the tilemap ready, actually using the tiles to paint a world was made to be very user-friendly in Godot, as it was as simple as selecting which tile I wanted to paint with (see figure 23 to see how each usable tile has an orange outline), and then physically clicking on the scene view window to paint the selected tile onto the game world. See figure 24 for a simple corridor and alcove created using my tileset.

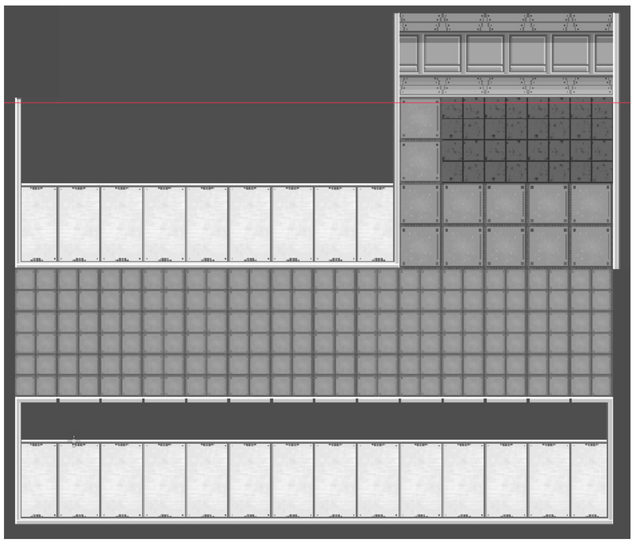


Figure 24. Simple, high-tech architecture, painted onto the scene, using Godot's tilemap feature.

A benefit to using tilemaps in Godot is that, similarly to the tiles stored within its atlas, the tilemap itself becomes reusable. This is extremely useful when creating roguelite games, as a core feature of roguelikes and roguelites is level randomisation (Harris 2021). For a level to generate, I needed premade rooms/areas to have been created and stored somewhere, as the game would pull rooms at random from storage and connect them together. As Godot handles tilemaps as individual nodes, the user can create a room inside of one tilemap, then store the tilemap in another scene, ready to be used in map generation. Multiple scenes should exist to account for differing geometries: rooms with one exit should be stored separately from rooms with 3 exits, for example. In the case of my own game, I did exactly that: I created some simple ‘rooms’ with differing elements such as room geometry, decoration, and door locations, all stored on unique tilemap nodes, and ready to be stitched together upon generation of a level. Additional elements such as enemy objects/spawn points, props, and in-game resources can also be stored alongside the tilemap.

The programming behind my level generation was unfortunately something I did not have time to completely implement, but I did create pseudocode and plans for this feature to explain how it would work. To avoid room for errors in level generation, I planned for map layouts to be predefined as a collection of entries (rooms) in a dictionary (the overall map).

Each entry would follow the formula shown in figure 25**,** assuming a map layout of 25 potential rooms spaces, in a 5x5 grid:

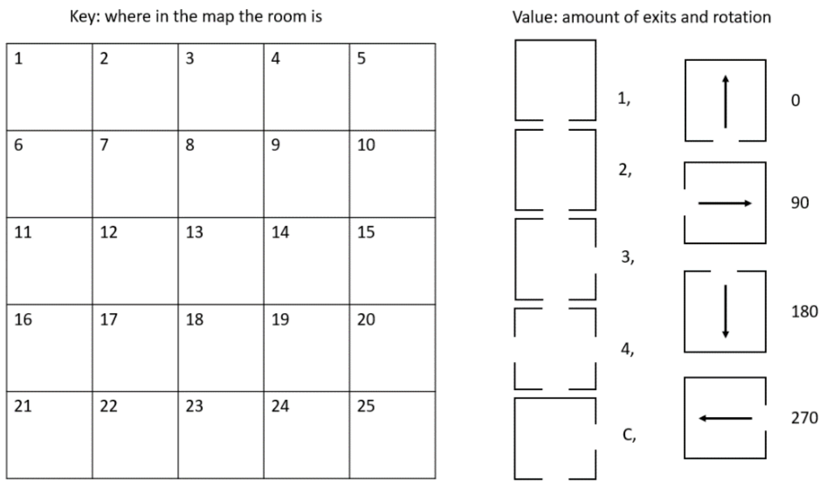

Key: where in the map the room is
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
Value: amount of exits and rotation
1,
2,
3,
4,
C,
0
90
180
270

Figure 25. *A visualisation of key:value pairings in a given map layout dictionary*

So, for example, an entry of '25: [1, 90]' would indicate the creation of a room in the bottom right corner of the map, with one exit, and rotated 90 degrees clockwise; an entry of '12: [C, 0]' would indicate the creation of a room one position to the left of the centre of the map, with a corner configuration, unrotated. Each dictionary would need as many entries as there are rooms in the map; any missing entries indicate a room shouldn't be generated. Figure 26 shows a possible configuration, using the dictionary of { 3: [1, 270], 4: [3, 90], 5: [1, 90], 9: [2, 0], 12: [C, 0], 13: [2, 90], 14: [3, 180], 17: [2, 0], 19: [C, 270], 20: [1, 90], 21: [1, 270], 22: [C, 180] }:

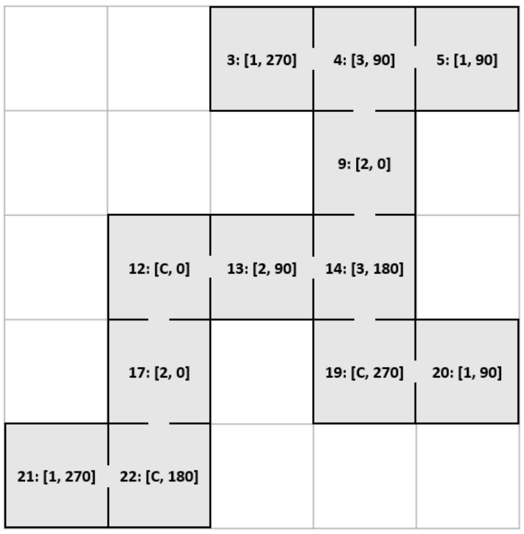


Figure 26. *A visualisation of the contents of a map layout dictionary*

With a collection of dictionaries like the above, all of different map layouts, stored in a large array, the game can randomly select one of them when generating a map, and then when generating each individual room, can pull rooms from the appropriate section: if the layout lists a room at position 14 with 3 doors, it will pull a room at random from the 3 door room storage scene, rotate it as required, and place it at the position 14's coordinates. The pseudocode is as follows:

```
import room1Storage, room2Storage, room3Storage, room4Storage, roomCStorage
var mapArr = [ (large array containing all map dictionaries) ]
var keyCoords { (dictionary with keys from 1-25 and values of their exact x
and y coordinates) }
func on_map_generation():
      var num1 = randomnumber(0, 25)
      var layout = mapArr.get(num)
      var room = null
      func generateroom(storage, num2, entry):
            var newRoom = storage[num2].duplicate()
            newRoom.set_parent(Base)
            newRoom.set_coordinates(keyCoords[entry])
      for entry in layout:
            match entry(first value):
                  1:
                        num2 = randomnumber(0, room1Storage.length)
                        generateroom(room1Storage, num2, entry)
                  2:
                        num2 = randomnumber(0, room2Storage.length)
                        generateroom(room2Storage, num2, entry)
                  3:
                        num2 = randomnumber(0, room3Storage.length)
                        generateroom(room3Storage, num2, entry)
                  4:
                        num2 = randomnumber(0, room4Storage.length)
                        generateroom(room4Storage, num2, entry)
                  C:
                        num2 = randomnumber(0, roomCStorage.length)
                        generateroom(roomCStorage, num2, entry)
```

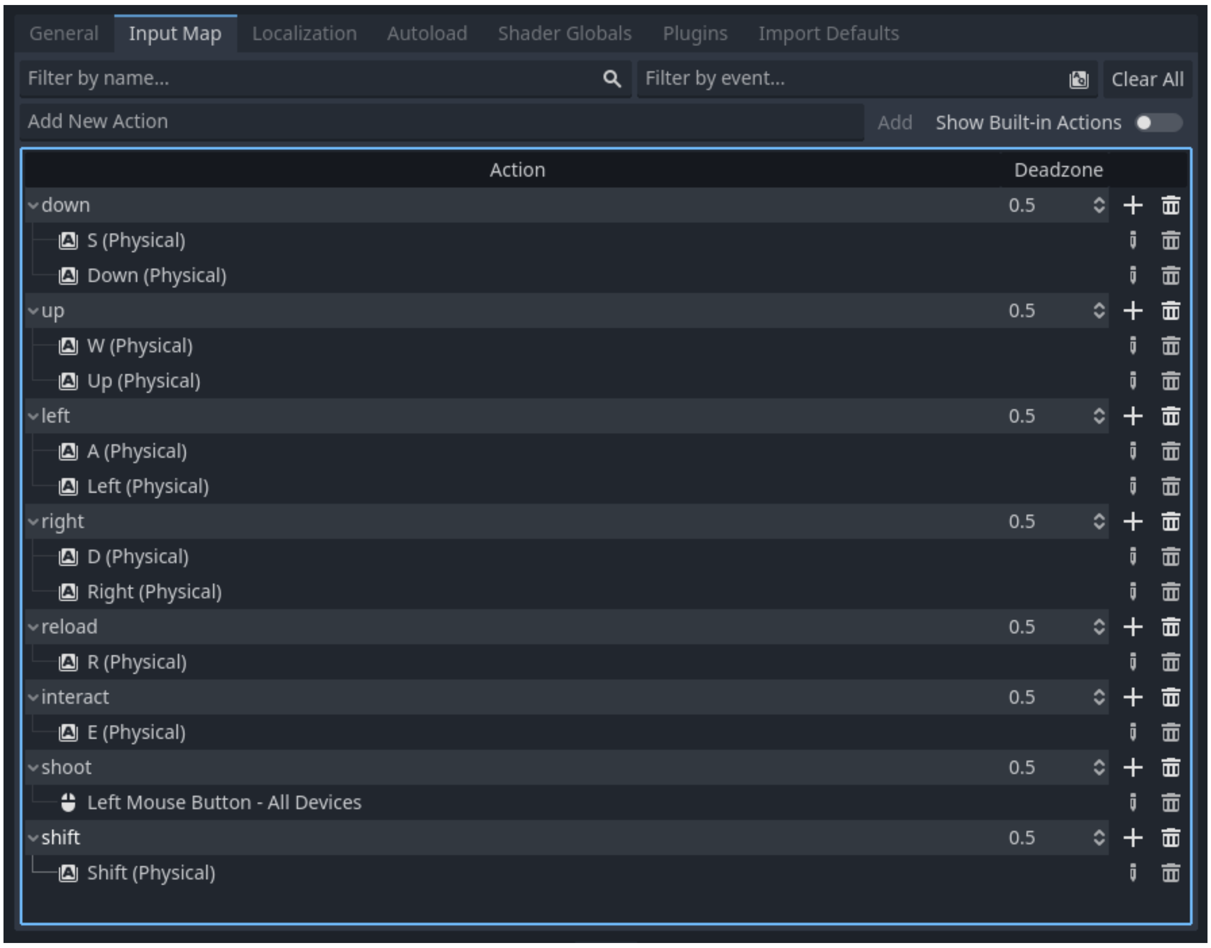
For additional functionality, boss and spawn rooms may also be stored using special values to indicate their unique status.

Control and Keymapping

In the vast majority of video games, there needs to be some way the player can manipulate the game using a peripheral like a controller or a mouse and/or keyboard. The means to create locomotion of a playable character in a video game vary from game to game, but generally, in the case of desktop gaming, players move and manipulate their character using the arrow keys on their keyboard, or alternatively the WASD keys; other keys may come into play to cause more effects, such as using the space bar to jump, or pressing ‘E’ to interact with in-game objects.

In the case of my game, I planned for the players to use the arrow or WSAD keys to move the character; use the mouse to manipulate where the character is aiming their weapon; ‘E’ to interact with objects; and clicking to fire their weapon or interact with in-game menus such as dialogue.

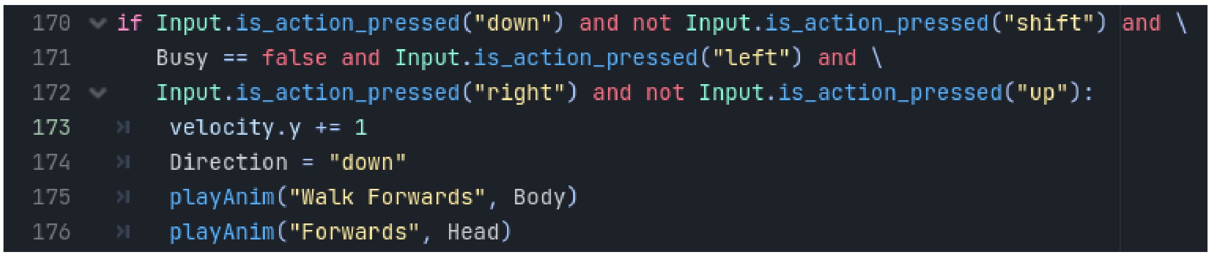
Programming which key presses send out what signal (keymapping) was simple to set up in Godot. The editor allowed me to set up named key bindings that, when pressed, sent out a signal that scripts could pick up.



Action	Deadzone
down	0.5
S (Physical)	
Down (Physical)	
up	0.5
W (Physical)	
Up (Physical)	
left	0.5
A (Physical)	
Left (Physical)	
right	0.5
D (Physical)	
Right (Physical)	
reload	0.5
R (Physical)	
interact	0.5
E (Physical)	
shoot	0.5
Left Mouse Button - All Devices	
shift	0.5
Shift (Physical)	

Figure 27. *The keymap (called Input Map in Godot) for my game, set up in Godot.*

Translating these button presses into in-game actions was a little more complicated to set up. For example, when making the player character walk around, the player is likely to be pressing more than one key at once at any given time, to make the character move diagonally; to interact with an object while walking by it; etc. Setting up which combinations of keys did what precisely resulted in a script of over 300 lines being created. Take for example one possible combination shown in figure 28:

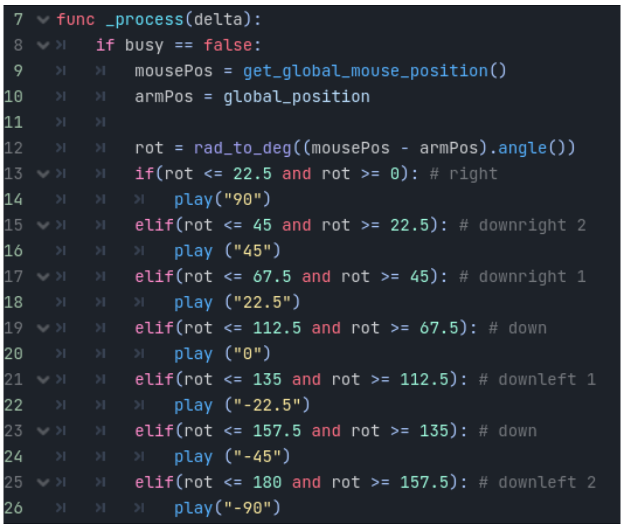


```
170  if Input.is_action_pressed("down") and not Input.is_action_pressed("shift") and \
171      Busy == false and Input.is_action_pressed("left") and \
172      Input.is_action_pressed("right") and not Input.is_action_pressed("up"):
173      velocity.y += 1
174      Direction = "down"
175      playAnim("Walk Forwards", Body)
176      playAnim("Forwards", Head)
```

Figure 28. Code for if a weird combination of keys is being pressed.

This if statement covers if the player is currently pressing down, left, and right on their keyboard; it translates to moving the player character down (as left and right movement cancel each other out). Line 173 defines that the character is moving with a velocity of 1 on the y-axis (downwards at a walking speed); 'Direction' on line 174 is a variable defined at the start of the script, to allow other scripts to know which direction the player character is moving; 'playAnim' on lines 175-6 define the animation each bodypart should be playing. Should this exact button combination be pressed again, while the player is also holding the shift key down, then the player will move in the same direction, but sprinting instead of walking, which increases the velocity, and changes the body animation to its sprinting forwards variant. Note on line 171 the variable 'Busy' is being checked to be false; this is as if the player character is 'busy', e.g teleporting, dying, or talking, they should not also be able to move.

Now, with the character able to walk and sprint in all directions based on user input, I had to set up the character's arms. The character should be able to move in a direction but aim their gun in another. So, I programmed it in such a way that the direction in which the character aims is based on the mouse's current position, as the character should aim at the mouse. This allows independent aiming and shooting, separate from where the character is walking. For this to look natural, I had to create a unique sprite for the character's arms based on the angle the mouse pointer was from the character's body, and a script that constantly detects the mouse pointer's position, updating the arms sprite accordingly.



```
7   func _process(delta):
8       if busy == false:
9           mousePos = get_global_mouse_position()
10          armPos = global_position
11
12          rot = rad_to_deg((mousePos - armPos).angle())
13          if(rot <= 22.5 and rot >= 0): # right
14              play("90")
15          elif(rot <= 45 and rot >= 22.5): # downright 2
16              play ("45")
17          elif(rot <= 67.5 and rot >= 45): # downright 1
18              play ("22.5")
19          elif(rot <= 112.5 and rot >= 67.5): # down
20              play ("0")
21          elif(rot <= 135 and rot >= 112.5): # downleft 1
22              play ("-22.5")
23          elif(rot <= 157.5 and rot >= 135): # down
24              play ("-45")
25          elif(rot <= 180 and rot >= 157.5): # downleft 2
26              play("-90")
```

Figure 29. *The first half of the code that covers arm rotation (the other half is more elif statements).*

In figure 29 above**,** I display how the arms' script detects which sprite the arms should be using; as the mouse position (mousePos) changes, its position from the body (armPos) is converted into an angle (rot) on line 12 by taking the armPos away from mousePos, taking the resulting angle in radians and converting it to degrees (rad_to_deg()). Then, in an elif chain beginning on line 13, the code works out within what range of degrees the angle is in and updates the arm sprite accordingly (e.g play("90")). Note that 'play' is the same function as 'playAnim' used in figure 28. Also note that the script checks if the player character is 'busy' on line 8; this ensures the character is not aiming their phaser while dying, teleporting, etc.

As the arms rotate in 'jumps' of 22.5 degrees, to ensure aiming the phaser looks precise, the phaser sprite renders separately from the arms, and rather than having multiple sprites at different angles like the arms, the phaser physically rotates in the game to the exact angle it needs to be at, ensuring that the phaser 'nozzle' is always pointing in the direction of the mouse; this ensures there are no weird visual artefacts, such as the phaser shooting lasers askew from its nozzle.

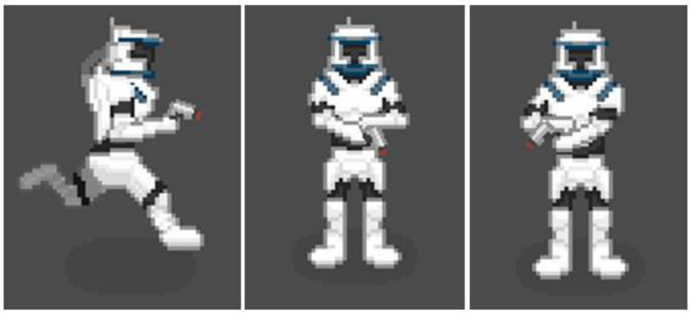


Figure 30. From left to right: Sprinting and aiming east; standing and aiming south, slightly east; standing and aiming south-west.

Collisions and Signals

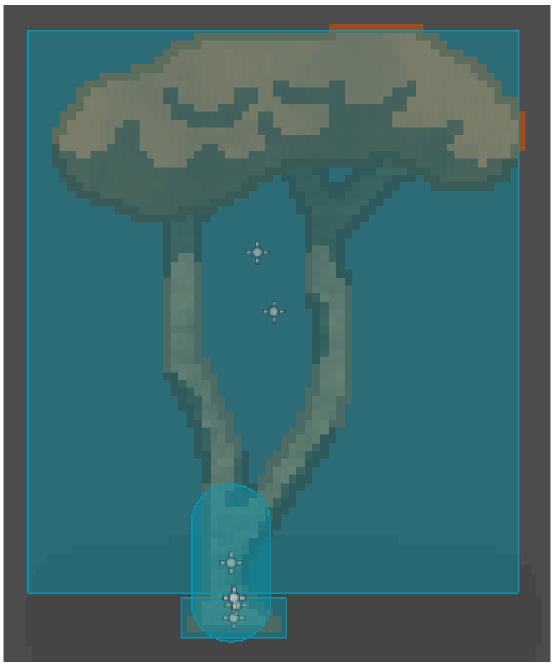


Figure 31. A tree sprite and its three collision shapes (the blue areas)

In order for the game to 'sense' and 'react' to certain events, such as the player character being hit by an enemy's attack, or the player character walking behind a tree, I implemented a system of 'collision shapes' and signals.

Collision shapes in Godot are primitive shapes, invisible from the player's perspective, that exist to 'sense' when it overlaps with another node; when an overlap occurs, it sends a signal to predetermined script(s), sending with it a reference to the exact node that overlapped it. They are also used in physics processing; a moving sprite might for example need to be blocked by something (e.g., the player character is walking into a tree), so the collision shape signals that it is being collided with and stops the offending sprite from moving into its space. For example, in my project, the 'tree' object (See figure 31 above) contains three collision shapes that perform different functions. First is a large rectangle area that senses if a sprite has moved into its space, checks if the sprite is the 'feet' area of a player or an enemy, and if it is, makes the tree sprite transparent; this is to allow the player to see what is happening behind the tree. The second, centred around the tree's trunk, is a collision shape that doesn't send out a signal itself, but exists for collision purposes; if a bullet detects it has overlapped with the trunk, the bullet may delete itself, to simulate it hitting the tree. The final collision shape, the rectangle at the very bottom of the tree, is a collision marker to stop the player character and other moving sprites from moving into the tree's space.

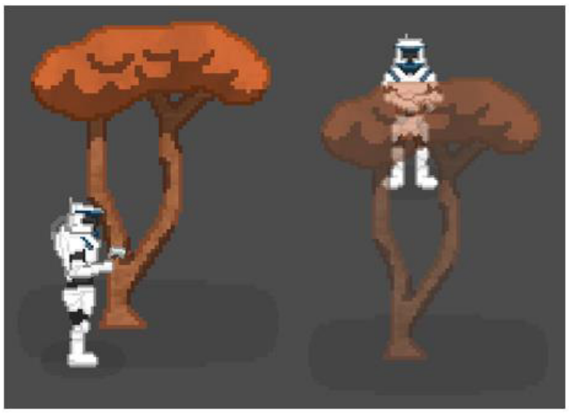


Figure 32. The player character standing in front of a tree, then behind it: note how the tree becomes transparent.

Godot displays signals under a ‘signals’ tab of a selected node, with said tab listing all available signals (including inactive ones), and gives the user the option to connect any given signal to a script (see figure 33 to the right). Upon connection, an empty function will appear in the connected script; the function will then get called if the signal is called. In the case of my own project, as an example, I had two connecting signals from the tree object; one for if something starts overlapping with the opacity-handling collision box (the largest collision shape in figure 31) (‘area_entered(area: Area2D)), and one for if a node *stops* overlapping. In figure 34 below, the resulting two functions appear in the tree sprite’s script: ‘_on_opacity_entered(area)’, and ‘_on_opacity_exited(area)’. The functions check if ‘area’ (the variable passing into the function, which is a reference to the node activated the related signal which called the function, is a feet position of either the player character or an enemy (lines 7 and 12). Then, the amount of nodes ‘behind’ the tree is modified (lines 8 and 13). If a node has gone behind the tree, the tree becomes transparent (line 9), but if the number of things behind the tree is 0 (line 14), the tree becomes opaque again (line 15).

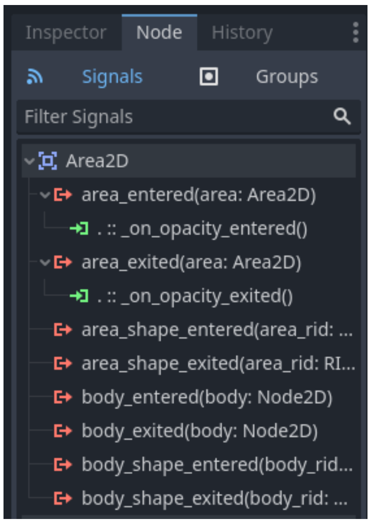
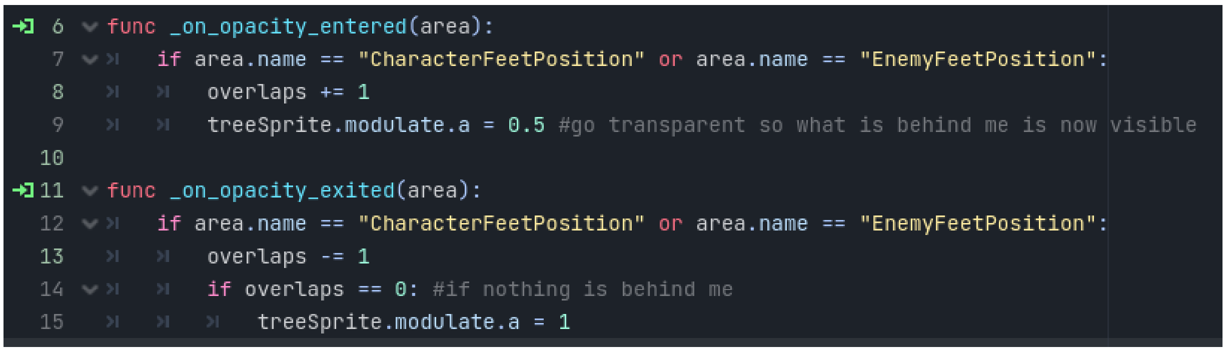

Inspector
Node
History
Signals
Groups
Filter Signals
Area2D
area_entered(area: Area2D)
. :: _on_opacity_entered()
area_exited(area: Area2D)
. :: _on_opacity_exited()
area_shape_entered(area_rid: ...
area_shape_exited(area_rid: RI...
body_entered(body: Node2D)
body_exited(body: Node2D)
body_shape_entered(body_rid...
body_shape_exited(body_rid: ...

Figure 33. The signals tab of a node. Note of the available signals, two have children, as those are signals I have connected to a script.



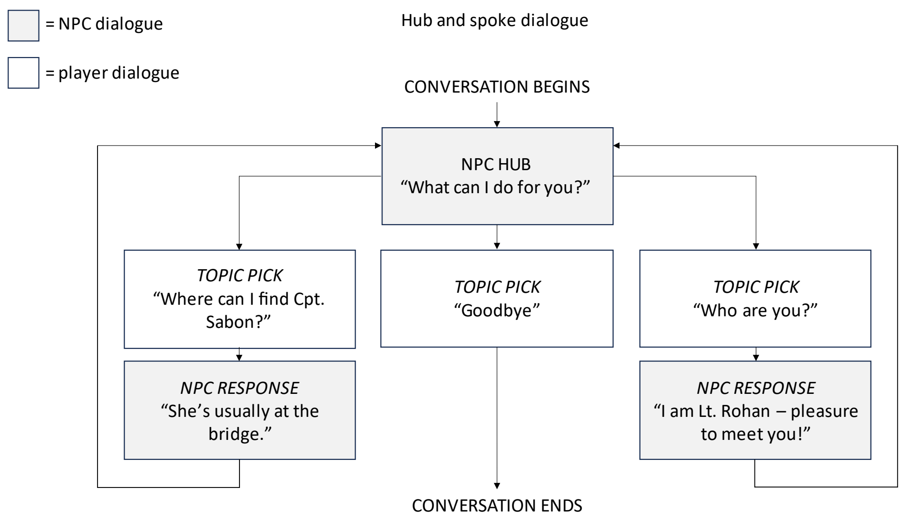
```
6   func _on_opacity_entered(area):
7       if area.name == "CharacterFeetPosition" or area.name == "EnemyFeetPosition":
8           overlaps += 1
9           treeSprite.modulate.a = 0.5 #go transparent so what is behind me is now visible
10
11  func _on_opacity_exited(area):
12      if area.name == "CharacterFeetPosition" or area.name == "EnemyFeetPosition":
13          overlaps -= 1
14          if overlaps == 0: #if nothing is behind me
15              treeSprite.modulate.a = 1
```

Figure 34. *Two functions, called by signals, belonging to the tree's sprite script.*

Dialogue

The in-game dialogue system was created with inspiration from many classic role-playing games (games wherein the player takes an active part in the narrative, usually engaging in dialogue with characters in-game and presuming a role in the universe) that include an interactive, branching conversation system. In games that include interactive conversation systems, the player usually interacts with the conversation by picking ‘replies’ or ‘topics’ on-screen; characters in-game then react to what the player has chosen. This may lead to more responses from the player, and the cycle continues. Freed (2014) writes that there are ‘two foundational structures for branching conversations: hub and spoke structures and waterfall structures’, though sometimes a mixture is used.

Freed writes that in a hub and spoke structure, the player chooses dialogue options at a ‘hub’, which can be defined as a neutral, generic opening statement from the NPC such as ‘hello! What can I do for you’ or ‘do you have any questions?’. The conversation then moves to a dialogue tree of the player’s choice based on whatever topic the player chooses. When the dialogue tree is exhausted, the hub is reached again, and the player may choose a new conversation topic. Exiting dialogue with an NPC is usually achieved via a ‘goodbye’ option. The hub and spoke system is a good for conveying as much information as possible to the player, and rarely does any dialogue go unseen by the player, unless they simply choose not to pick a topic. However, Freed notes that the system is ‘game-y’ and ‘immersion-breaking’ as every conversation becomes an interrogation of sorts, of constant questions by the player. See figure 35 below for a diagram of the hub and spoke system.



= NPC dialogue
= player dialogue
Hub and spoke dialogue
CONVERSATION BEGINS
NPC HUB “What can I do for you?”
TOPIC PICK “Where can I find Cpt. Sabon?”
TOPIC PICK “Goodbye”
TOPIC PICK “Who are you?”
NPC RESPONSE “She’s usually at the bridge.”
NPC RESPONSE “I am Lt. Rohan – pleasure to meet you!”
CONVERSATION ENDS

Figure 35. A diagram of the hub and spoke dialogue system. Note the circular structure, in that all topics except 'goodbye' send the tree to the hub again.

Freed writes that the alternative, the waterfall structure, sees the player never go backwards in conversation; response options are never repeated. The conversation 'moves forward until it reaches a predefined conclusion', which can be one of many. Player choices may 'branch the conversation and branches may or may not recombine', 'but a choice not taken is lost forever.' The advantage of the system is that conversations feel more natural and immersive, and a well-designed waterfall system is quicker for the player to use as they don't need to worry about having to trawl through lots of dialogue to find key information; the only thing the player needs to worry about is the impact their dialogue choices may have on the game. Freed also warns that developers should be careful not to 'bury key topics under player choices that may not be picked', due to the fact that the player will, under a waterfall system, not actually see the majority of the game's possible dialogue, therefore key information must be included in all possible NPC responses. See figure 36 below for a diagram of the waterfall system.

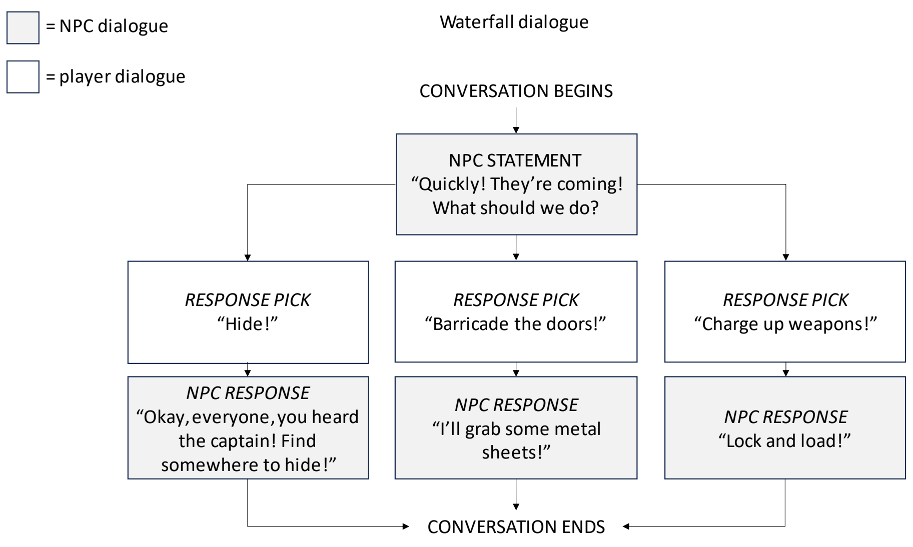
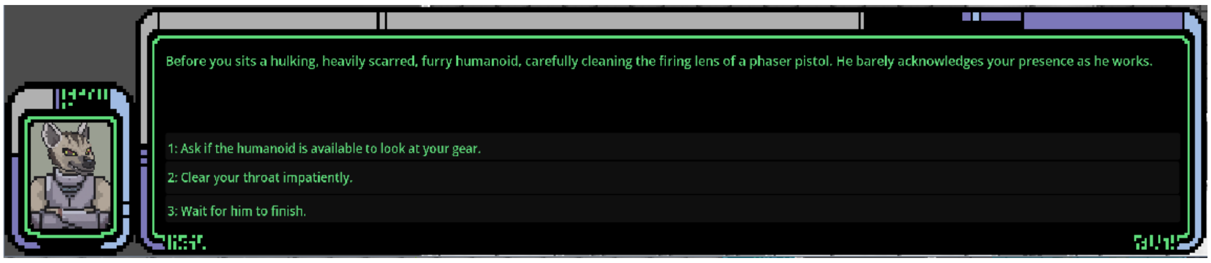


Figure 36. A diagram of the waterfall dialogue system. Note how, once a response has been chosen by the player, there is no way to 'restart' the conversation.

Which system is 'best' is down to the game it is being developed for, and many (if not most) modern games use a blend of the two – for example, Mass Effect (BioWare 2012), a sci-fi RPG series, allows the player to 'sidetrack from waterfall conversations into an 'investigate' submenu which offers a hub and spoke structure' (Freed 2014); Divinity: Original Sin 2 (Larian Studios 2017), a fantasy RPG game, has a blended dialogue system based on circumstances – talking to a key character who has a lot of information to provide follows a hub and spoke system, where the player can backtrack and ask different (or the same) questions, whereas being confronted by an enemy (for example, a goblin) with whom conflict can be avoided through dialogue, the game uses a waterfall system, as the route in dialogue the player takes can change the outcome of the encounter irreversibly. It is important to consider the requirements of the game being developed, or in the case of games with hybrid dialogue systems, which system to use at a given time, to create an effective and impactful experience that is fun for the player.

In the case of my own game, I chose to develop dialogue using a waterfall system as I intended for dialogue choices in the game to have consequences on relationships with crewmates, and I did not want the player to be able to backtrack on choices, to give a sense of realism and personability to the NPCs. Additionally, with how often the player returns to the spaceship, to their crew, they will always have comments and sentiments on both the player's actions on the mission, and updates to the situation on the ship since the player has gone, furthering the story, with the opening topic of conversation at any given time being tied to certain in-game events, programmed in the form of 'prerequisites' – e.g, Chief Aurora will only talk about the recent passing of a meteorite if said meteorite has passed the spaceship, and not before. Presentation of dialogue was achieved using a window that 'pops up' when dialogue is activated, with triggers including 'activating' an NPC (by pressing a key while standing near them), or just walking into their general area, as they may wish to initiate conversation themselves! The figure below displays the opening dialogue upon 'activating' Lt. Kraul's sprite by pressing 'E' on the keyboard while standing by him, which causes the player to stop moving, and the dialogue window to appear.


Before you sits a hulking, heavily scarred, furry humanoid, carefully cleaning the firing lens of a phaser pistol. He barely acknowledges your presence as he works.

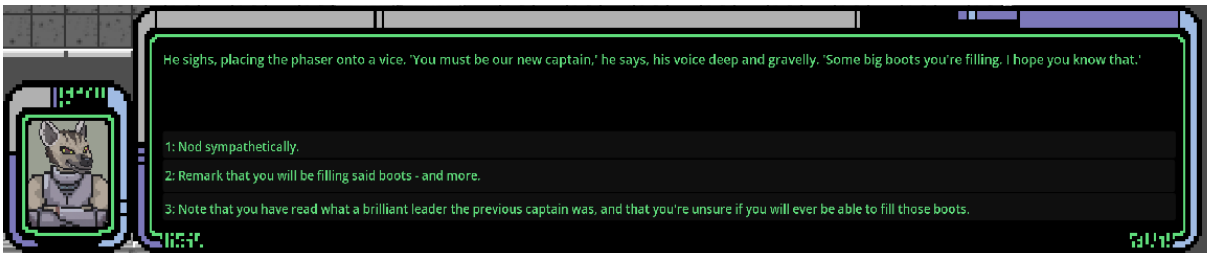
1: Ask if the humanoid is available to look at your gear.

2: Clear your throat impatiently.

3: Wait for him to finish.

Figure 37. The dialogue window, which currently tells of how the player has just encountered Lt. Kraul for the first time, giving them three potential ways of proceeding.

The player can see who is speaking on the left, as their portrait appears in the smaller portrait window (in case dialogue involving multiple characters occurs, in which case the portrait switches to whoever the active speaker is). The main window describes the situation, and what the NPC says. The player is then presented with one to three potential responses, which can be chosen by simply clicking one of them. The main window then updates, displaying the NPC's own response, and offering the player more options to select. This gives the dialogue system its 'branching' quality, as different options will lead the conversation in different directions, which have their own options! See the figure below for Lt. Kraul's response to the player picking the first option.



He sighs, placing the phaser onto a vice. 'You must be our new captain,' he says, his voice deep and gravelly. 'Some big boots you're filling. I hope you know that.'

1: Nod sympathetically.

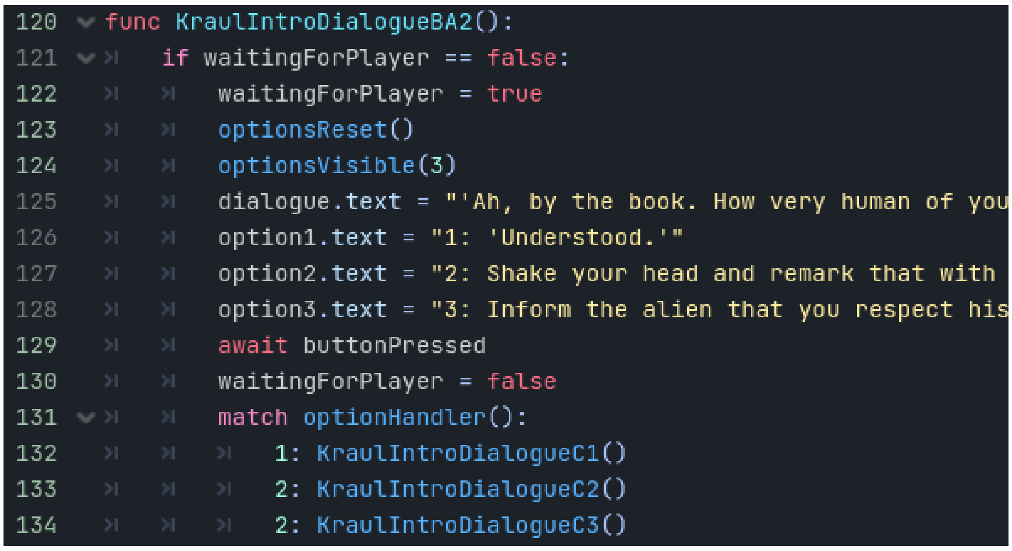
2: Remark that you will be filling said boots - and more.

3: Note that you have read what a brilliant leader the previous captain was, and that you're unsure if you will ever be able to fill those boots.

Figure 38. Kraul seems grumpy, and he leaves the player wondering if maybe they should have been more patient!

When a dialogue tree ends, the dialogue window closes, and the player's character can now move around again.

To program the dialogue system, I simply had to enclose each 'branch' within a function, giving it a sequential name, such as 'KraulIntroDialogueB2', which indicates the dialogue belongs to the KraulIntroDialogue, and is one 'layer' in, indicated by the 'B', and the player chose the second dialogue option in layer A, indicated by the '2'. Below is the code for one such dialogue branch:



```
120 func KraulIntroDialogueBA2():
121     if waitingForPlayer == false:
122         waitingForPlayer = true
123         optionsReset()
124         optionsVisible(3)
125         dialogue.text = "'Ah, by the book. How very human of you
126         option1.text = "1: 'Understood.'"
127         option2.text = "2: Shake your head and remark that with
128         option3.text = "3: Inform the alien that you respect his
129         await buttonPressed
130         waitingForPlayer = false
131         match optionHandler():
132             1: KraulIntroDialogueC1()
133             2: KraulIntroDialogueC2()
134             2: KraulIntroDialogueC3()
```

Figure 39. A code sample, displaying how dialogue for a specific branch of dialogue with Lt. Kraul is handled.

Lines 121 and 122 ensure that multiple dialogue instances cannot open at the same time. 123 and 124 handle the list of options a player can pick, calling two separate functions, the first of which 'resets' them by clearing any string information displayed and then hiding them all (making them unclickable), and the second function then causes as many options as the dialogue branch needs to become visible (an input value of two or lower will keep the rest of the options hidden and unclickable). Lines 125 to 128 modify the text boxes within the dialogue window to display the stated strings to the players. Line 129 then pauses the script, until one of the options is clicked by the player. Line 130 declares that the player has clicked an option, and that the dialogue box can be changed. Lines 131 to 134 handle what should happen when each dialogue option is clicked: they can be calls to new dialogue branch functions, or even something else entirely, such as a function to close dialogue completely and damage the player!

Evaluation

In this section, I evaluate the end-product, tying it back to the core aims of this project, and analyse the success of each aim, highlighting any issues with the project.

The game, while unfinished, remains as a strong foundation for a larger, more fleshed-out product. Game development is a process that can take years of work, but a strong foundation is crucial to a strong, problem-free development, to avoid time-consuming reworks and bugfixes that could have been easily avoided by ensuring the foundations of the game are right first time. I also believe that using standard game development practices helped the project immensely for numerous reasons. The first such reason is the GDD; it was instrumental in keeping the project on track, and having the game planned out in advance in a single, easy-to-follow document allowed me to work continuously without having to stop and plan out what to do next; I always knew what feature needed working on, and reminding myself about what steps I needed to take was easy due to the clarity that the GDD provided by offering a continuous reference point. Additionally, working in an agile manner, as is common in the industry, allowed me to respond flexibly to my supervisor's suggestions after each meeting by readdressing features that needed to be worked on further or dropping features that weren't a requirement and working on something else.

The game includes important, standard gameplay features like movement, player and enemy health (with player health displayed on the in-game UI), loss-conditions (a mission is lost if the player reaches 0 hit points from taking too much damage from enemies), a branching dialogue system, a map for the player to explore, and an in-game currency (the collection of which is updated on the in-game UI). Currently, as the game is unfinished, currency does little more than update the UI, and no shops or other ways to spend it exist in-game. Additionally, there is no way for the player to heal after taking damage. Most video games use 'health packs' or 'hearts', consumable items found around the map or obtained as a reward for beating enemies that heals a portion of the player's health. Another glaring issue is the lack of any sound or music for the game, which was another issue that came down to small amount of available time I had to work on the project. Playing a completely silent game is jarring and immersion-breaking, as there is no thematic or genre reason for the game to be silent.

The game is sufficiently science-fiction themed, appropriately inspired by science-fiction media: the player character is wearing futuristic, space-themed armour; the tile set used to build the game's environment is inspired by depictions of space stations and space ships; the in-game UI is inspired by the visuals of the LCARS computer system from Star Trek; in-game dialogue uses naval terminology which is common in science-fiction media (for example the use of naval ranks like Captain, Lieutenant etc., which is seen in Star Trek, Halo, Red Dwarf, and Star Wars, to name a few). If there was more time to work on the project, the inclusion of more sci-fi-inspired enemy types would have benefitted the game; currently, the only enemy type is a spider, which as the sole enemy of the game, feels more like something out of a fantasy game than science-fiction, though this is purely opinion-based. Humanoid enemies, perhaps ones armoured similarly to the player, who use laser weaponry, would potentially be more befitting of a science-fiction game, and allows the player to make use of the cover system that is currently in-game.

I included plans for implementation of a diverse cast of NPCs for the player to interact with, with whom I included different characteristics, such as different sexualities or trans identities (Commander Sabon is lesbian; Lt. Rohan is asexual; Chief Aurora is a transgender woman), disabilities/neurodivergence (Lt. Kraul uses a wheelchair and has post-traumatic stress disorder; Lt. Rohan is autistic), and cultural identities (Cmd. Sabon is a native central African; Lt. Rohan is south Indian; Lt. Kraul and Chief Aurora are aliens). Implementation of all four characters into the game unfortunately was not achieved due to the amount of time I had to work on the game, however a portrait sprite of Lt. Kraul and opening dialogue the player can have with him was completed so as to showcase the dialogue system; this specific aim could have been achieved at a greater level had I reallocated time away from sprite work to programming more dialogue, as then there would be more interaction with the NPCs, allowing players to learn about the characters through play.

The dialogue system of the game was something that worked fine, although it left me wondering if there was a more efficient way of programming it. In the interest of 'future-proofing' the game, with continued development, as more and more dialogue is added to the game, it would be prudent to reduce the 'size' of each dialogue entry by removing as much duplicate information as possible between them, ideally with each entry just containing string (raw text) information and little else, unless absolutely necessary. The current system, where each dialogue entry is contained as its own function that handles both the lexical contents of the dialogue and also the handling of the player's available dialogue options and calls to functions (where the player's choice leads the dialogue), clearly contains a lot of superfluous information.

With some refactoring, we could take inspiration from the template method design pattern, which is defined by Refactoring Guru as 'behavioral design pattern that defines the skeleton of an algorithm in the superclass but lets subclasses override specific steps of the algorithm without changing its structure.' Using this, a better system could be a single master function that takes a single input: an object containing no more than an array with up to four entries containing raw string data (one entry being text for the NPC's speech, then the other three being up to three possible options), and then an array containing the variable names of the dialogue object each player response dialogue leads to. These objects could optionally contain more data to override the algorithm, such as references to the required portrait needed to show which NPC is talking, or even methods that can be called that alter the game state (for example, if a dialogue option opens a door, or damages the player). This would lead to a refactored dialogue system that contains less bloated dialogue entries, and an easier way for developers to add more dialogue options.

It was a mistake that I did not think to use a version control system such as Git, as this is the norm in software and game development. While I did push the final version to a Github repository in the end, it did not really serve any purpose at that point. Had this been a project where I had worked with other developers, the use of Git would have been essential to keep track of working versions, avoid conflicts and signpost updates.

I write the final paragraph of this section with the knowledge that I could have easily fixed this issue in post and omitted any mention of it, but in the interest of highlighting the importance of care when including diversity and inclusivity in games, I chose to leave the mistake in as it makes for a good example of carelessness harbouring harmful stereotypes, and also an example that education, and/or exposure to diversity and cultural inclusivity in media really can change the views and opinions of those consuming it. Upon creation of the character of Lt. Rohan at the very start of the project, I made the creative decision for him to be both asexual and autistic. Months later, as this project wrapped up to a close, I realised that I had created a harmful stereotype, as there is an issue in media of autistic people being portrayed disproportionately as asexual (McAlpine 2018), due to the misconception that autistic people are likely to be asexual (when in reality, they are just as likely to be asexual as people who are not autistic). With this knowledge, further development of the game should continue with a different character being asexual instead, rather than outright removal of asexuality from the game, so as to avoid total erasure.

Conclusion

The development of this game was largely a success, as I fulfilled the majority of my aims and the project specification. The end-product is a solid prototype that, with continued development, can be turned into a marketable game. Science-fiction elements were central to the core design of the game, and this is clear through the game's visuals and gameplay, rendering an experience very clearly science-fiction inspired. The diversity shown in the game itself was somewhat scarce due to the foundations of the core game mechanics taking up the bulk of the project's time, though continued development, with the core of the game out of the way, can easily include a bigger focus on diversity and cultural inclusivity, considering there is a fully functioning dialogue system in-place where the bulk of these themes would be explored. I found that the game development process and its tasks are best shared among a team rather than a solo developer and recommend the use of the agile development methodology as it was very beneficial to the workflow of the project in the face of changing requirements and recommendations by the client (my supervisor). I found that I strongly recommend the creation and use of a game design document as it helped *immensely* with both the preproduction and production phases of the project. I also urge academics interested in video games to further contribute to literature on the topic of diversity in video games due to its currently small size, and I also urge game developers to create games that include diversity and cultural inclusivity, as even the smallest amount of inclusion makes a huge difference.

Reflection

The Godot Engine and GDScript

Godot Engine is often praised as a great choice for beginner game developers, and I agree that this is most certainly the case. As I stated earlier in the dissertation, I found its suite of tools to be very intuitive to use and I would readily recommend the engine to beginners and to those already in the software engineering field who want to dip their toes into game development. However, I cannot help but wonder if it was the right choice for me for a few reasons.

As I entered this dissertation with game design experience already, a part of me regrets not choosing a more advanced game engine (particularly Unity), to stretch my skills further and gain experience using an engine that is well-established in the game development industry. Godot is not often used outside of hobbyist game development due to its simplicity and lack of customisability compared to Unity or Unreal (though this may well change, considering how Roblox Studio has become very popular in recent years), and so Godot Engine experience is not as sought after in the game development industry compared to Unity or Unreal experience. Additionally, as it is a beginner-friendly engine, while I did learn a lot about the engine itself and how it works, I did not learn as much as I had hoped about game development as a whole, aside from my preliminary reading on the subject; the level of complexity Godot reached was no more complex than using Roblox Studio, something I have years of experience using.

My decision to use Godot's built-in language, GDScript, while being a good choice accompanied by my decision to use Godot (as the language was specifically designed for use with the engine), was possibly not the choice that would have developed my skills the most. By using another language Godot offered support for, such as C# or C++, or by using a different game engine like Unity, which would have forced me use C#, I may have felt a greater sense of achievement and personal development; my reasoning for this is that when I began the course for which this dissertation is being submitted, I was already familiar with Python and Lua, programming languages similar to GDScript, and I was also familiar with JavaScript and HTML5; during the course I learned Java, a language that was new to me, but bears semblance to C#. C# would have been a better language to use in terms of bettering my skills the most as in the game development industry, one of the most popular languages used is C# (alongside C++; Java; JavaScript and HTML5; and Lua (Carpenter 2021)) – so therefore, to expand my relevant programming language skillset to better my job prospects, I would need experience using either C++ or C#. GDScript is not a language popularly used in the game development industry, likely because Godot is seldom used by game development companies. Additionally, GDScript knowledge is not a transferrable skill to have outside of the world of Godot Engine game development, whereas C++ and C# are fairly common languages to see in other fields, such as in regular software development.

However, all-in-all, I am happy to have given Godot Engine a go; it was fun to use, and I never dreaded working on this project as Godot Engine made the experience very simple and fun to use. Additionally, I did end up developing a game that, while unfinished, is something I am personally quite proud of.

Working Alone

Looking at my game development process as a whole, I found developing this game to be a deeply rewarding, but slow process. Working as a solo developer, I found it hard (if not impossible) to find the time to juggle every part of a process that would normally be undertaken by multiple people with different expertise, though I at least got to experience what each person might perform in a game development team. Throughout this project, I played the roles of project lead, designer, writer, artist, and developer, and learned a lot about the responsibilities and tasks of each of these roles.

I found managing the project to be easy, frankly because I did not have a team to manage other than myself! I ideated the game myself and created plans and the GDD, and the rest of the project ran smoothly, following an agile workflow. My supervisor acted as my 'client', as I was always ready to change the course of development and focus on tasks based on his feedback and comments.

Designing and writing for the game became tricky, as it is very easy to write a plotline or design characters in your head or on paper, but I had to accept that it would be me who had to then develop or draw up these ideas. While I am confident of my skills as a developer, I am certainly not as confident in my artistic capabilities! This led to a lot of feature-cutting, such as numerous NPC sprites, enemy diversity, etc. as they would require a huge amount of artwork to create, and I had very little time in the grand scheme of game development to actually implement them.

As artwork in video games is important (though not always essential, such as in text-based games, or classic roguelikes that use ASCII symbols to signify player position, map design, etc.), I initially felt it imperative to get as much sprite work done as possible. Unfortunately, I feel as though I spent too much of the valuable time I had creating the sprites I needed and did not spend as much time on programming as I would have liked. I do concede that my game looks visually impressive, but I cannot help but wonder how much more of the game’s mechanics could have been developed were the sprites even simpler, or even existent at all. However, ultimately, I did write hundreds and hundreds of lines of code for the game, and it is an excellent prototype and base for something bigger: perhaps something I can work on after the dissertation period is over. With more work, many more inclusions to the game can be made, including a sound and music system, a title screen, a pause menu, saving the game, etc. However, in its current state, it is still something I am very happy to include in my portfolio, alongside this dissertation itself, and I am happy to have finished this dissertation with new skills and experience under my belt, and with my own contribution to academia on diversity in video games. I truly hope I get to work more on this project in the future as I had a lot of fun with its development, and it was such a great learning experience!

List of Diverse and Culturally Inclusive Sci-Fi Games

Below I have listed some good science-fiction games with a focus on diversity and cultural inclusivity, for further reading purposes. Please note that where publisher websites do not list their respective game, I have linked to the Steam page instead.

- 2064: Read Only Memories: http://2064.io/index.html
- Apex Legends: https://www.ea.com/en-gb/games/apex-legends
- Disco Elysium: https://discoelysium.com/
- Fallout: New Vegas: https://fallout.bethesda.net/en/games/fallout-new-vegas
- Half-Life Alyx: https://half-life.com/en/alyx/
- If Found: https://annapurnainteractive.com/en/games/if-found
- Mass Effect: https://www.ea.com/en-gb/games/mass-effect/
- Overwatch 2: https://overwatch.blizzard.com/en-gb/
- Rimworld: https://rimworldgame.com/
- Star Trek Online: https://www.playstartrekonline.com/en/
- Star Wars: Knights of the Old Republic: https://www.starwars.com/games-apps/knights-of-the-old-republic
- Starfield: https://bethesda.net/en/game/starfield
- The Last of Us Part I and II: https://www.playstation.com/en-us/the-last-of-us/
- The Outer Worlds: https://outerworlds.obsidian.net/en/enter
- The Walking Dead: https://store.steampowered.com/agecheck/app/1449690/
- Valorant: https://playvalorant.com/en-gb/
- Watch Dogs 2: https://www.ubisoft.com/en-gb/game/watch-dogs/watch-dogs-2

References

- ActivePlayer.io Game Statistics Authority, 2022. *Roblox: Roblox Live Player Count and Statistics.* Available at: https://activeplayer.io/roblox/# [Accessed: 3 August 2023].
- Agloo et al. 2023. Black Game Developers. Available at: https://www.blackgamedevs.com/ [Accessed: 11 September 2023]
- Aleem, S., Fernando Capretz, L. and Ahmed, F. 2016. Game development software engineering process life cycle: a systematic review. Journal of Software Engineering Research and Development 4. doi: 10.1186/s40411-016-0032-7
- Almeida, M.S.O. and da Silva, F.S.C. 2013. A Systematic Review of Game Design Methods and Tools (from the Entertainment Computing – ICEC 2013, ICEC 2013, 2013). *Lecture Notes in Computer Science* 8215. doi: 10.1007/978-3-642-41106-9_3
- Andrade, A. 2015. Game engines: a survey. EAI Endorsed Transactions on Serious Games 2(6). doi: 10.4108/eai.5-11-2015.150615
- Atlassian, N.D. What is the Agile methodology? Available at: https://www.atlassian.com/agile [Accessed: 10 September 2023]
- Avalanche Software. *Hogwarts Legacy*. Burbank: Warner Bros. Games.
- Bailey, K. 2022. *Rockstar Quietly Removes GTA 5 Content Described As Transphobic From New-Gen Releases.* Available at: https://www.ign.com/articles/rockstar-gta-5-transgender-content-removed [Accessed: 15 August 2023].
- BioWare. 2012. *Mass Effect.* Alberta: BioWare.
- Blizzard Entertainment. 2016. *Overwatch.* Irvine: Blizzard Entertainment.
- Bond, M. and Beale, R. 2009. What makes a good game? Using reviews to inform design. *Proceedings of the 2009 British Computer Society Conference on Human-Computer Interaction, BCS-HCI.* Cambridge, 1-5 September 2009. doi: 10.1145/1671011.1671065
- Boords. N.D. *Boords.* Available at: https://app.boords.com/ [Accessed: 11 September 2023]
- Buday, R. Baranowski, T. and Thompson, D. 2012. Fun and Games and Boredom. Games For Health Journal 1(4), pp. 257–261. doi: 10.1089/g4h.2012.0026
- Carpenter, A. 2021. *6 Most Popular Programming Languages for Game Development.* Available at: https://www.codecademy.com/resources/blog/programming-languages-for-game-development/ [Accessed: 5 September 2023]

- Cicchirillo, V. and Appiah, O. 2014. The impact of racial representations in video game contexts: Identification with Gaming Characters. *New Media and Mass Communication* 26.
- Colby, R. and Colby, S. R. 2019. Game design documentation: four perspectives from independent game studios. Communication Design Quarterly 7(3). doi: 10.1145/3321388.3321389
- Crytek. 2002. CryEngine. [software]. Available at: https://www.cryengine.com/
- Entertainment Software Association. 2020. 2020 Essential Facts About The Video Game Industry. Washington D.C: Entertainment Software Association.
- Entertainment Software Association. 2021. 2021 Essential Facts About The Video Game Industry. Washington D.C: Entertainment Software Association.
- Entertainment Software Association. 2022. 2022 Essential Facts About The Video Game Industry. Washington D.C: Entertainment Software Association.
- Entman, R. 1989. How the Media Affect What People Think: An Information Processing Approach. The Journal of Politics 51(2), pp. 347 – 370. doi: 10.2307/2131346
- Epic Games, 1998. *Unreal Engine.* [software]. Available at: https://www.unrealengine.com/en-US
- Fabricatore, C. 2007. Gameplay and game mechanics design: a key to quality in videogames. *OECD-CERI Expert Meeting on Videogames and Education.* Santiago de Chile, Chile, 29-31 October 2007. doi: 10.13140/RG.2.1.1125.4167
- Fowler, J. 2023. Dissertation Proposal. *CMT221 Topics, Research and Skills in Computing.* Cardiff University. Unpublished Assignment.
- Freed, A. 2014. Branching Conversation Systems and the Working Writer, Part 2: Design Considerations. Available at: https://www.gamedeveloper.com/design/branching-conversation-systems-and-the-working-writer-part-2-design-considerations [Accessed: 9 September 2023]
- GaymerX. 2013. GaymerX. Available at: https://gaymerx.org/ [Accessed: 11 September 2023]
- Godot, N.Da. Godot Engine 4.2 documentation in English: GDScript reference. Available at: https://docs.godotengine.org/en/stable/tutorials/scripting/gdscript/gdscript_basics.html#doc-gdscript [Accessed: 23 August 2023]
- Godot, N.Db. Godot Engine 4.2 documentation in English: Internal rendering architecture. Available at:

https://docs.godotengine.org/en/stable/contributing/development/core_and_modules/internal_rendering_architecture.html [Accessed: 23 August 2023]

- Harris, J. 2021. *Exploring Roguelike Games.* Boca Raton, Florida: CRC Press.
- Howarth, J. 2023. *How Many Gamers Are There? (New 2023 Statistics).* Available at: https://explodingtopics.com/blog/number-of-gamers [Accessed: 15 August 2023].
- KDE. 2005. *Krita*. Version 4.4.0. [software]. Available at: https://krita.org/en/download/krita-desktop/
- Larian Studios. 2017. *Divinity: Original Sin II.* Larian Studios: Ghent.
- Linietsky, J. et al. 2023. *Godot Engine.* Version 4.0.3. [Software]. Available at: https://godotengine.org/ [Accessed: 4 September 2023]
- Marklund, B. et al. 2019. What Empirically Based Research Tells Us About Game Development. *The Computer Games Journal* 8, pp. 179 – 198. doi: 10.1007/s40869-019-00085-1
- McAlpine, M. 2018. *Autistic =/= Asexual.* Available at: https://marykatemcalpine.medium.com/autistic-asexual-3a9488435b2a [Accessed: 8 September 2023]
- Mozilla. N.D. *Tiles and tilemaps overview.* Available at: https://developer.mozilla.org/en-US/docs/Games/Techniques/Tilemaps [Accessed: 11 September 2023]
- Murphy-Hill, E. et al. 2014. Cowboys, ankle sprains, and keepers of quality: how is video game development different from software development? ICSE 2014: Proceedings of the 36th International Conference on Software Engineering. Hyderabad, India, 31 May – 7 June. doi: 10.1145/2568225.2568226
- Osborne-O'Hagan, A. et al. 2014. Software Development Processes for Games: A Systematic Literature Review (from the EuroSPI 2014: Systems, Software and Services Process Improvement, Luxembourg, 25-27 June 2014). Communications in Computer and Information Science 425 pp. 182-193. doi: 10.1007/978-3-662-43896-1_16
- Pickell, D. 2019. *The 7 Stages of Game Development.* Available at: https://www.g2.com/articles/stages-of-game-development [Accessed: 22 August 2023]
- Pixel Overload. 2020. *What Canvas Size Should you use for Pixel Art? (Pixel Art Tutorial).* Available at: https://www.youtube.com/watch?v=Z8earctNBxg&ab_channel=PixelOverload [Accessed: 24 August 2023]

- Refactoring Guru. N.D. *Template Method*. Available at: https://refactoring.guru/design-patterns/template-method. [Accessed: 10 August 2023]
- Roblox Corporation. 2016. *Roblox Studio.* [software]. Available at: https://create.roblox.com/
- Rockstar Games, 2013. *Grand Theft Auto V*. New York, Rockstar Games.
- Roddenberry, G. 1990. *Star Trek: The Next Generation.* CBS, 28 September.
- Roddenberry, G. 1993. *Star Trek: Deep Space Nine*. CBS, 3 January.
- Rogers, E. 2022. *5 Upcoming indies from LGBTQIA+ owned Game studios.* Available at: https://indiegamefans.com/5-upcoming-indies-from-lgbtqia-owned-game-studios/ [Accessed: 15 August 2023].
- Sirani, J. 2023. *The 10 Best-Selling Video Games of All Time.* Available at: https://www.ign.com/articles/best-selling-video-games-of-all-time-grand-theft-auto-minecraft-tetris [Accessed: 15 August 2023].
- Stojanovic, M. 2023. *Gamer Demographics from 2023: No Longer a Men-Only Club.* Available at: https://playtoday.co/blog/stats/gamer-demographics/ [Accessed 15 August 2023].
- TheGamer. 2017. 10 Times Games Got Diversity Right (And 10 They REALLY Didn't) Available at: https://www.thegamer.com/10-times-games-got-diversity-right-and-10-they-really-didnt/ [Accessed: 10 September 2023]
- Toftedahl, M. 2019. *Which are the most commonly used Game Engines?* Available at: https://www.gamedeveloper.com/production/which-are-the-most-commonly-used-game-engines- [Accessed: 10 September 2023]
- Unity Technologies. 2005. *Unity*. [software]. Available at: https://unity.com
- Vohera, C. et al. 2021. Game Engine Architecture and Comparative Study of Different Game Engines. *12th International Conference on Computing Communication and Networking Technologies (ICCCNT). pp. 1-6. doi: 10.1109/ICCCNT51525.2021.9579618.*
- Ward, F. and Ross, C. *A breakdown of the J.K Rowling transgender comments, as fans boycott the new Harry Potter TV series.* Available at: https://www.glamourmagazine.co.uk/article/jk-rowling-transphobia-explained [Accessed: 15 August 2023]