Panamerican Mathematical Journal
ISSN: 1064-9735
Vol 35 No. 4s (2025)

Developing Epilogue: A 2D Platformer Made on Godot

Sumer Singh[1], Mohsin Ali[2], Chhaya Moghe[3]

[1] Sumer Singh is a research scholar in Department of Computer Application in Medi-Caps University
[2] Mohsin Ali is an assistant professor in Department of Computer Application in Medi-Caps University
[3]Chhaya Moghe is an assistant professor in Department of Computer Application in Medi-Caps University

Article History:

Received: *15-01-2025*

Revised: *22-03-2025*

Accepted: *05-05-2025*

Abstract: Since the inception of video games, two dimensional games have captured the imaginations of players worldwide. From more cartoonish platformers like *Super Mario Bros.* and *Donkey Kong Country*, to even more cinematic platformers like *Another World* and the Apple II *Prince Of Persia* games, brawlers like *Splatterhouse* and *Double Dragon,* and shooters/run-and-gun games like *Contra* and *Metal Slug* have left their mark on thousands of game developers around the world. Though at a layman's glance it may seem that the world has moved onto more three dimensional endeavors started as far back as the pseudo-3D effects of ID Software's seminal classics *Wolfenstein 3D* and *Doom*, the market and gaming interest in two dimensional space, especially in independent game development cycles still remain fresh. With recent successes of *Cuphead* and *Hollow Knight* providing a great template for modernization of said older concepts, and more retro styled games such as *Streets Of Rage 4* and *Shovel Knight* (with a few modern tweaks for better player enhancements) shows that interest in retro pixel art games still remain as fresh as their supposed market hay-days during 3rd and 4th console generations. In the meantime as the independent game development cycle increases, engines and asset packs become more widely available and easy to use. No longer do you need to create a framework from scratch or have to buy a licensed engine from a proprietary retailer, and can instead use an existing framework of engine and its features, and utilize it for your game. That is why using the templates of level and game designs of older successful platformers and shooters – cinematic, cartoonish, brawlers, and otherwise, and mixing them with modern enhancements and features, alongside the framework and features provided by the Godot engine, we attempt to create a two dimensional game with the working title of Epilogue, and mix the concept of a melee combat with the shooting, and mix them under one satisfying loop of gameplay – where instead of one contradicting or being any alternative for each other, they both work in tandem of each other and encourages the usage of one when the other's use has subsided.

Keywords: Game, 2D Platformer, Run & Gun, Godot Game Engine, C#

1 Introduction

While a lot of game development tools and engines are available for an aspiring game developer to make their games on, in recent years, Godot has garnered acclaim and attention in recent years due to its no-nonsense approach to availability on a greater user spectrum. For unlike unity that may require a developer to log into the unity hub and remain constantly connected and approved to even properly work on their projects, or the unreal engine that can prove taxing on the lower end pc, Godot engine provides offline work environment and is comparatively less taxing on lower-end pc's, making development easier for developers on lower-end spectrum of internet bandwidth and computer specs. That is why choosing said engine for this game Epilogue felt like natural choice, especially in regards to the engine's strong and easy to understand 2D framework, that supports in-engine level creation and editing. [1]

Coming onto the development of the game itself (which is going to be the main crux of the rest of this research paper), Epilogue, we try to understand the conception of this idea, and the implementation of the sum of its components that may eventually lead to a playable application. While taking several inspirations from classics such as *Contra, Splatterhouse,* and *Ninja Gaiden,* while also some modern inspirations in its conceptual design such as *Hollow Knight* and *Scorn,* Epilogue's core design is based on single principal – the circular repetition between melee and gunplay. Now, for the longest time in game design, melee and gunplay have been two different conceptual beings. Sure there are plenty of games where the player can shoot and the player can melee, but said two concepts always feel like alternative pathways to achieving the same goal, instead of being on singular thread to an achievement. In simple combat terms, a player can either punch the enemy or shoot the enemy, but both these concepts don't tangentially lead to the other's conception outside of just being separate ways to hurt enemies. With Epilogue, the idea is simple – if the player melee an enemy, then it can execute them and take out the gun from their corpses, and once said gun has run out of bullets it cannot be reloaded, only dropped, hence forcing back the player onto melee again.

The idea behind said concept is to force the player to adapt their approach to combat according to given gameplay scenario they find themselves in – which in turn may mirror the scenario a character may find themselves in. With melee combat the player will find themselves in a more reserved combat approach, with slow response time to melee attacks and more risk of getting hit back by enemy, meanwhile, when a gun is received through successful melee in means, it may create a more liberal approach to the combat, with the player now more willing to take combat risks due to the weapon at hand allowing combat from a greater distance.

Mixing said combat with light platforming, a narrative to bind events and stages, and four control schemes (with addition to player customization) approaches to adaptability for diverse sectors of players, this project aims to create and show its core concept while producing a final result that plays smoothly and provides an enjoyable gaming experience – alongside pretty art style, atmospheric music, and a binding narrative.

2 Research Method And Inspirations

2.1 Research Method

The waterfall model has been utilized in great many game development and analysis because of its systematic and structured development process for any proposed system, yielding a greater understanding and framework for a game development lifecycle. [2]

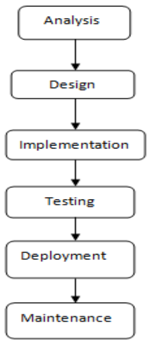


Fig.1. Representation of Waterfall Model

The analysis phase of the waterfall model was used in this game's development to analyze the scope and rhythm of the game. The very conclusion reached for this project was to restrict it to a full scale goal of twelve stages, with each stage book-ended in a boss fight to not only signal to the player that the stage has ended, but also to, in player experience front, prove as the testing ground for the for the player's learned skills and abilities throughout said particular stage. After said scope was analyzed, other requirements such as the engine needed, the roles needed, and the main form of communications between possible team members.

The next phase of waterfall model utilized was the design phase, where rough sketches and blueprints of potential ideas were made. These included creation of level linings, creation of algorithms and flowcharts for movements utilizing available open source assets, and concept art. Then in the implementation phase, the game's assets were made, the mechanics related to movements were implemented, and a functional build was created. In testing phase is where we test said functional builds (whether in relation to the working of the game as a whole or its individual features in separate builds) and detect and fix any potential issues that were noticed. In deployment stages the final build of the game is to be released to potential customer base, and after feedback from said customer base, in the maintenance phase, the game is to be further fixed or re-structured depending on the appropriate feedback.

2.2 Inspirations

In its mechanics the game takes heavy inspirations from run and gun style of games like the *Contra* series, with certain narrative elements taken from the *Planet Of The Apes* movies and art style having varying inspirations ranging from the works of the artists *H.R. Geiger* and *Zdzisław Beksiński* to art style of games such as *Super Castlevania IV, Splatterhouse,* and *Darkseed.*

Alongside serving another additional source of inspiration for the art style, the 2022 horror game *Scorn* also serves a bit of inspiration for the game's one of the more unique mechanics, i.e., creating weapons out of dead body parts. Whereas *Scorn* uses said mechanic for the purposes of exploration and basic level combat, while our purpose of taking said concept is to attempt to bridge the gap between melee combat and shooting combat, and tie both sectors of the gameplay into a complimentary loop of one another that leads to one play style, instead of both attempts at gameplay being alternate forms of play styles.

3 Game In Concept

3.1 Narrative

The narrative of the game takes us through the journey of the main character simply named *Hestmor*, as she attempts to find the mysterious source of a song she starts hearing. While simplistic in its nature, other elements like the nature of other creatures around including their individualistic politics, survival tactics, and the ruling tactics of the final boss of the game (named *Zagara*) will hopefully keep the player engaged through a narrative satisfaction. As while the core gameplay may satisfy the player, an engaging narrative that doesn't interfere too much into the flow of the gameplay but still provide additional context to the player's actions can enhance the player experience [3].

The approach we are taking here, is for the cutscenes to appear at a particular contextual point in the player's journey, where the story element may feel like a reward of completing through a certain challenging obstacle than break from otherwise enjoyable loop.

All the narrative triggers are to be performed in-game in order to avoid artistic inconsistencies, and to improve immersion.

3.2 Level Design

The level design of the game follows a simple 2-Dimensional traversal with ledges for the player character to grab, obstacles in between like spike and falling platforms, and spaces left for other enemies.

For starters we created simple outlines of the levels on a page to better mark out the plans we have for the player progression, and later (while completely not adapting to the linings 1:1) we used them as a context for the actual level creation on the Godot Engine.

The level creations in the Godot engine works on an instance of drag and dropping the assets into a given framework. Said framework can be 2-dimensional or 3-dimensional in nature, and depending on that the parameters and flexibility of said implementations may differ. For our game we chose a 2-dimensional framework, therefore all assets are to be implemented to 2 dimensionally. The engine therefore calculates their implementations in the x-y axis, and adjustment of sprite sizes in relation to the framework is also done in said measurements. For our current implementation of background sprites, we have assigned each tile from the tileset the size of 36 (x-axis) and 36 (y-axis).

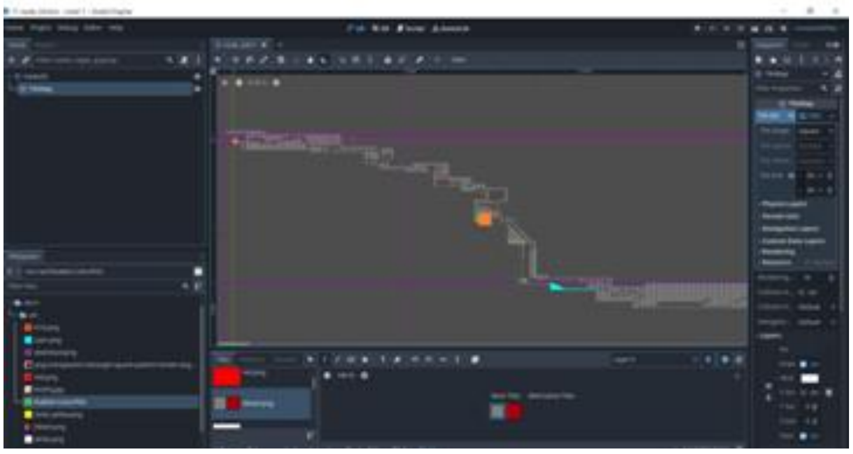


Fig.2. A screenshot of blockings done for Stage 1

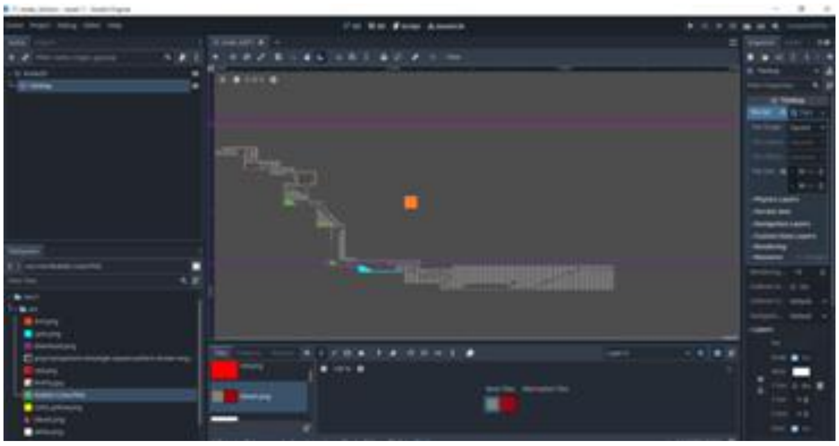
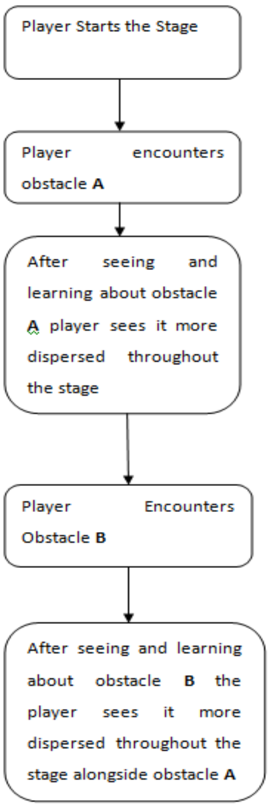


Fig.3. A screenshot of blockings done for Stage 2

The basic level design structure works in terms of introducing obstacles like spikes and enemies one at a time to ensure fair balancing and ease of understanding for the player in relation to workings of individual obstacles. [4] There when we start implementing the level, they follow a rhythm something similar to this:



Player Starts the Stage

Player encounters obstacle **A**

After seeing and learning about obstacle **A** player sees it more dispersed throughout the stage

Player Encounters Obstacle **B**

After seeing and learning about obstacle **B** the player sees it more dispersed throughout the stage alongside obstacle **A**

Fig.4. Representation of the planned player progression design and philosophy

For level designing, the last thing to keep in mind would be to take programming implementations like coyote time in context to designing and defining the distances of the platforms in which the player will traverse. [5] That way distancing in context to the player position may not create an uneven idea of player movement and create misleading sense of early or late input from player end. [6] [7]

3.3 Objectives

Regular Objectives

The regular objectives of the game to kill the enemies you encounter, avoid/dodge obstacles and pitfalls, and reach the end of a stage. Each stage would be bookended by a boss fight, each of whom would have unique weakness that would need to be exploited by the player. Shooting and killing is the name of the game. So, each enemy is killable until the narrative requires so.

Boss Fights

Certain boss in certain stages may appear before hand to scare the player into thinking that they have no chance of defeating it, as no amount of bullets or swiping will hurt them, and so for brief periods in those stages, it would become the gameplay of running away from the boss, while killing any enemies who may pose a threat to the player in their path. Moments of relief would be provided in said stages, where only regular enemies may arrives, and at the end of said significant stages, you may find an environmental object to manipulate that would probably weaken the boss (like breaking their shell to expose their damageable areas, kicking off the proper boss fight.) However, most bosses will have their own arena and will be killable via any number of weaknesses that would be visually hinted at. (An example of this can be a boss's lower body parts being covered in armor, but the head being exposed indicates to the player that the head of the boss is damageable.) Every boss fight, to a certain extent, bookmarks the end of a stage with utilization of an autosave or checkpoint.

Secret Rooms (New Game + Mode)

This exploration to certain parts of the stages will be saved for New Game+ that unlocks after the player has beaten the main game on regular difficulty. Certain areas that were previously locked would now be unlockable, and most of these areas will provide story related context and details of the events of narratives through either collectibles like notes or through visual indication of the arrangements of art assets themselves, for example, using the sprite of a dead body of one character in a particular spot to indicate to the player the nature of said death and its significance to the area at hand.

The introduction of these secret rooms in a separate difficulty level is to challenge the idea of the traditional perception regarding video game difficulties which may usually boil down to increasing of values on enemy damage end or to provide additional enemy numbers. [8]

Bonus Stages (New Game+ Mode)

The Bonus stages will have the same loop and objectives as the main game but will provide additional story events, areas to explore, and newer form of obstacles to overcome.

3.4 Mechanics

Movement

Due to the 2-dimensional nature of the game, the player will be moving in either left or right directions. In addition to movements such as walking left and right, the player will also be able to run, jump, grab, and slide, to clear certain obstacles over a press of a button (usually in combination of already existing walking keys or in context to environment detections). The contextual movements of the slide and jump may actually vary on the basis of the velocity of the speed executed before them. For example, if the player is standing still before pressing the jump button then the player character will just simply jump in a vertical motion upwards and come back. However if the player is already walk before the press of the button then they may jump forwards with an actual trajectory. Similarly if the player is running before pressing the jump button, the velocity will allow player to jump much farther away in distance.

Combat

The game utilizes the core concept of stringing together the melee combat and shooting combat in one loop that makes to two core concepts reflective of each other rather than an alternation of one another. In that the melee combat leads to shooting combat and vice versa. Due to the game having no mechanics related to the gun reloads, the loop goes as follows:

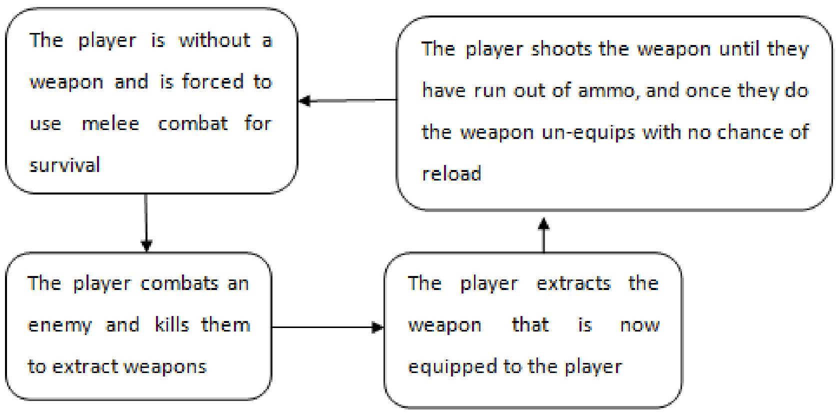

The player is without a weapon and is forced to use melee combat for survival
The player shoots the weapon until they have run out of ammo, and once they do the weapon un-equips with no chance of reload
The player combats an enemy and kills them to extract weapons
The player extracts the weapon that is now equipped to the player

Fig.5. Representation of Combat-Loop

To make sure the player is equipped with a gun and doesn't remain gun-less throughout the game or feel like they haven't been educated regarding the mechanics of shooting, the primary button used to execute the melee attack is also assigned to the one of the buttons used to extract the weapons from the defeated enemies.

This works in the manner that the enemies don't automatically die upon reaching a 0 health-point, instead they reach a temporary stun state from where the player can execute them to extract the gun from them. In that temporary stun state, the players get the choice of a fast kill (that yields low ammo but the extraction of weapon from the enemy is faster) or a slow kill (that yields more ammo but the extraction of weapon from the enemy is slower, leaving the player vulnerable for longer). If the player is unable to kill the enemy in said period of time, then the enemy will go in a state referred internally as "kamikaze mode" where the enemy will start combating with the player again with added buffs to the amount of damage and the speed of damage they may do to the player. The player will need to execute the enemy in order to stop being attacked themselves, leading to said extraction of the gun and allowing player to be in said combat loop without the need for overtly written instructions to do so.

4 Result And Discussion

4.1 Controls

To test the game controls, a QWERTY keyboard was used in terms of PC control testing and a PlayStation controller was used for the testing of Controller controls. However, with certain control logic defined in the code, the game is also controllable through an Xbox controller, even if the control mapping of said device is not used for the following tables shown below. For the retro control mapping for controllers, a PS1 controller was taken into context for the buttons.

Table 1. Representation of Movement Controls

Actions	Retro (PC)	Modern (PC)	Retro (Controller)	Modern (Controller)
Walk Right	Right arrow key	D	D-pad Right	Tilting Left Analog stick to right
Walk Left	Left arrow key	A	D-pad Left	Tilting Left Analog stick to left
Run	Shift	Shift	X	L1
Jump	F	Space Bar	Triangle	X
Slide	Space Bar	F	Circle	Circle
Crouch (when weapon is not equipped)	Down Arrow Key	S	D-pad Down	Titling Left Analog stick down
Squat (when weapon is equipped)	Down Arrow Key	S	D-pad Down	Tilting Left Analog stick down
Crawl (when weapon is not equipped)	Down Arrow Key + Directional Key	S + Directional Key	D-pad Down + Directional Button	Tilting Left Analog stick down and leaning on the direction to go

Table 2. Representation of Combat Controls

Actions	Retro (PC)	Modern (PC)	Retro (Controller)	Modern (Controller)
Melee (Swipe attack)	Left Ctrl	Mouse Button 1	Square	R1
Slow Glory Kill (when near an enemy)	Left Ctrl	Mouse Button 1	Square	R1
Fast Glory Kill (when near an enemy)	X	Mouse Button 2	X	L1
Pick Up/Drop Gun	X	Mouse Click 2	D-pad Up	Triangle
Growling (when no gun is equipped)	X	Mouse Click 2	D-pad Up	Triangle
Shoot Gun	Left Ctrl	Mouse Button 1	Square	R1
Aim Upwards	W	Mouse movement up	L1	Tilting Right Analog stick up
Aim Downwards	S	Mouse movement down	R1	Tilting Right Analog stick down

Aim Left	A	Mouse movement left	L2	Tilting Right Analog stick left
Aim Right	D	Mouse movement right	R2	Tilting Right Analog stick right
Aim Diagonal Up-Left	W+A	Mouse movement diagonal up-left	L1+L2	Tilting Right Analog stick diagonally up-left
Aim Diagonal Up-Right	W+D	Mouse movement diagonal up-right	L1+R2	Tilting Right Analog stick diagonally up-right
Aim Diagonal Down-Left	S+A	Mouse movement diagonal down-left	R1+L2	Tilting Right Analog stick diagonally down-left
Aim Diagonal Down Right	S+D	Mouse movement diagonal down-right	R1+R2	Tilting Right Analog stick diagonally down-right

The controls of the game were chosen after certain testing conducted to test their flow and ease of use on the player end. The options between retro and modern were provided to appeal to both old school style of playing games – that may appeal and be at-ease and familiarity for players who are familiar with the style of game Epilogue is aiming to be – and to new and more modern audience in mind (mainly those who maybe unfamiliar or uncomfortable with older control schemes).

The reason we took our time in allowing said flexibility in control schematics was to test allow the game to be accessible by a larger crowd of player. However, in order to reach said larger crowd, the goal wasn't to eradicate the unique experience we wanted to convey artistically through the game, but to allow said experience to reach the player through their own scenarios and conditions by providing flexibilities that still leads to a unified experience among the players. [9]

The modern controls on the PC front also provide movements and aiming system utilizing joint mouse and keyboard control schemes. While the game is 2-Dimensional, the mouse and keyboard control scheme tries to somewhat simulate the working of how a player may be used to said shooting controls in a 3-Dimensional game. [10]

Said application were then cross-checked in relation to the eye movement behavior in 2-Dimensional games [11] , and then adjusted in regards to their smoothness and motion and the aiming of the player character snapping to one of the core eight directional aiming sectors.

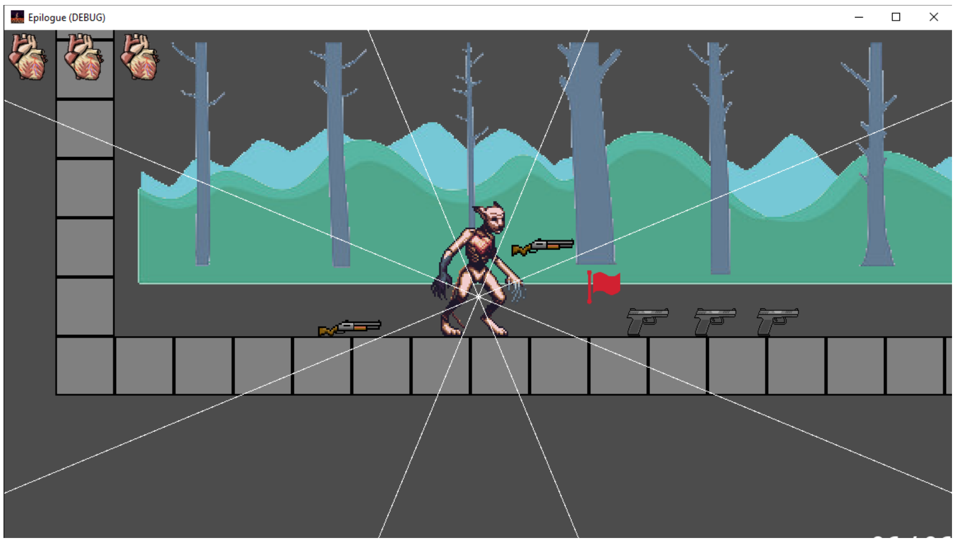


Fig.6. A screenshot of the player sprite holding a gun, with the eight diagonal lines indicating the sectors where the player can aim

4.2 Coding And Testing

Godot has native support for C# and GDScript, a proprietary language heavily based on Python. We chose C# due to its better performance, more features, and also the team's familiarity with it.

We make heavy use of inheritance on the game to avoid duplicate code and to create a better structure. For example, both the player character *Hestmor* and the NPCs (Non-Playable Characters) inherit the same base class, i.e., "Actor".

To be more specific, each NPC inherits the base class "NPC", which in turn inherits "Actor". This means that any code present in the "Actor" class will affect any character, but we still have the ability to write code just for *Hestmor*, the NPCs as a whole, and also individual NPCs.

Perhaps the best feature to manage gameplay is the State Machines for each Actor. With them we can precisely control what actions each Actor can or cannot perform, while also making sure there are no unintended transitions like being able to crouch in midair while jumping or falling.

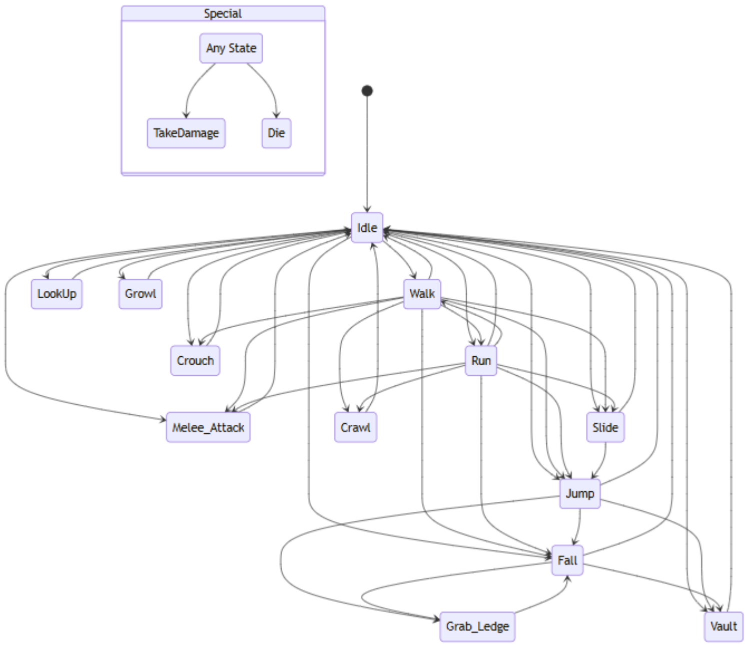


Fig.7. Representation of the player character's state machine

We used Github as the Version Control system, in that each mechanical aspect was individually first tested on individual branches which were separate from the main (or develop) branch and only when the particular mechanical aspect is achieved without any severe issues that they are merged onto the main branch.

4.3 Enemy AI and Spawning

The enemies in the game don't just have strict spawning and attacking patterns, but employee a wide variety of behaviors to increase the immersive experience of the player. At first glance, during the gameplay the enemy behavior may seem typical of what we may presume to be a traditional NPC behavior of simply spawning at the defined spot and attacking the player [12], however, despite said initial presumptions, the NPCs are actually comparatively complex than what the initial behavior may suggest.

The enemies will spawn way before they even appear under the player sight, and will display an ideal behavior that can be observed by player if they are at a certain distance from which they do not trigger the NPC's detection system. Said behaviors can range from them simply interacting with the environmental assets – for example, drinking water or rolling in mud – to interacting with other NPCs – like a primary example as shown below in the screenshots of the AI flowchart of the NPC *Icarasia* drinking nectar from another enemy NPC named *Terra Bischem*, if one so happen to be nearby.

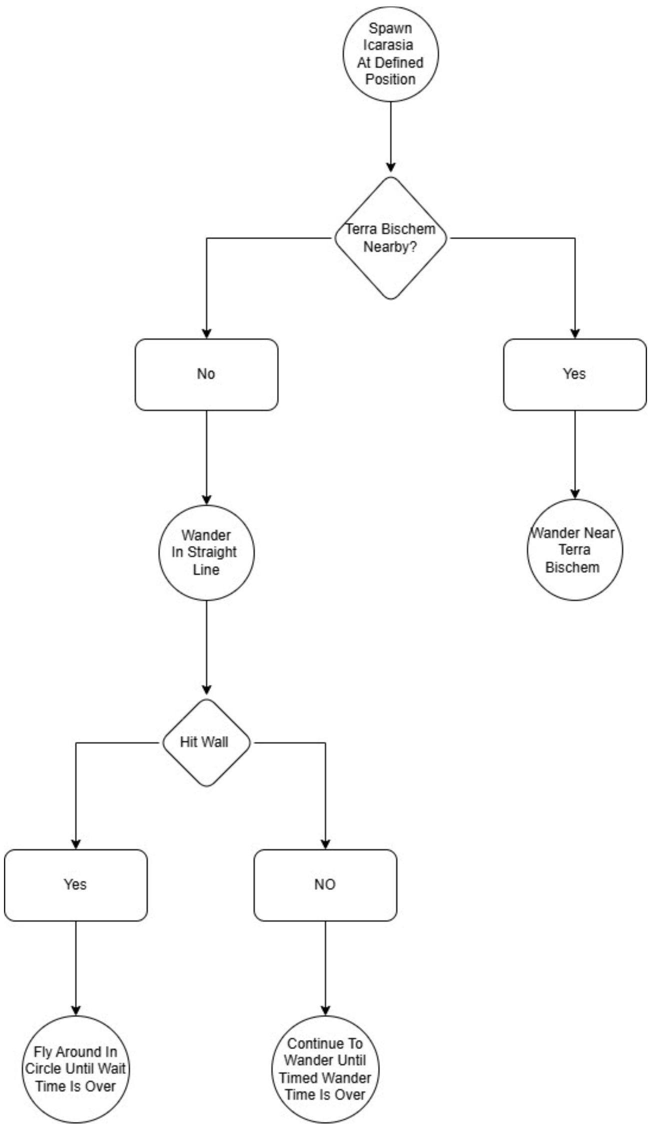

Spawn Icarasia At Defined Position
Terra Bischem Nearby?
No
Yes
Wander In Straight Line
Wander Near Terra Bischem
Hit Wall
Yes
NO
Fly Around In Circle Until Wait Time Is Over
Continue To Wander Until Timed Wander Time Is Over

Fig.7. An example taken from the larger AI Flowchart showcasing a sample of NPC behavior

The enemy behavior doesn't only stop to their idle state, but also extends to how they may act in a combat state. Enemies will react according to player's action – like some of them coming closer or going farther depending on if the player shoots them or not, or what distance the player is to them, and other varying factors like the environment in which the combat is taking place in and if the player is approaching them with a gun or not.

The aforementioned "kamikaze mode" in the combat section, also comes into play here. With different enemies choosing different attack patterns depending on their individually assigned designs and sometimes other factors like the aforementioned parameters of distance, environment and player behavior.

4.4 Response Time And Velocity Values

While working on the game, certain aspects like movement velocity and response time of the inputs were tested and re-iterated upon.

In terms of input detection, we tested the time taken between pressing of a button and animation that was executed. In doing so we found that a lot of animation showcased an input delay on screen not because of any problems regarding coding or detection issues regarding the input devices that the player was using, but in fact due to the length of frames of an animation. While not always the cause for the input delay as on other moments the game required tweaking in terms of its input detection system, the case was common enough that we ended up reducing the frames of certain animation which resulted in faster response time for the player.

Table 3. Comparison of previous and current response time for inputs

Animation	Previous amount of frames	Previous response (in milli-seconds)	Current amount of frames	Current response time (in milli-seconds)
Player Walking	8	9	6	7.5
Melee attack	10	8.6	8	7
Slide	8	9	6	6.5
Jump	10	8	6	6.6

Another thing we recorded and tested heavily were the velocity utilized for player interactions regarding jump and slide, as a way to measure the detection system of the game, and what velocity value is required for certain player actions to help the character have the right amount of feel when being controlled, i.e., neither being too heavy and nor too light.

An example of this could be that when we applied the trigger of the slide animation, no changes were applied to the character's HitBox (i.e. collision detection of the character sprite in context to its environment) but this may change in the future. When the slide ends, the character's horizontal speed is reduced by 50% while the Standing Up animation plays. When the slide starts, the value of the "FloorMaxAngle"(the variable assigned to the gravity response of the player) of the character will be set to 0, meaning it cannot go up slopes, and will move faster when going down slopes. After the slide ends, the value of "FloorMaxAngle" is set to 45, returning the slope interaction to its default state. The speed of the slide is determined based on the player character's previous State, according to the following table:

Table 4. Velocity utilized for Slide

Previous State	Slide Type	Slide Speed (in terms of pixel per second)
Idle	Front Roll	100
Walk	Knee Slide	160
Run	Long Slide	220

During testing these states, it was recorded that each transitional state took up-to 0.3 seconds to complete, except for the Front Roll (executed via the Idle state) that took 0.5 seconds in its entire state to execute, however, to course correct and provide an earlier exit from said animation, it was made so that pressing the upper key during said transition would cancel out the animation and take back the player to the idle state, cancelling out said velocity mid-way to provide an exit from wrongful player input.

Same logics that were utilized for the slide animation were also utilized during the jump. When Triggering the jump animation, the character's horizontal speed is set to 0 as the animation plays. When the first part of the animation (the character crouching to gain height) ends, the second part of the animation (the character leaving the ground) plays, and the character's horizontal speed will be set to a value according to its facing position and jump type, according to the following table:

Table 5. Velocity utilized for Jump

Previous State	Jump Type	Horizontal Speed (in terms of pixel per second)
Idle	Vertical Jump	0
Walk	Low Jump	80
Run	Long Jump	160

Slide	Long Jump	160

The type of jump determines the horizontal speed used. All 3 types use the same vertical speed. Can transition to the following states:

- **Fall**: If the character's vertical speed increases above 0 (meaning the character is no longer rising from the jump, but falling after reaching its peak)
- **Idle**: If the character hits the ground while ascending (for example, by jumping towards a slope)
- **Grab Ledge**: If the player touches a ledge that's between 0 and 30 pixels below *Hestmor's* head
- **Vault**: If the player touches a ledge that's more than 30 pixels below *Hestmor's* head

5 Limitations

Despite having native support for C#, Godot is still made for GDScript (the programming language made exclusively for Godot) first and C# second. The vast majority of functionalities are available in C#, but some more specific stuff simply cannot be ported. For example:

One very common occurrence while writing the code is it needing to set a default value to a variable. If the value is a base type (int, float, string, etc.), this can be done as soon as the variable is declared (for example, int x = 10). However, if you're using the variable to store a complex type (say, for instance, a reference to a Node), then you cannot assign a value while declaring it, you need to use the "_Ready()" method, which runs as soon as the Node the script is attached to finishes loading. This is a such a common scenario that Godot allows you to use a special command to declare a variable and assign a value to it in the "_Ready()" method in a single line: "@onready var x = get_node("example")." The "@onready" annotation will tell the engine to only run this line of code when the script's "_Ready()" method is called. C# has no support for this type of behavior, so we're forced to declare a variable in one line, then override the "_Ready()" method just so we can assign a value to it. While not a major challenge, it would be more ideal to be done in a single line for it'd make the code way cleaner.

Another limitation arrives in form of *Signals*. The way you're meant to work with them with GDScript is to use the "Connect()" method to connect a signal to a method that will be executed when said signal is detected. In C# it's also possible to use the "Connect()" method, but the official documentation recommends using the "+=" operator from C#. This results in exactly the same behavior most of the time, but one huge issue we found is that the "Connect()" method support parameters, while the operator does not, and one of those parameters is a flag telling how the reaction to the signal will occur. For example, one very useful feature is to use the "ConnectFlags.CONNECT_ONE_SHOT" flag to execute a method only once when the signal is detected, then automatically break the connection. But when using the "+=" operator, you cannot define these flags, so you must:

- Create a proper method that will be executed (instead of an anonymous method)

- Connect the signal to this method using the "+=" operator

- After the execution is complete, break the connection using the "-=" operator

If we try to connect a signal to the same method more than once, or break a connection that doesn't exist, Godot prints a bunch of errors to the console, and the game will start working improperly since the execution flow got interrupted.

A non-programming issue comes in form of playtesting. While the game by its nature is of smaller scale when compared to more commercial projects done by corporations and its creation engine is readily available without any cost, the need for playtesting and its scope is almost similar to that of a big commercial project. [13] [14] While gathering playtesting resources by reaching out to public forums and sharing an open build does help alleviate some tension in regards to design flaws and the shortcomings of the executions of mechanics, said public feedback can often be unreliable, for a lot of feedback may miss the point of the core execution and a feedback might be the complete opposite in its suggestions to what we wish to achieve. In other cases, the knowledge or experience of the playtester might be in question especially if they haven't experienced similar games before, or in case of more technical requirements, does not possess the same knowledge. Finding more expert feedback and playtesters in relation to the core demographic of the game's aim might alleviate some worries regarding the final product, but doing that itself may take a longer time and resource for a developer on a lower scale of the game development spectrum.

6 Future Work And Conclusion

The current scope of this game for the sake of presentation is only three levels that are more decidedly planned out with some open source assets utilized to better showcase the core functionalities of the project, and with that base now established on how the core combat and environments of the game works, we can move forward with establishing actual artistic assets and more defined contextual triggers to story and level events. Some hopeful plans indicate for a twelve level complete structure with two bonus stages, with each stage representing its own specific look to distinguish itself artistically, but also introducing certain elements and changes into environmental obstacles and enemies to keep the gameplay fresh and the player on its toes.

With the current build made, we hope to see what improvements could be brought upon the game during the testing and feedback phases, and what aspects of the game can be retooled or reworked to make it more smoothly flowing, and to tweak timings and values of the mechanical response time that may help make the game much snappier but still in relation to its core function loop of a risky but reward melee-shooting loop. All these planned additions, feedbacks, and tweaking can hopefully make the game a much appealing prospect to a potential player, and make the game better overall.

References

[1] Dhule, Maithili. (2022). Beginning Game Development with Godot: Learn to Create and Publish Your First 2D Platform Game. 10.1007/978-1-4842-7455-2.

[2] Akbar, Yuma & Ammaar, Mohammad. (2024). 2D Platformer Game Prototype on Indonesian History Using Scratch. International Journal Software Engineering and Computer Science (IJSECS). 4. 1066-1076. 10.35870/ijsecs.v4i3.3070.

[3] Fanfarelli, Joseph. (2014). The Effects of Narrative and Achievements on Learning in a 2D Platformer Video Game.

[4] BYRNE, E. 2005. *Game Level Design*. Charles River Media.

[5] McDonald, Peter. (2024). Run and Jump: The Meaning of the 2D Platformer. 10.7551/mitpress/14478.001.0001.

[6] Killick, Michael. (2022). 2D Platformer Tutorial. 10.1007/978-1-4842-8789-7_8.

[7] Smith, Gillian & Whitehead, Jim. (2008). A framework for analysis of 2D platformer levels. 10.1145/1401843.1401858.

[8] B., Henry & Mikami, Koji & Kondo, Kunio. (2018). Perception of Difficulty in 2D Platformers Using Graph Grammars. 22. 38-46. 10.20668/adada.22.2_38.

[9] Anderson, Sky. (2024). The Ground Floor Approach to Video Game Accessibility: Identifying Design Features Prioritized by Accessibility Reviews. Games and Culture. 10.1177/15554120231222580.

[10] Warburton, Matthew & Campagnoli, Carlo & Mon-Williams, Mark & Mushtaq, Faisal & Morehead, John. (2023). Kinematic markers of skill in first-person shooter video games. 10.1101/2023.02.27.530169.

[11] Lin, Na & Feng, Lei & Kang, Tian & Fan, Shiyu & Zhang, Yunhong. (2017). An Eye Movement Research on 2D Interactive Game Design. 10.1007/978-3-319-41691-5_17.

[12] Zhou, Jiansong. (2024). Intelligent Agent and NPC Behavior Modeling: From Traditional Methods to AI Driven Interactive Game Design. Applied and Computational Engineering. 112. 85-91. 10.54254/2755-2721/2024.17913.

[13] Denisova, Alena & Bromley, Steve & Mekler, Elisa & Mirza-Babaei, Pejman. (2024). Towards Democratisation of Games User Research: Exploring Playtesting Challenges of Indie Video Game Developers. Proceedings of the ACM on Human-Computer Interaction. 8. 10.1145/3677108.

[14] Linåker, Johan & Bjarnason, Elizabeth & Fagerholm, Fabian. (2024). Pre-Release Experimentation in Indie Game Development: An Interview Survey. 10.48550/arXiv.2411.17183.

Games Cited

[1] *Super Mario Bros*. Nintendo (NES), 1985

[2] *Donkey Kong Country.* Nintendo (SNES), 1994

[3] *Another World*. Delphine Software (Amiga), 1991

[4] *Prince Of Persia*. Broderbund (Apple II), 1989

[5] *Splatterhouse.* Namco (Arcade), 1988

[6] *Double Dragon*. Taito (Arcade), 1987

[7] *Contra*. Konami (Arcade), 1987

[8] *Metal Slug*. SNK (Arcade), 1996

[9] *Wolfenstein 3D*. Apogee Software (MS-DOS), 1992

[10] *Doom*. id Software (MS-DOS), 1993

[11] *Cuphead*. Studio MDHR (Windows), 2017

[12] *Hollow Knight*. Team Cherry (Windows), 2017

[13] *Streets Of Rage 4*. Dotemu (Windows), 2020

[14] *Shovel Knight*. Yatch Club Games (Windows), 2014

[15] *Ninja Gaiden*. Tecmo (NES), 1988

[16] *Scorn*. Kepler Interactive (Xbox Series X/S), 2022

[17] *Super Castlevania IV*. Konami (SNES), 1991

[18] *Darkseed*. Cyberdream (MS-DOS), 1992