School of Electronic Engineering
and Computer Science

Project Report

Programme of study:
Computer Science

Project Title: GUN FOR ALL

Supervisor:
Kit Mills Bransby

Student Name:
Laurence Dunbar

Final Year
Undergraduate Project 2024/25

Queen Mary
University of London

Date: 06/05/2025

Abstract

Multi-window functionality is a relatively new and experimental feature in the sphere of video games, allowing a game to make use of an operating system's native windows and incorporate them in various ways, such as allowing a character to move between different open windows or having an enemy disrupt the player by affecting the size of a window

Ultimately, this mechanic is underutilised; few games currently exist that use multi-window functionality at all. Therefore, the objective of this project is to create a game in the popular platformer roguelike genre that utilises said functionality.

As many games currently exist under the platformer roguelike genre, this project will focus on the multi-window functionality to bring a novel and unique approach to the genre by implementing various mechanics and gameplay elements that cannot be achieved without multi-window functionality.

Contents

Chapter 1: **Introduction**

1.1 Background

About a year ago, while searching for game development inspiration, I managed to come across a newly released game made by a single developer called Torcado called WindowKill. It was the first time I had encountered a game with any sort of multi-window mechanic that could affect gameplay.

Multi-window functionality is the ability of a game to act as a window manager. Sanders (2018) outlines that the primary responsibility of a window manager is ”to keep track of all aspects of each of the windows being displayed” particularly calling attention to the ability of windows to overlap, be hidden, be cascaded and stretched.

In this way, the game would be able to directly create various windows and control aspects of them, such as their position or size, in conjunction with in-game actions or events. For example, enemy actions could affect the size of particular windows on screen, or perhaps a window could follow the player’s position on-screen.

I was immediately interested to find more examples of similar games to show the depth of this novel idea but quickly discovered that of the 2 other prominent examples I could find, both were made by the same developer. This then led me to the idea of developing my own game that would utilise multi-window mechanics in a novel and unique way.

My idea culminated in the plan to create a 2D game, *Gun for All*, in the roguelike-platformer[1] genre which could incorporate these multi-window mechanics. My choice of genres was made based on a few factors: I chose a platformer as they are not only simple enough to create within the time-frame of the project but also simple for the player to understand. I chose roguelike as the genre is, by design, endlessly replayable as they often make use of procedural generation to make every run unique. Furthermore, the roguelike and platformer genres are an effective and popular pairing as the latter’s simplicity works well to create a game where the difficulty is derived not from complexity but from effective use of strategy, careful execution and quick thinking.

[1] In this case, roguelike would refer to a genre of game characterised by permanent player death and procedurally generated levels. Platformer would refer to a genre of game characterised by level design centred around running and jumping, usually onto platforms. The terms are defined in more detail in Chapter 2

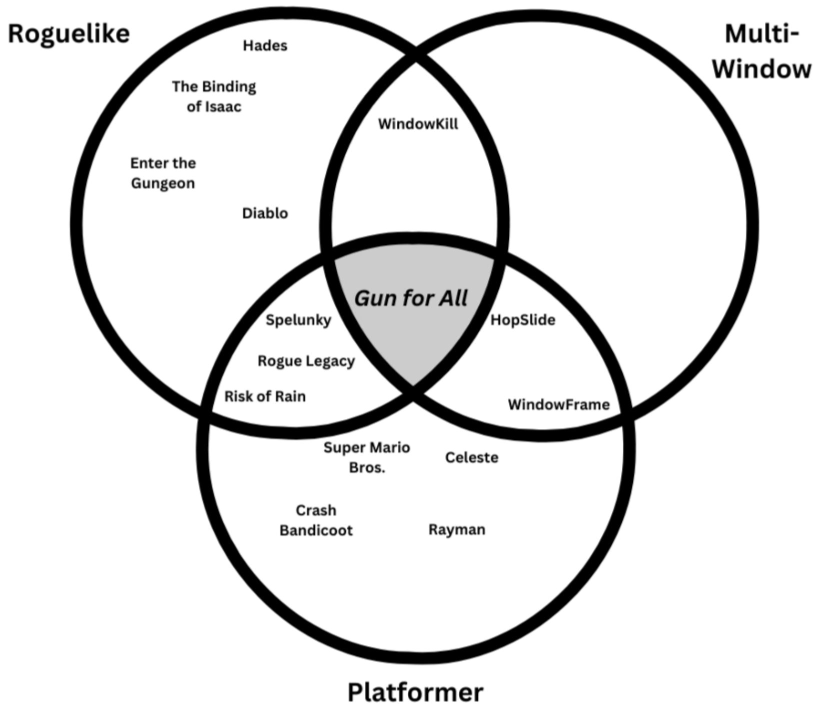


Figure 1: Venn Diagram of genre overlap with multi-window mechanics

In the figure above, I collated and categorised many popular examples of games in the aforementioned genres. As shown, *Gun for All* is the only example that exists as part of both genres and utilises multi-window mechanics.

1.2 Problem Statement

There is a clear lack of games that make use of multi-window mechanics of any sort. The mechanic is broad and adaptable and can be utilised in a range of different ways over a range of different genres. However, even within the games where they are utilised, there is limited variety in how this mechanic is used.

1.3 Aim

This project aims to create a 2D single player roguelike platformer game with procedurally generated levels and shooting based combat that also utilises multiple windows to create challenges for the player that would be impossible in standard roguelike platformers that would utilise single window interfaces

1.4 Objectives

- Develop a keyboard-and-mouse based control system
- Develop an endless an endless level system that will utilise a Procedural Content Generation Algorithm (PCG)

- Design an upgrade system to allow the player to improve their characters stats
- Design a system that will use multiple windows to show the player, the goal and boss enemies.
- Design enemies that will be able to manipulate window sizes, create new windows and destroy open windows
- Design a shop system that will allow the player to purchase different weapons and items that have different effects
- Design a health system that will carry over through subsequent rounds

1.5 Research Questions

Through the implementation and evaluation of *Gun for All*, the following research questions were answered.

1. Can multi-window mechanics effectively be implemented in platformer roguelike games
2. Is there a positive benefit to adding multi-window mechanics to a game

1.6 Report Structure

Chapter two makes use of relevant, available literature in the field of video games and game development to explore and define, in depth, many of the important terms used throughout the report. The chapter also highlights and analyses similar existing games to compare them to *Gun for All*.

Chapter three details the choices made in the design of the project.

Chapter four provides a detailed explanation of the methodologies and tools used in the implementation of *Gun for All* and covers all the core elements of the game

Chapter five looks at the outcome of the project and, while considering its limitations, evaluates this against the initial aims, objectives, and research questions.

Chapter six will discuss a reflection of the project, highlighting the challenges faced and the achievements throughout the project. The chapter will also discuss any potential future work that could be done on the project.

The References chapter will contain a full list of sources referenced throughout the report and the Appendices will contain any additional information not highlighted by other chapters, such as diagrams and tables.

Chapter 2: **Background Research**

This chapter will use the available literature to explore and define some of the terms commonly used in this report. The report will then analyse some of the similar existing games.

2.1 Literature Review

2.1.1 Novelty in games

Novelty in video games is a concept often taken for granted. In an environment where games are often neatly and easily categorised into genres in which the majority of important gameplay elements are shared between other games of that genre, Novelty in games, especially newer games, can often feel superficial; it's difficult to surprise players who have played many games of the same genre before.

In the past decade, game development has become more and more accessible.For example, in a journal about independent game development as a craft, the author highlights initiatives focused on ”creating and supporting game making activities that are accessible to and inclusive of new voices” (Westecott, 2013).

There has been a growing number of independent developers trying to create their own interpretations of popular games and, in doing so, have often subverted many common aspects of their genres. Even these indie games, however, have spawned many games inspired by them meaning even their novelty has started to fade.

However, novelty remains an important factor in games and their appeal to players. In a study on difficulty, novelty and suspense, the author finds that ”moderate novelty maximizes player engagement” and ”the novelty effects may be particularly beneficial for simple games” (Peng et al., 2021).

Due to this, the design of *Gun for All* is focused on blending the familiarity of the popular roguelike genre with the novelty of multi-window mechanics to create this perfect ”moderate novelty” while also maximising the appeal of this novelty by introducing the simplicity of the platformer genre, and its related mechanics.

2.1.2 Roguelike

Roguelikes, as defined by Harris (2020), are games, set in a randomly generated world, known for their tremendous difficulty, unpredictability, and permanent character death. They are aptly named after the 1980 dungeon crawling computer game, Rogue, developed by Michael Toy and Glenn Wichman.

In Rogue (Toy et al., 1980), players seek to explore several levels of a dungeon with the goal to find the Amulet of Yendor which sits on the dungeon's lowest level. During their adventure, players can collect various treasures, such as weapons, armour, potions, scrolls that can help them against monsters that roam the dungeon. Due to the fact that Rogue is turn-based and that it famously implements player permadeath, in which the player cannot respawn and will have to start from the very beginning if they die, Rogue encourages careful thought and strategy behind each move. Combined with its other characteristic technique of procedural level generation, each time playing the game was a unique and difficult challenge that required continuous careful thought and planning.

The ”big three” roguelike games, as Harris (2020) describes them, are: NetHack (DevTeam, 1987), Angband (Contributors, 1990), ADOM (Biskup, 1994). These are all classic roguelike games, some of the first to exist, and are all still often played. Roguelikes, in their classic dungeon crawler form (which I will refer to as true roguelikes from here on), are a niche genre of games, even to this day, and reached the peak of popularity in the late 80s into the early 90s. However, during the mid 2000s, there was a resurgence in the genre through independent developers who combined the key elements of roguelikes, such as proceedural generation and permadeath and combined them with various other genres and styles of games, subsequently making them much more accessible to a wider audience.

2.2 Existing Applications

2.2.1 WindowKill

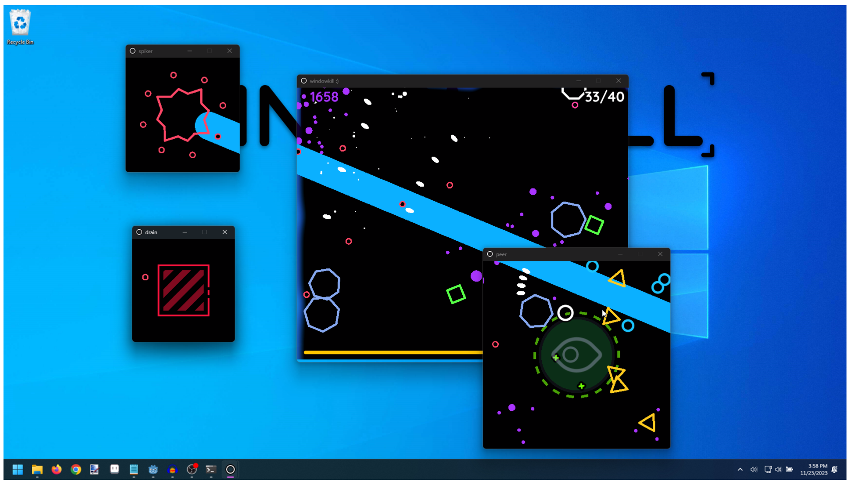
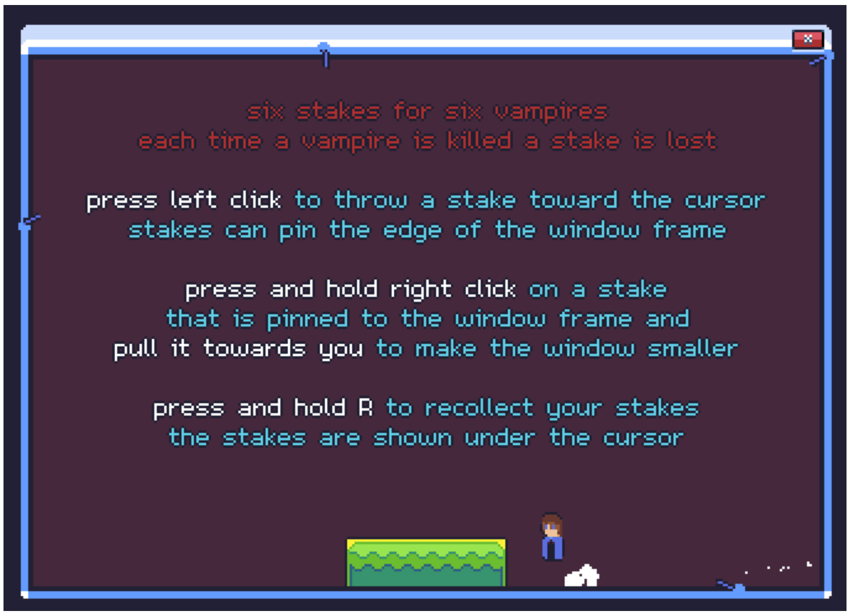


Figure 2: Gameplay from Windowkill

Windowkill (Torcado, 2022) is a multi-window shoot-em-up developed by Torcado. It acted as the main inspiration for *Gun for All*. In this game you control a circle that endlessly destroys other shapes in an attempt to survive as long as possible against enemies that can attack you in many different ways by manipulating the window.

This game both is a roguelike and uses multi-window mechanics but unlike *Gun for All* it lacks the platformer mechanics and loses a whole dimension of complexity because of it

2.2.2 WindowFrame


six stakes for six vampires
each time a vampire is killed a stake is lost

press left click to throw a stake toward the cursor
stakes can pin the edge of the window frame

press and hold right click on a stake
that is pinned to the window frame and
pull it towards you to make the window smaller

press and hold R to recollect your stakes
the stakes are shown under the cursor

Figure 3: Screenshot from WindowFrame

Windowframe (Linssen, 2015) is a pseudo multi-window game developed by Daniel Linssen. It also inspired *Gun for All* in a few ways. In the game you control a character making their way through stages by adjusting the sides of the window the player is in.

However, this game only interacts with the window in one way, lacks the difficulty and replayability of the roguelike genre, and as the windows aren't true system windows but a fake window generated inside the game, it loses a level of immersion

2.3 Requirement Review

The section below will outline all the functional and non-functional requirements gathered throughout the chapter

2.3.1 Functional Requirements

1. The system should allow players to start a new run from the main menu
2. The system should create a new level every new run
3. The system should allow players to change their sensitivity and controls from the main menu
4. The system should allow players to control their character using the keyboard and mouse
5. The system should prevent players from being able to directly manipulate the position of the enemy windows
6. The system should close all game windows if the player closes the player (main) window

7. The system should allow the player to access a shop each round
8. The system should allow the player to buy items/abilities while in the shop
9. The system should allow the player to upgrade their stats while in the shop

2.3.2 Non-Functional Requirements

1. The system should be able to load each round within 5 seconds
2. The system should have a filesize below 10GB
3. The system should run on computers that use Windows

2.4 Summary

Considering the findings of the chapter, there is perfect motivation and equally perfect opportunity to create *Gun for All*. There is not only a need for novelty in games, which can be provided through multi-window mechanics, but the roguelike platformer genre gives an excellent opportunity to introduce engaging new mechanics

Chapter 3: **Design**

This chapter will discuss the high-level design of the game. It will outline the structure of the game and any design patterns used, describe the relationship between each subsystem, and discuss in detail the reasons behind each.

3.1 Structure Overview

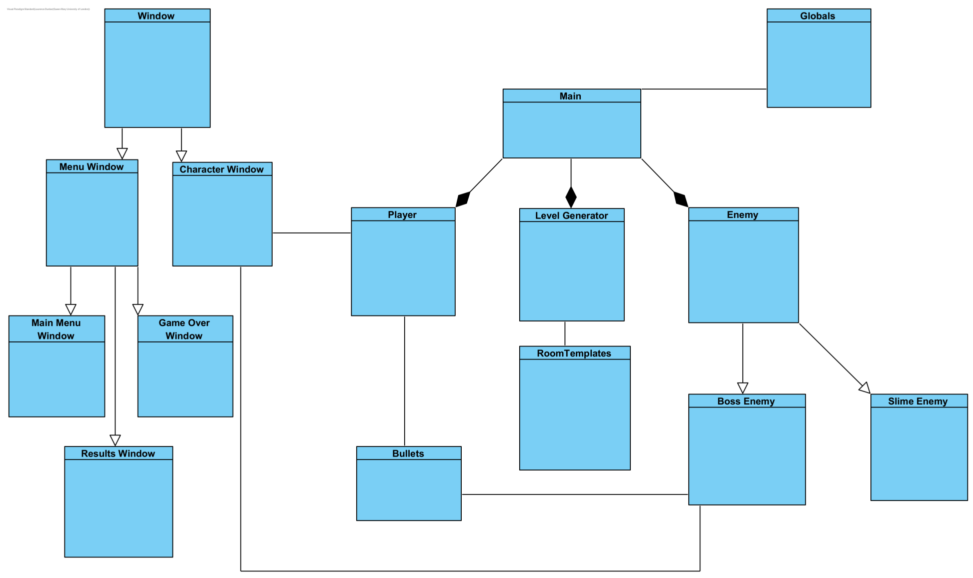


Figure 4: Class diagram depicting general game structure

In its simplest form, *Gun for All* will consist of a procedurally generated level, which is traversed by a player, alongside 2 types of enemy: a boss enemy, and a slime enemy.

The figure above displays a simple depiction of the relationship between each of the game's systems. They will be discussed in depth during this chapter

3.2 Procedural Generation

In Chapter One, the 2nd objective described an "endless level system". In this case, an endless level system would mean a system that would be able to continuously provide the player with levels they could play until they eventually fail. The only way this can be achieved is through the use of a Procedural Content Generation algorithm (PCG), which would, upon the player clearing a level, randomly create a new one influenced by certain parameters.

Level design is a core part of any platformer, and is perhaps the most important factor in its challenge and difficulty. Therefore choosing a fitting PCG is vital.

Considering the genres of platformer and roguelike and the multi-window mechanics, the PCG must meet a few requirements:

1. Each stage must have coherent level design so that the player encounters realistic challenges on their path to beating it

2. The levels need to have enough elements that can be randomised to avoid being repetitive
3. The stage must have consistent dimensions and scale

With the requirements in mind there are 2 PCGs, commonly used in platformers, that might apply. The first is a noise-based algorithm, like Simplex noise or Perlin noise, and the second is a grid based algorithm.

3.2.1 Noise-based algorithm

Developed in 1983 by Ken Perlin, the Perlin noise algorithm (Perlin, 1985) was created in response to his frustration with the ”machine-like” look of computer graphics (Perlin, 1999), his algorithm aimed to create more natural looking constructs than fully random generation. In 2001, Perlin created the Simplex noise algorithm (Perlin, 2002)as a more efficient, improved version of his prior algorithm, however they both function similarly.

The algorithm works by creating a grid and assigning a gradient vector to each point in this grid. When sampling a point in space, the algorithm finds the 4 surrounding points and calculates the dot products between the gradient each point and the vector from the grid point and the sample point. Finally the algorithm interpolates these values using a smooth curve e.g. cubic interpolation. The result is a smooth, continuous surface of values that varies gradually and naturally.

In the context of a 2D platformer, this algorithm can be applied by taking the generated noise and rounding each point to 1 or 0, this grid of 1s and 0s can then be turned into either solid tiles or empty space. This results in a cave-like structure that can be navigated through.

This is good as it generates a natural looking level that won’t be repetitive. However, its drawback is that level design cannot be guaranteed. levels will often be plain and will need to be added to significantly if you were to use it to generate an engaging level.

3.2.2 Grid-based algorithm

The algorithm I decided to use in *Gun for All* was a variation of a grid-based algorithm. A grid-based algorithm is a way of dividing space into a regular grid and then using it to guide how things are connected. Each cell in the grid represents a small area of the world, and the algorithm works by filling and linking these cells based on rules.

The way that grid-based algorithms work vary greatly and each implementation does it in a different way. However, for *Gun for All*, my main inspiration was the algorithm used in Derek Yu’s Spelunky which is an example of modular room-based procedural generation.

In essence, this algorithm works by stitching together hand-made, room templates and creating unique levels using them. In Spelunky, the room templates are categorised into different types, which are determined by their exits: left, right, top or bottom. When the level is generated, a 4x4 grid is created, a point on the top row is chosen and a room with left and right exits is selected. A direction is then randomly picked and the algorithm progresses to the next cell. This room, now, must be of a type that has an exit that connects to the previous room. New directions are repeatedly selected until the algorithm reaches the bottom row, if the algorithm tries to progress down while on the bottom row, it stops, places an exit, and fills in the remaining squares with random rooms which aren't guaranteed to connect. This results in a structured but still varied level.

This is perfect for *Gun for All* as it matches all of the previously defined requirements. Details of *Gun for All's* implementation will be discussed in Chapter Four.

3.3 Window System Design

The window system is the crux of the project and the main contributor to the game's novelty. It was designed to add to the traditional platformer experience by including multiple windows that can be moved independently. Rather than having everything displayed within a single screen, like normal, the game is split up into a number of connected windows that interact with each other in various ways.

It adds a new aspect to the gameplay as the player needs to not only manage their in game actions, but also requires them to be aware of and manage, the positions and functions of different windows on their desktop making the game far more immersive.

In *Gun for All* this is implemented in a unique way: the player, and the boss of each level are given their own separate windows. The level generator will create a random level at the start of the game, its size will be static and proportional to the screen the game is being displayed on, unlike traditional platformer games, where the position and size of the level on the screen would change as the player travels through it. Since the full is never shown, the user can only interact with the game through these windows. This gives the effect of the player character moving around and exploring the user's desktop.

Several other types of windows are also used to interact with the game, like menus, game over screens and other types of information screens.

This sort of gameplay style can lead to interesting and unique challenges that are completely unachievable in traditional, single-window game design. In order for this system to work, a few key goals must be met:

- The player must only be able to interact with the windows in specific ways e.g. Not all windows should be able to closed as this will disrupt the game
- Regardless of the player's actions, each window must be independently updated and match the game shown in the other windows to avoid confusion
- Each window needs to be clear and distinct, but not obscure the player unnecessarily

The specifics on how each system was implemented to match each goal is discussed in Chapter Four.

3.4 Game Flow

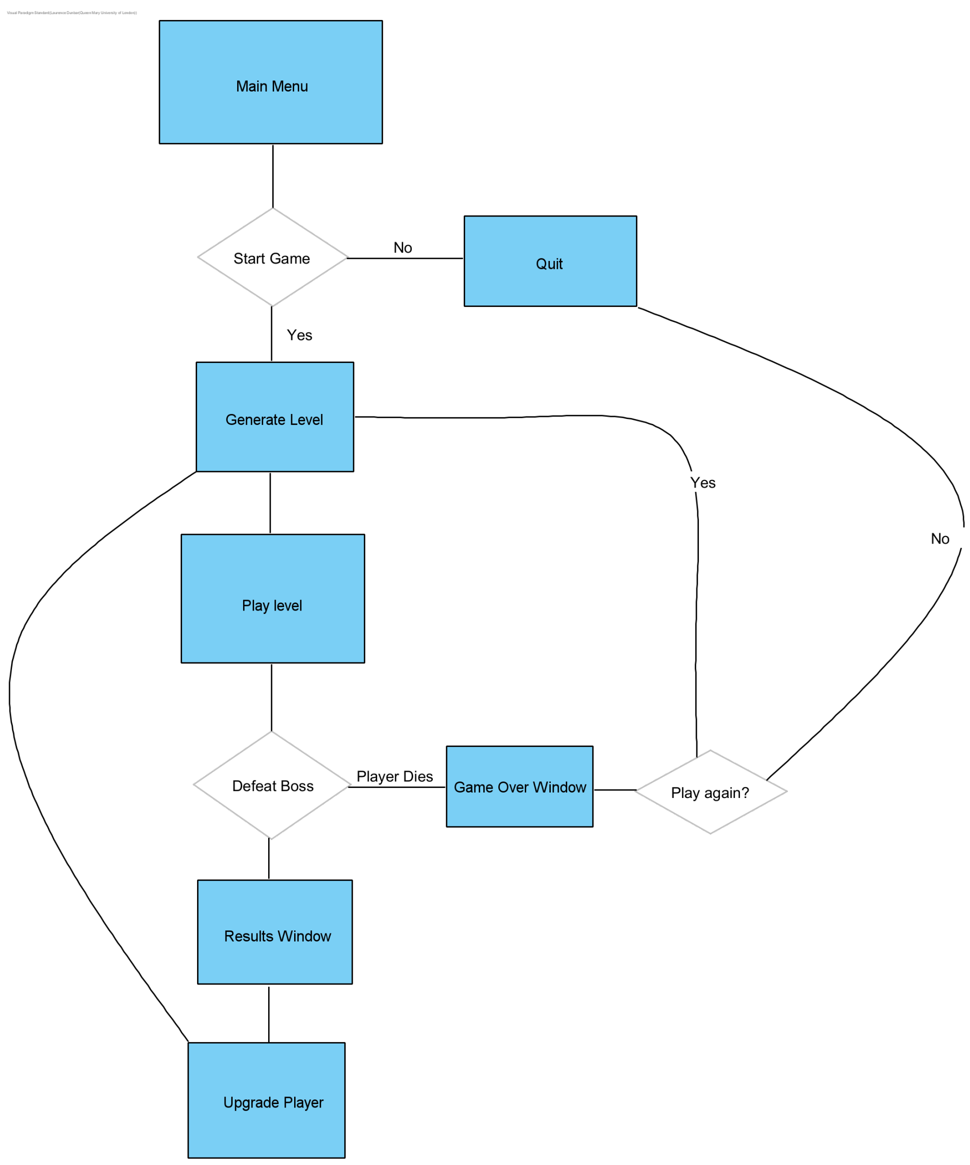


Figure 5: Flowchart depicting game flow

The flow of the game follows the traditional Roguelike style gameplay loop where a level will be generated, the player plays through the level and then gets to choose some upgrades. After starting the game, the player will be faced with a main menu where they will be able to start the game or quit. Upon selecting the option to start the game, a level will be generated and the player must play through the generated stage. If the player dies at any point the run will immediately end and the player will be presented with a game over window, at which point they will be prompted as to whether they'd like to continue or not, if the player refuses the game will end. However, if the player chooses to try again, the game will start again from the beginning and generate a new level. If the player manages to beat the level, a window will pop-up to show the player the results of the previous level and then allow them to pick some upgrades to strengthen the player. After this another level will immediately be generated with a slightly raised difficulty. This will continue indefinitely, until the player dies.

This gameplay loop works perfectly for *Gun for All* as it is simple and so it won't confuse the player when put alongside the more complex multi-window mechanics. It keeps the slower, less interesting menu navigation to a minimum and maximises the time the player can spend interacting with the more interesting multi-window systems.

3.5 Design Patterns

Across the design of *Gun for All* several design patterns were used to organise and structure the game's functionality. Some of the more notable examples are as follows.

3.5.1 State Machine

To begin with the most important use, a state machine was used in all of the character classes: player, boss enemy, and slime enemy to give them more complex behaviour, such as being idle, moving, jumping or chasing. In *Gun for All* they are managed by independent state classes connected to the character object that handle different behaviour vital to the character.

The state machine is an important pattern in this project as it allows characters to handle complex behaviour without relying on complex intertwined conditionals.

3.5.2 Singleton

The next most important pattern is the Singleton which is used in the Globals class. The Globals class is a class the holds important shared game data and functions that need to be accessed throughout the project.

A notable example of this is the main window[2], since every window is connected to it, when a new one is created, a reference needs to be made to the instance of the main window. The Globals singleton ensures that only one instance of the main window can ever be accessed at a time.

3.5.3 Factory

The factory pattern appeared throughout the project through the dynamic instantiation of bullets, slimes and windows at runtime. Rather than manually creating new instances of them, the game preloads the relevant classes and generates new instances of them at runtime where necessary.

This is excellent for future additions to the game as it is much easier to add more types of enemy, windows or bullets later on as the reduced coupling means that there are fewer places that need to be directly changed.

3.5.4 Observer

Communication between different parts of the system was done using the Observer pattern. For example when a slime or the player is defeated or when a menu button is selected, signals were emitted that were then received and responded to by other parts of the system.

[2]the specifics of the multi-window implementation is discussed further in Chapter Four

This was incredibly useful as it reduced the need for directly coupling different parts of the system through strings of references, which improved the modularity of the project.

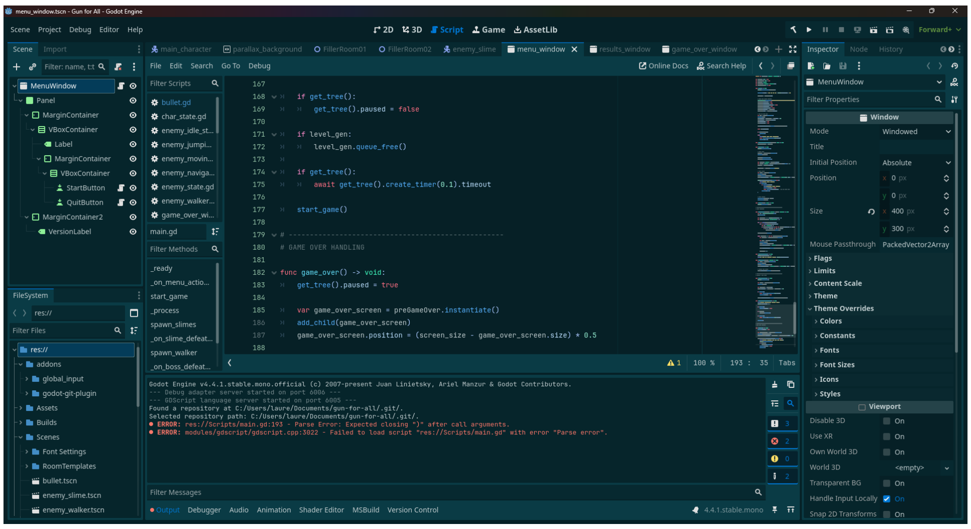
Chapter 4: **Implementation**

This chapter will follow up on the designs discussed in chapter Three by detailing the technical and practical implementation of the systems that comprise the project.

4.1 Development Environment

For most video games, the choice of development environment is an important one. The choice of engine can determine many different aspects of a game from what systems your game will be able to make use of to which platforms your game will be able to work on.

For *Gun for All*, however, the choice was quite simple. Due to the unique nature of multi-window functionality, very few engines have the capability to directly control system windows. As multi-window functionality was such an integral part of the project it greatly narrowed down the list of potential engines.After extensive research, the only fitting choice was an engine called Godot



```
	if get_tree():
		get_tree().paused = false

	if level_gen:
		level_gen.queue_free()

	if get_tree():
		await get_tree().create_timer(0.1).timeout

	start_game()

# -------------------------------------------------
# GAME OVER HANDLING

func game_over() -> void:
	get_tree().paused = true

	var game_over_screen = preGameOver.instantiate()
	add_child(game_over_screen)
	game_over_screen.position = (screen_size - game_over_screen.size) * 0.5
```

```
Godot Engine v4.4.1.stable.mono.official (c) 2007-present Juan Linietsky, Ariel Manzur & Godot Contributors.
--- Debug adapter server started on port 6006 ---
--- GDScript language server started on port 6005 ---
Found a repository at C:/Users/laure/Documents/gun-for-all/.git/.
Selected repository path: C:/Users/laure/Documents/gun-for-all/.git/.
● ERROR: res://Scripts/main.gd:193 - Parse Error: Expected closing ")" after call arguments.
● ERROR: modules/gdscript/gdscript.cpp:3022 - Failed to load script "res://Scripts/main.gd" with error "Parse error".
```

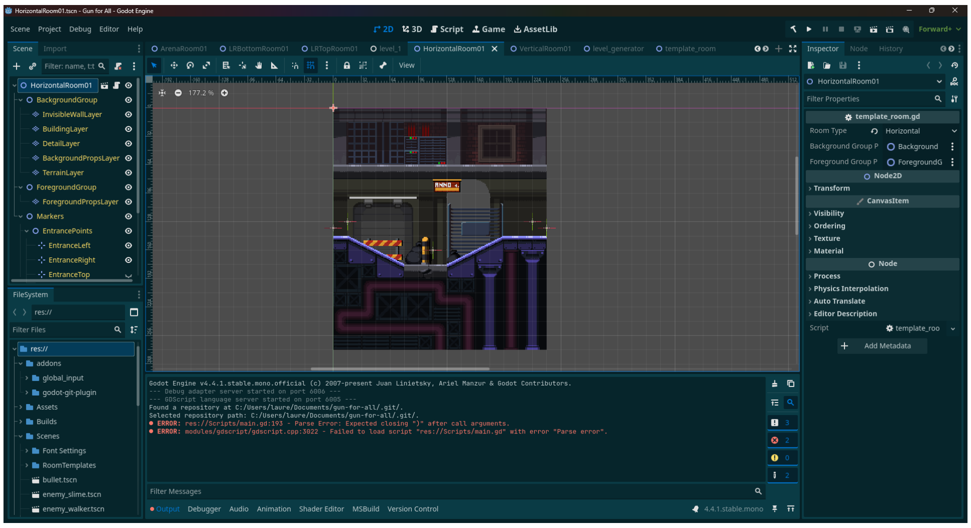
```
Window
Mode                 Windowed
Initial Position     Absolute
Position             x 0 px   y 0 px
Size                 x 400 px   y 300 px
Mouse Passthrough    PackedVector2Array
Viewport
Disable 3D           On
Use XR               On
Own World 3D         On
World 3D             <empty>
Transparent BG       On
Handle Input Locally ✓ On
Snap 2D Transforms   On
```

Figure 6: Screenshot of the Godot editor

Godot is a free, an open-source, cross-platform game development environment with a built in code editor and its own high level scripting language called GDScript. There were several reasons why Godot was the obvious pick for development environment:

1. The editor is very lightweight and responsive which is great when making frequent changes
2. It has powerful 2D tools, which are ideal for this project
3. Godot's architecture uses a scene and node based design which makes creating object oriented projects very intuitive
4. It natively supports managing system windows which very few other engines offer
5. It is open source and easy to customise and add to

4.2 Procedural Generation



```
Godot Engine v4.4.1.stable.mono.official (c) 2007-present Juan Linietsky, Ariel Manzur & Godot Contributors.
--- Debug adapter server started on port 6006 ---
--- GDScript language server started on port 6005 ---
Found a repository at C:/Users/laure/Documents/gun-for-all/.git/.
Selected repository path: C:/Users/laure/Documents/gun-for-all/.git/.
● ERROR: res://Scripts/main.gd:193 - Parse Error: Expected closing ")" after call arguments.
● ERROR: modules/gdscript/gdscript.cpp:3022 - Failed to load script "res://Scripts/main.gd" with error "Parse error".
```

Figure 7: Screenshot of the Godot editor depicting a template room

As described in chapter Three, *Gun for All* uses a Modular Room Based Procedural Generation algorithm these were built off of room templates like the one in the figure above. Six types were created: Horizontal rooms, Vertical rooms , Left Right Bottom (LRB) rooms, Left Right Top (LRT) rooms, Arena rooms and Filler rooms. The practical difference between each type of room is where the exits are place in each room, most of the rooms are self explanatory, with an Arena room having all 4 exits open and the Filler room having no exits open.

Listing 1: exerpt from generate_level

```
# Main path generation
var current_pos = Vector2i(randi() % grid_width, 0)
visited[current_pos] = true
main_path[current_pos] = true
path.append(current_pos)

var last_move_down = false

while true:
    var possible_moves = []

    if current_pos.x > 0 and not visited.has(current_pos +
       Vector2i(-1, 0)):
        possible_moves.append(Vector2i(-1, 0))
    if current_pos.x < grid_width - 1 and not visited.has(
       current_pos + Vector2i(1, 0)):
        possible_moves.append(Vector2i(1, 0))
    if current_pos.y < grid_height - 1 and not visited.has(
       current_pos + Vector2i(0, 1)):
        possible_moves.append(Vector2i(0, 1))

    if possible_moves.is_empty():
        break

    var move = possible_moves.pick_random()
    var next_pos = current_pos + move

    if move == Vector2i(0, 1):
        grid[current_pos.y][current_pos.x] = RoomType.VERTICAL if
            last_move_down else RoomType.LRB
        last_move_down = true
    else:
        grid[current_pos.y][current_pos.x] = RoomType.LRT if
           last_move_down else RoomType.HORIZONTAL
        last_move_down = false

    current_pos = next_pos
    visited[current_pos] = true
    main_path[current_pos] = true
    path.append(current_pos)

# Final room
if grid[current_pos.y][current_pos.x] == null:
    grid[current_pos.y][current_pos.x] = RoomType.ARENA
```

When generating a level, the algorithm first initialises the grid and picks a random cell in the top row of the grid to be the first room in the grid. Next, the algorithm finds all the valid moves that can be taken from that point in the grid, (note that up is never a valid move) adds their respective vector to an array of the possible moves and then picks one at random and calculates where the next cell will be.

After having moved to the next cell, the algorithm will then set the room type of the room the algorithm travelled from based on the direction they came from e.g. traversing the grid to the left creates a horizontal room, or an LRB room coming from above. After this, the algorithm adds the cell's coordinates to the main path array and repeats until a cell is reached with no valid directions. This will always be one of the corners of the bottom row. That room is then set to be an arena.

Listing 2: exerpt from try_branch_from

```
for i in range(branch_length):
    directions.clear()

    if is_valid_position(Vector2i(pos.x,pos.y+1)):
        if grid[pos.y+1][pos.x] == null and pos.y < grid_height -
            2:
            directions.append(Vector2i(0, 1))
    if is_valid_position(Vector2i(pos.x+1,pos.y)):
        if grid[pos.y][pos.x+1] == null:
            directions.append(Vector2i(1, 0))
    if is_valid_position(Vector2i(pos.x-1,pos.y)):
        if grid[pos.y][pos.x-1] == null:
            directions.append(Vector2i(-1, 0))
    if directions.is_empty():
        if !first_step:
            grid[pos.y][pos.x] = RoomType.ARENA
        break

    prev_dir = dir
    dir = directions.pick_random()
    var next_pos = pos + dir


    # Check validity
    if is_valid_position(next_pos) and not visited.has(next_pos):
        if first_step:
            branch_starts[next_pos] = true
            first_step = false

        # Mark the room type depending on move
        if dir == Vector2i(0, 1):
            if grid[pos.y][pos.x] == RoomType.HORIZONTAL:
                grid[pos.y][pos.x] = RoomType.LRB
            elif grid[pos.y][pos.x] == RoomType.LRT:
                grid[pos.y][pos.x] = RoomType.ARENA
            elif grid[pos.y][pos.x] == null:
                if (prev_dir != Vector2i(1, 0)) and (prev_dir !=
                   Vector2i(-1, 0)):
                    grid[pos.y][pos.x] = RoomType.VERTICAL
                else:
                    grid[pos.y][pos.x] = RoomType.ARENA
        elif dir == Vector2i(1,0) or dir == Vector2i(-1,0):
            if grid[pos.y][pos.x] == RoomType.VERTICAL:
                grid[pos.y][pos.x] = RoomType.ARENA
            elif grid[pos.y][pos.x] == null:
                if prev_dir != Vector2i(0, 1):
                    grid[pos.y][pos.x] = RoomType.HORIZONTAL
                else:
                    grid[pos.y][pos.x] = RoomType.LRT

        visited[pos] = true
        pos = next_pos
    else:
        break
```

The next most important part after creating the main path to the boss, is to create several optional paths to make the game more engaging. And then to fill the rest of the empty space will filler rooms.

The algorithm does this through the try_branch_from function which has a chance to generate a branch after every main path room is decided. In this function a branch of variable length is created, in the current version of the game it is 3-5 rooms. To create the branch the algorithm first finds out all the possible moves by taking the 3 possible movement vectors like in generate_level but making sure the room wasn't already set on the main path, or on the bottom row. This will stop it interfering with the main path.

The algorithm then sets those rooms to specific types based on the room type of the room it traversed from, much like generate_level. After running out of valid rooms to traverse to, the algorithm ends the branch with an arena.

4.3 Window System Implementation

The window system, despite being the most important part of the game has many layers to its implementation and affects most parts of the game. In essence, the system works by creating a large ”invisible” main window the size of the the screen the game is on. This will be where the level is generated to and where the players will be instantiated to. Multiple sub-windows are then generated and set to display the level generated by the main window. This means that when moving a sub-window, it displays a segment of the level relative to where it would be if the main window was visible.

Listing 3: exerpt from the window script

```
var main_window: Window = Globals.get_main_window()
var windowType: int = 0  # 0 = follow parent, 1 = stationary

func _ready() -> void:
        main_window = Globals.get_main_window()

        transient = true  # Make this window considered as child
           of main

        _Camera.anchor_mode = Camera2D.ANCHOR_MODE_FIXED_TOP_LEFT

        _initialise_background()
```

Listing 4: exerpt from the window script

```
func follow_parent() -> void:
        if parent:
                var target_position = parent.global_position -
                   Vector2(size.x, size.y) / 2
                position = target_position
```

The excerpt is a excerpt from the script used to generate sub-windows. As this window needs to be instantiated for the player or enemy characters, it needs to able to reference the main window, so it uses the Globals singleton to find the reference to the main window. The window also has 2 different types which is decided at runtime, with the follow_parent behaviour being unique to type 0.

4.4 Player & Enemy Implementation

Listing 5: exerpt from player_main

```
    func change_state(new_state_name: String):
        if current_state:
                current_state.exit_state()
        current_state = get_node(new_state_name)
        if current_state:
                current_state.enter_state(self)
```

Listing 6: exerpt from jumping_state

```
extends char_state

func enter_state(char_node):
        super(char_node)
        char.player_sprite.play("jumping")
        char.velocity.y = char.JUMP_VELOCITY
        char.coyote_timer = 0.0
        char.jump_buffer_timer = 0.0

func handle_input(delta):
        # Variable Jump Height
        if char.velocity.y < 0:
                char.player_sprite.frame = 0  # Going up
        elif char.velocity.y > 0:
                char.player_sprite.frame = 1  # Falling down

        if not Input.is_action_pressed("ui_accept"):
                if char.velocity.y < 0:
                        # Released early -> cut jump height
                        char.velocity.y *= 0.5
                elif char.velocity.y > -20 and char.velocity.y <
                   20:
                        # Released late near peak -> force
                           downward
                        char.velocity.y = 200  # Start falling
                           fast


        if char.is_on_floor():
                char.change_state("IdleState")
        elif Input.get_axis("ui_left", "ui_right") != 0:
                char.change_state("MovingState")
```

The state machine for all characters works in a similar way, it consists of 2 or more state classes which are attached that can be switched between through a function in the main script for that character. Each state, like jumping_state has unique behaviour for example setting the sprite, so that the characters can be animated, or handling the logic that lets players jump.

Listing 7: exerpt from player_main

```
_PlayerWindow = preWindow.instantiate()

# Add it to the scene before modifying its properties
add_child(_PlayerWindow)
```

```
_PlayerWindow.set_parent(self)

# Wait for the node to be added before modifying world_2d
await get_tree().process_frame

player_window_id = _PlayerWindow.get_window_id()

#window setup
#sharing the same world as subwindow
_PlayerWindow.world_2d = _MainWindow.world_2d

# Pass the main window reference to the subwindow
_PlayerWindow.windowType = 0

# Close main window if this is closed
_PlayerWindow.close_requested.connect(_on_close_requested)

# Keep player window in focus
_PlayerWindow.unresizable = true                # Prevent
   resizing the window
if player_window_id != -1:
    DisplayServer.window_set_transient(player_window_id, -1)  #
       Remove transient status
    DisplayServer.window_set_flag(DisplayServer.
       WINDOW_FLAG_ALWAYS_ON_TOP, true, player_window_id)
```

Both the player and enemy_walker (Boss) classes open by instantiating the window class we looked at previously and adding it as a child thus creating a sub-window that will follow the character around. For quality of life, the window is set to always be on top, so that even if the player opens another program they won't lose track of the game.

_main

```
    func shoot() -> void:
        if not player:
                return

        var bullet = preBullet.instantiate()
        bullet.shooter = self
        bullet.global_position = muzzle.global_position
        bullet.direction = (player.global_position -
           global_position).normalized()
        get_parent().add_child(bullet)
```

```
    var result = get_world_2d().direct_space_state.intersect_ray(
       query)

        if result:
                var collider = result.collider
                if collider != shooter and (collider.is_in_group
                   ("player") or collider.is_in_group("enemy")):
                        if collider.has_method("take_damage"):
```

```
                                    collider.take_damage(damage)
                  hit()
```

Both player and enemy_walker have almost identical versions of this function, the only difference is how it is called. It instantiates a bullet and fires it in a specific direction. That bullet, then travels until it hits something and calls its take_damage function if it is either a player or an enemy.

When the boss dies, a signal is emitted to tell the main window to end the round and create a results screen that will let the player choose upgrades and continue on to the next level. However when a player dies, a signal is emitted to tell the main window that the player has lost the level and to create the game over screen that would allow them to retry.

4.5 Enemy Nagivation

The main way Enemies differ from players is the way they navigate the level. Unlike the player, who will navigate the level using keyboard controls, the enemies must use AI to navigate each room. Both the enemy_walker (Boss) and the enemy_slime use a Navigator node which is connected to their state machine that will help them navigate the level.

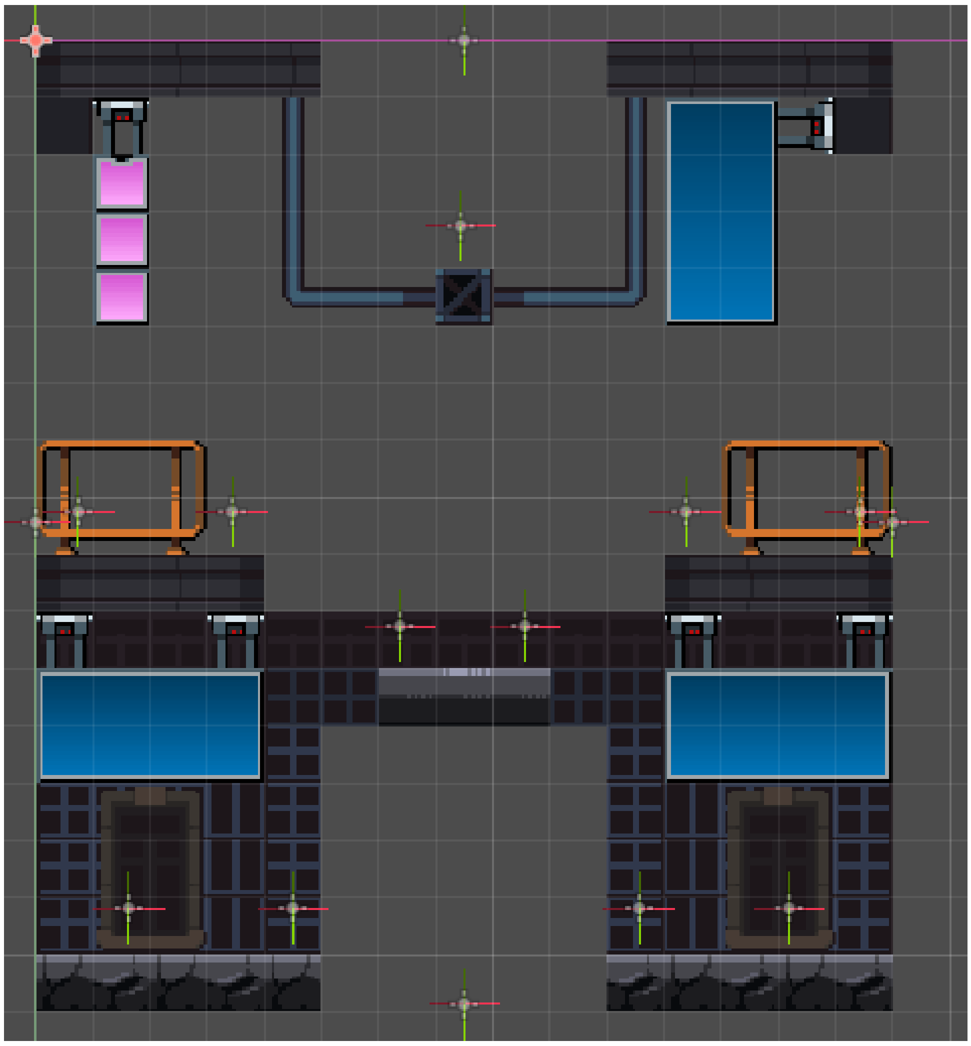


Figure 8: Screenshot of the Arena room depicting NavPoints

In the figure above, are an array of NavPoints placed strategically throughout every room, they represent points where the enemies can travel.

```
extends Marker2D
class_name NavigationPoint

# Store neighbors: each is a dictionary with the neighbor node
   and a "type" (walk, jump, fall)
var neighbors: Array = []

# Called automatically by the level generator's
   auto_connect_navpoints
func add_neighbor(other_navpoint: NavigationPoint, move_type:
   String) -> void:
        neighbors.append({
                "target": other_navpoint,
                "type": move_type
        })

# Optional: Find nearest neighbor of a certain type
```

```
func get_neighbors_by_type(move_type: String) -> Array:
        var filtered = []
        for neighbor in neighbors:
                if neighbor.type == move_type:
                        filtered.append(neighbor.target)
        return filtered
```

NavPoints are represented by a point in space, alongside an array of close NavPoints that are reachable from that point. These are essential to the function of the enemies. Once an enemy is spawned, its navigator will have access to all the NavPoints of the room they are in. It will set its current node to the one it's closest to and pick a destination at random from that point's neighbours. Then the state machine will work as usual, managing the movement behaviour necessary to get it to the destination NavPoint.

If a destination is reached a new one is picked at random. This is repeated endlessly unless a player comes within range, at which point the slime_enemy will chase the player and the enemy_walker will fire at the player.

Chapter 5: **Evaluation**

In this chapter, the project will be evaluated against the objectives and research questions, and any challenges will be discussed

5.1 Player Testing

To evaluate whether the project succeeded in the objectives, I had 7 people test the game and complete a short questionnaire about it. The feedback was mixed.

The testers enjoyed the keyboard controls and the upgrade system. They also thought the procedurally generated levels were mostly engaging, however, very few players liked the health system carrying on into the next round due to its difficulty.

However, most importantly, most players' favourite aspect of the game was the multi-window mechanic as it added to the fun and interest of the game.

5.2 Discussion

Throughout the project the main challenge faced was that of time. The many different complex systems such as multi-window functionality and procedural generation were difficult to create in the time-frame and so left the project with fewer features than I had originally intended, notably: more enemy types that could interact with the windows in more creative ways and new player abilities that would make fights against all enemies more engaging by introducing more ways to approach battles.

It also left the game with more bugs than intended and relatively plain presentation, which may have had an effect on the enjoyability of the game.

Chapter 6: **Conclusion**

This chapter will discuss what I have learned from this project and any future plans

6.1 Discussion & Future Plans

Even considering the drawbacks of the project, the feedback from the player testing leads me to conclude that the project was a success overall, as the novelty of the multi-window mechanics alongside the gameplay style of the other genres created a more fun and engaging experience.

As I plan to continue working on this project, the changes I would make to the project are quite relevant. I would, in future, make sure to prioritise a stable gameplay experience over adding new features as the player encountering bugs that disrupt the gameplay experience is more detrimental than the benefits of a new addition. Other than that, I would allocate more time to an ambitious project like this to ensure that each system can be completed without compromise.

References

Biskup, T. (1994). Ancient domains of mystery (adom) [A roguelike game combining dungeon crawling and role-playing elements.]. https://www.adom.de/home/index.html

Contributors, A. (1990). Angband [A roguelike dungeon exploration game inspired by J.R.R. Tolkien's Middle-earth.]. https://rephial.org/

DevTeam, T. N. (1987). Nethack [A highly influential roguelike game building on Rogue's legacy.]. https://nethack.org/

Harris, J. (2020). Exploring roguelike games. CRC Press.

Linssen, D. (2015). Windowframe [A platformer where the player manipulates the game window itself.]. https://managore.itch.io/windowframe

Peng, X., Balloch, J. C., & Riedl, M. O. (2021). Detecting and adapting to novelty in games. arXiv preprint arXiv:2106.02204.

Perlin, K. (1985). An image synthesizer. ACM SIGGRAPH Computer Graphics, 19(3), 287–296.

Perlin, K. (1999). Noise: Making random look good [Archived lecture on procedural noise and randomness aesthetics.]. https://web.archive.org/web/20071008162217/http://www.noisemachine.com/talk1/4.html

Perlin, K. (2002). Improving noise. Proceedings of the 29th Annual Conference on Computer Graphi
681–682.

Sanders, A. F. (2018). 8.1 what is a window manager. Revival: The Handbook of Software for Engineers and Scientists (1995), 108.

Torcado. (2022). Windowkill [An independent video game featuring system window manipulation as a core mechanic.]. https://torcado.itch.io/windowkill

Toy, M., Wichman, G., & Arnold, K. (1980). Rogue [The original dungeon-crawling roguelike game.]. https://en.wikipedia.org/wiki/Rogue_(video_game)

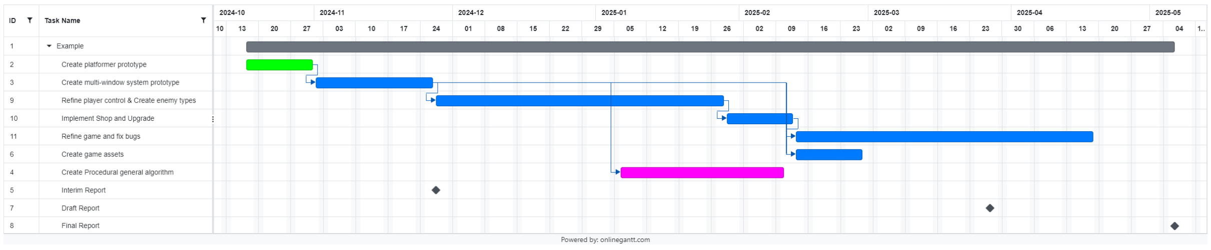
Westecott, E. (2013). Independent game development as craft. Loading... The Journal of the Canadian Game Studies Association, 7(11), 78–91.

Appendix A - Risk Assessment

ID	Risk	Likelihood	Impact	Actions
1	Failure to debug software or correctly assemble hardware	Medium	Low	Make sure to develop a strong design plan and leave enough time to debug afterwards
2	Corruption/ loss of files	Low	High	Ensure all work is frequently backed up using version control and can be accessed when needed
3	Illness	Medium	Medium	Make sure plan is followed to minimise significance of delays and take breaks to maintain health
4	Poor time management	Medium	Medium	Ensure plan is followed and achievable goals and milestones are created so that workload isn't overwhelming
5	Poor implementation of new concepts	Medium	Medium	Thoroughly research and plan the implementation of each new project to reduce mistakes

Table 1: Risk Assessment

Appendix B - Work Plan



ID	Task Name
1	Example
2	Create platformer prototype
3	Create multi-window system prototype
9	Refine player control & Create enemy types
10	Implement Shop and Upgrade
11	Refine game and fix bugs
6	Create game assets
4	Create Procedural general algorithm
5	Interim Report
7	Draft Report
8	Final Report

Figure 9: Gantt Chart depicting project work plan