UNIVERSITAT JAUME I

VIDEO GAME DESIGN AND DEVELOPMENT DEGREE

Degree Final Project

Design and development of top down 2D action-adventure video game with hack & slash and bullet hell elements

Daniel **Estradera Benedicto**

Supervised by:

María **Barreda Vayà**

June 17, 2019

Abstract

The following document contains the report for the Final Project of the Video Game Design and Development Degree. This project's final result will consist of an original 2D action-adventure video game rendered in pixel art style.

The video game genre is 2D real-time action-adventure inspired by the 8-bit and 16-bit games such as *Legend of Zelda: A link to the past*. The core of the gameplay is based on a challenging arcade combat, framed in a fantasy world. The combat blends elements from the genres **hack and slash** and **bullet hell**. This means that the player will fight with a melee weapon versus enemies that fire an overwhelming number of projectiles. To overcome this challenge, the player will have at his/her disposal two defensive skills: bullet deflection and dodge. After fighting several lesser enemies along its path, the player will finally face the main antagonist. The emphasis of the project lies in this climaxing endgame battle.

The project is going to be developed in the Unity3D engine and will aim to craft an engaging action gameplay with stunning visuals that bring to life the world where the adventure takes place. The target platform is a PC and it is designed to be played with a XBOX controller (or mouse and keyboard as an alternative).

Keywords: Action-adventure, Boss battle, Pixel art, Unity, Bullet hell, Hack & slash.

Index

1 Technical Proposal

1.1 Introduction and Motivation

In an era of ever growing titles and massive triple A sagas, costs in the video game industry are raising exponentially to reach the self imposed standards in each new launch. Doing this while keeping competitive prices that lure new players, requires more ingenious and aggressive tactics. Inevitably, this will reach a stalemate when the gap between salaries and sales are irreconcilable.

While big-budget games rely on established ideas and struggle with changes, indie games (as independent low-budget video games) have the freedom to unleash their creativity, reinvent the wheel and tackle the video game creation as an art instead of a service.

In the near future, artificial intelligence (AI) driven content creation will for sure change the landscape and provide us with cost-efficient mainstream titles. In the meantime, independent games may shine again as players appeal to them in search for fresher or simpler concepts instead of their baroque and over bloated counterparts.

The aim of this project is developing a video game that emulates the style and philosophy behind well-known independent titles that strive to deliver a tight experience without creative limitations. With every aspect of it developed by one person, the scope of the game is small but it leaves space to experimentation and gameplay innovation.

This project is a genuine and personal creation to show the game design skills I have learnt and trained during my degree studies. Nevertheless, the game itself is different to others of the same genre in its design, narrative and art.

In the video game, the player controls a small heroine armed with a sword. She stumbles upon a village that is being assaulted by zealots of a cult obsessed with fire. Despite of being none of her business, she gets ambushed and forced to fight back. After the skirmish, a lone peasant survivor points her to a huge cathedral where the Priest, the leader of the perpetrators, dwells.

This prologue to the main action of the game will serve as a tutorial for the player and will give context and meaning to the encounter with the main villain of the game. The entire game will include 7 levels, the last one being a long combat versus the Supreme Priest of the cult. This combat will take place inside the cathedral and will be divided into 3 phases, each featuring different skills, patterns and animations for the Priest.

1.2 Related Subjects

VJ1215 - ALGORITMOS Y ESTRUCTURAS DE DATOS

Building up on the concepts upon the previous *Programación I* and *Programación II* , this subject teached me how to apply my programming skills to solve problems using efficient code and proper data structures.

VJ1224 - INGENIERÍA DE SOFTWARE

In the same fashion as the previous subject, this one continued developing my programming skills introducing methodologies and principles of software engineering.

VJ1222 - DISEÑO CONCEPTUAL DE VIDEOJUEGOS

The principles used in the definition of mechanics and rules for the game design were introduced in this subject.

VJ1227 - MOTORES DE JUEGOS

This subject served as an introduction to the Unity 3D game engine. This was the software used to craft an put together this whole project.

VJ1231 - INTELIGENCIA ARTIFICIAL

Some of the techniques for artificial intelligence introduced in this subject were implemented in this project to perform the movement, pathfinding and decision making of the enemies.

1.3 Objectives

The main goal of this project is to design, implement and polish the gameplay and art of the described video game to a point where its quality is equated to popular indie video games of similar scope.

In order to achieve the main purpose, the work is divided into several specific objectives which are summarized as follows:

- To design an engaging gameplay that is entertaining and challenging (easy to learn and hard to master). It combines the effortless action packed aspect of the hack'n'slash genre with the frenetic and precise controls that require the bullet hell genre.
- To implement efficient artificial intelligence for the Priest (final boss) and its minions to produce an interesting behaviour. It is designed to pose a real threat, while avoiding erratic and unpredictable actions in order to be understandable by the player.
- To create an original setting and a variety of characters consistent with the world rendered in pixel art style. It requires frame by frame hand drawn art for the animations.

1.4 Project planning

To reach the final goal of the project, we must manage the required workload in advance. Initially we have to set the project's scope, following by a breakdown of the tasks and their estimate costs. The duration of the tasks should be balanced to ensure that, even in the worst case, a minimum viable product is achieved. Roughly 40% of the time will be destined to programming, 30% for the drafting of the Final Memory. The rest is assigned to art, however, drafting and coding are prioritised. This is because this area is not as vital to reach a functional state of the project as the other two. The detailed planning is the following:

- **Game Design Document (15 hours)**
- **Implement the game mechanics and core systems (80 hours)**
    - Implement the player controller with custom collision system.
    - Implement the hitboxes and projectiles as a foundation for the combat.
    - Implement the core mechanics: dash, bullet deflection.
    - Implement the button buffering and combos: lunge, backflip, sprint, critical strike.
- **AI, behaviour and pacing of the encounter's phases (40 hours)**
    - Implement the state machine to control enemy behaviour.
    - Implement the steering behaviours and pathfinding.
    - Design 4 types of lesser enemy, each one with a distinct role.
    - Implement the system to coordinate lesser enemy groups.
    - Design and implement 3 final boss phases with different patterns and skills.
- **Character animations and VFX (50 hours)**
    - Animate heroine and her FX for sword slashes/hits.
    - Animate the Priest, with a different set of animations for each phase.
    - Animate 4 types of lesser enemy.
    - Fire FX: projectiles, explosions, etc.
- **Environmental art and world building (30 hours)**
    - Forest and village background tiles and props (decoration for the background).
    - Cathedral exterior and interior tiles and props.
    - Cathedral seen from the distance.
    - Animated decorations and critters to breathe life into the scene.
- **SFX and music (10 hours)**
    - Boss battle music that changes on each phase.

    - Ambient music for the introductory levels.
    - Specific sound effects for the heroine skills.
    - Specific sound effects for the Priest skills.
    - Sound effects for fire projectiles and explosions.
    - Background sounds.
- **Implement the non-gameplay systems and miscellanea (10 hours)**
    - Implement the menu.
    - Implement the controller bindings for player customization.
    - Cover art for the start of the game.
    - Scripted scenes between gameplay.
- **Final Memory (60 hours)**
- **Project's Defense (5 hours)**

1.5 Expected results

By the end of the project, a fully playable video game is expected to be developed. In this game the player will control a heroine with a range of expressive skills. As the heroine, the player will engage in fights that combine elements of the hack & slash genre with the bullet hell genre. The action will be framed in a beautiful fantasy setting to explore, rendered in pixel art and divided in 7 levels. The first levels will be introductory to the plot and gameplay, leading to a final interior set, where a huge battle takes place. There will be four types of lesser enemies and a powerful boss with three increasingly difficult phases, each one with its own skills, patterns and animations.

1.6 Tools

Game engine - **Unity3D**

Unity 3D is a well established industry standard game engine. It is used widely for its cross-platform support and its accessibility. I chose to work with it because the scripting

API uses C#. While not as fast as C++, it does not overcomplicate the development process and still offers much more control than visual scripting tools offered in other game engines. Although the engine was first conceived for 3D, it now extensively supports 2D rendering, physics and animation.

IDE - **Visual Studio**

Visual Studio is an integrated development environment from Microsoft that can be obtained along with Unity. It is also an industry standard for video game coding.

External software - **Unity’s Libraries**

The scripting API in Unity is used to code the rules for the gameplay and how the game responds to the player input. That includes graphical effects, the physical behaviour of objects and the artificial intelligence of enemies.

Animations - **Aseprite**

Aseprite is a software to create 2D art and animations for games. It specializes over retro style graphics of the 8-bit and 16-bit era, in other words, pixel art. It is really lightweight and its workflow helps to speed up the creation process.

Maps - **Pyxel edit**

Pyxel edit is designed for tile-based art. It offers a better pipeline to create this type of asset, which otherwise would be much more time consuming.

FX - **Pixel Fx Designer**

As the previous two tools, this is an specialized piece of software for pixel art. This one eases the process of creating special effects in pixel art style.

Concept Art - **Photoshop**

Photoshop is the industry standard for digital art as a whole. Although I could use photoshop to create all the art in the game, this powerful software is more suited for conventional art and photo editing.

Music and SFX - **Bosca Ceoil**

It is a free, easy to use tool for creating music. Since most of the time music and sound effects are neglected during the development of a game, I wanted to use an accessible software to simplify this task as much as possible.

1.7 References

The following titles are some of the games where this project will draw inspiration from. They share genre elements and pixel art graphics.

Nuclear Throne [1], top-down 2D shooter roguelike. The gameplay premise of the game is simple but really polished, with a focus on being responsive and engaging. The frenetic action and chaos embraced by the design of the game works because of the robust and tested core mechanics.

Hyper Light Drifter [2], is a top-down 2D action-adventure hack & slash. The aesthetics of the game are carefully crafted to build a complex and rich world. The game does not feature text or dialogue but still manages to convey an intriguing narrative through environmental storytelling. This is done through hints left along the game that suggest the player a sequence of events so they come up with their own interpretations of the story. Aside from the world and art, the action is quite immersive as well.

Enter the Gungeon [3], is a top-down 2D bullet hell roguelike. This game is a good example of an action game that features and combines elements of the bullet hell genre. It showcases really challenging and interesting bullet patterns in the enemies and bosses.

Titan Souls [4], is a top-down 2D action-adventure. The catch of this game is a simple mechanic that in turn has an interesting depth. The player has a single arrow that can be thrown at will, but then it must be retrieved to be used again. However, the arrow can be called back anytime. This opens a wide range of uses that emerge from a minimal concept. It is also relevant that the structure of the game features only challenging boss encounters without any lesser enemies in between.

2 Game Design

2.1 Game Overview

2.1.1 The high concept

This is the fast paced action-adventure game where you slash through hordes of fire obsessed zealots. Dodge and deflect their fire projectiles back at them and fight their Supreme Priest in a frenetic boss battle.

2.1.2 Story overview

You play as a wandering swordfighter named Pira. Along your journey, you stumble upon a small village that is being ravaged by a group of brainwashed cultists. Under the rubble you find a lone survivor. He reveals you where the lair of the cult is located. They occupied the sacred ruins of an abandoned cathedral and only you can kick them out of these lands.

2.1.3 Technological requirements

The game will be developed in Unity for the PC platform. The use of an xbox controller is prefered, as the game is designed with that control scheme in mind. Nonetheless, the use of mouse and keyboard is a valid alternative and it is supported too.

2.1.4 Art style

The art style used will be pixel art. Character animations will be drawn frame by frame and backgrounds will be composed using tiles and decorated with reusable props.
To achieve a true pixel perfect graphics, textures are imported in Unity using point filter mode and without compression.

2.2 Game Structure

2.2.1 Title screen and flowchart

The title screen of the game features a pixel art illustration that shows the player character in heroic pose over a background with the cathedral. The player character and several details in the scene are animated to show the movement caused by soft wind gusts.

The options listed to be selected are: *Play*, *Settings* and *Quit*. Once the player starts a new game for the first time, the *Play* option will be switched to *Resume* and *New Game*. *Resume* continues the game from the last checkpoint reached by the player, while *New Game* restarts any progress made. The flowchart shown in Figure 2.1 describes the structure of all the interconnected scenes.

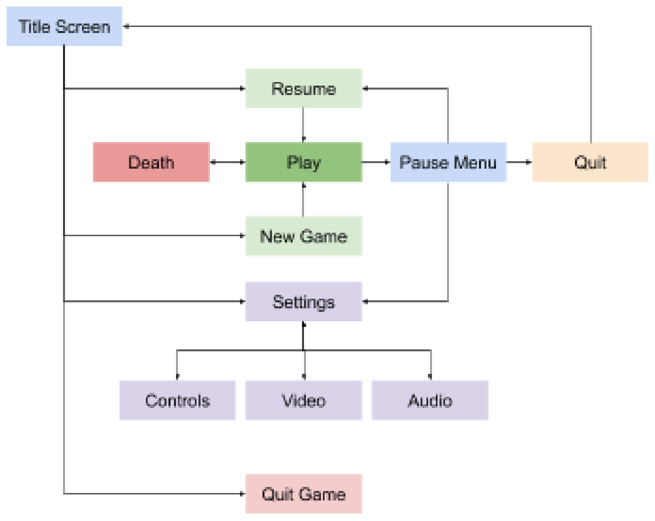


Figure 2.1. Flowchart of the game

2.2.3 Loading screen

The use of a loading screen is mostly cosmetic in the case of this game, because loading times are brief. It will serve as a transition between the Title screen and playing the actual game. It will also appear briefly in between levels, to let the game load the map layout and the enemies.

2.2.4 Game camera

The camera follows the character from an overhead perspective. It can offset towards the aiming direction so the player is able to look ahead. In certain level sections, the camera will be locked inside predefined bounds. This is used in order to hide the limits of the world or even to lock the camera in place to frame a fight encounter you cannot bypass.

It is set as an orthographic view that snaps to the pixel grid at a native resolution of 480x270 pixels (16:9 aspect ratio). Usually, PC screens are four times bigger (1920x1080 pixels), so the game is upscaled four times. Moreover, since the gameplay action happens in a 2D world, there is a problem with how to tackle the depth sorting in the rendering of juxtaposed elements. To solve this, instead of using complex masking or manual layer sorting, the camera is tilted 60º and the all graphics in the game are skewed by a factor of 2 in the Y axis and a factor of 1.155 in the Z axis.

2.2.5 HUD system

The head-up display system for this game is simple and unobtrusive. It displays the player health in a bar of health points in the top left corner and a reticle that shows where player is aiming at.

During the boss battle, a big health bar is displayed at the bottom center of the screen. This bar shows the remaining health points for the current phase of the boss.

2.3 Player character

2.3.1 Description

Pira is a wandering swordfighter, last daughter of a long dynasty of once revered swordsmen. She travels the land fighting the evil, keeping the family oath: feed her spell bound sword with the blood of the twisted and the wicked.

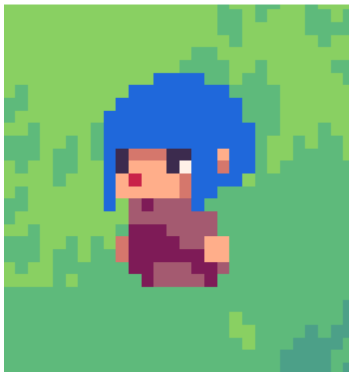


Figure 2.2. Pira, the player character

2.3.2 Controls

Pira has the following set of basic moves that are binded to the controls shown in Figures 2.3 and 2.4:

Move: When using the left analog stick of the XBOX controller or the W-A-S-D keys in the keyboard, Pira runs in that direction.

Roll: Evasive move that displaces Pira in the direction she is running. This action prevents the player from taking any damage. It is useful to dodge incoming attacks and navigate with ease between enemies. It can be triggered with the left bumper in the XBOX controller or the Space key of the keyboard.

Attack: It is the skill with the most complex interactions of the kit. Melee attacks are the primary source of damage, but they require you to get in the middle of the fray, which may lead to a risky situation. On the other hand, this same sword slashes are able to deflect incoming projectiles from afar, offering a strong defense from enemy attacks, yet more unreliable damage output. The right bumper in the XBOX controller or the left click of the mouse are tied to this action.

Aim: Attacks can be aimed at 360 degrees, and parried projectiles get deflected in the direction the player is aiming. The input is taken from the right analog stick of the XBOX controller or the movement of the mouse.

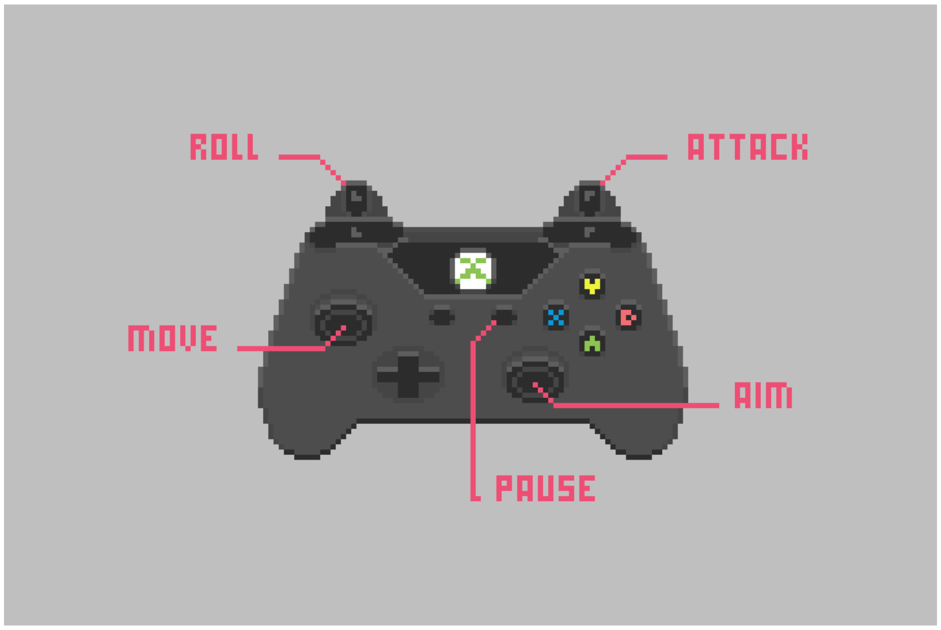


Figure 2.3. Xbox Controller controls

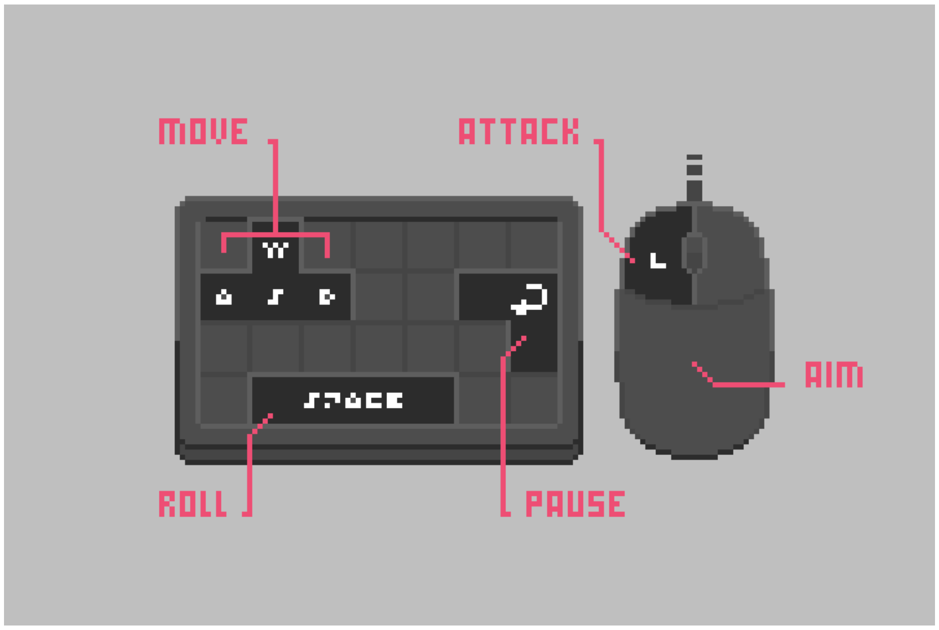


Figure 2.4. Mouse & Keyboard controls

2.3.3 Combat

During combat, the player must juggle between slashes to deal damage and rolling to prevent other sources of damage that can not be deflected. However, keeping a constant motion can be tricky and could sometimes end up in a bad spot. If surrounded by a thick barrage of projectiles, the player can hold down the attack button to unleash a flurry of quick attacks. This special move creates a protective barrier destroying the projectiles. Although it comes with a downside, as the movement of the player character and the aiming become restricted for the duration. While situational, it can be a powerful tool when overwhelmed.

2.3.4 Health

Health is displayed at the top left corner of the screen. Five health points are represented in a bar. Enemy projectiles and attacks that hit Pira take away her health points. When depleted, you die. However, the bar can be refilled by gathering soul fragments. These souls fragments are dropped by enemies that get hitted by special attacks. Getting 4 of them replenishes 1 health point. Fragments are also displayed next to the health bar.

2.4 World

2.4.1 Level Design and World-map

The world is divided in 7 levels with a linear progression. This means that each one connects to the following one, and the game can only be completed by beating each one in succession. The Figure 2.5 shows all the maps interconnected from start to end.

Each level is designed with some principles in mind. The main priority is to keep an interesting pace for the player, mixing segments of intense action with more paused sections. Along the way there will be scattered hints and detail for environment for storytelling to sell the narrative. Finally, the goal is to effectively teach the user how to play the game as it progresses.

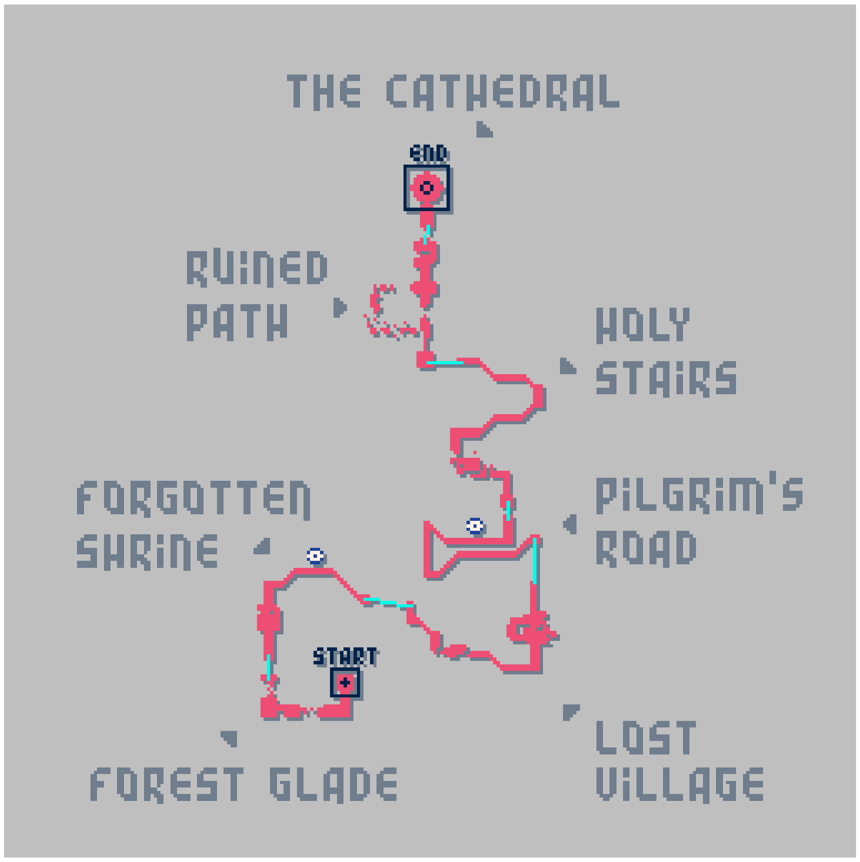
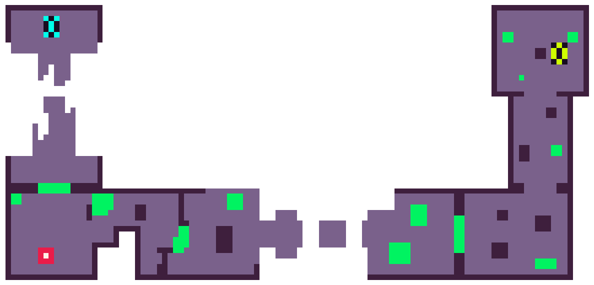


Figure 2.5. World-map featuring all the levels in their context

2.4.3 Level details

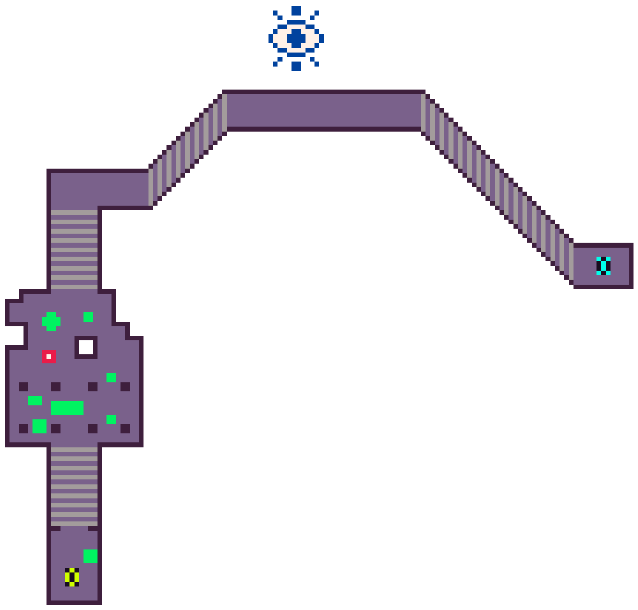
1. Forest glade

Tutorial level where the player can get a grasp of the basics. It showcases destructible obstacles to introduce attacks, platforms over a precipice to teach how to roll and a single enemy.



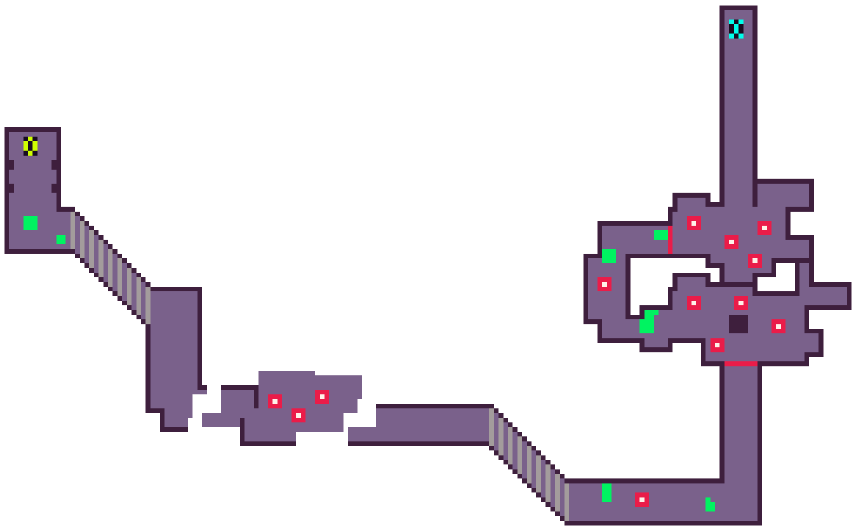
2. Forgotten Shrine

This level starts with a fight versus a tougher enemy to settle down the mechanics of combat. Then, it follows with a serene walk to a viewpoint that overlooks the valley ahead, dominated by a huge structure, the Cathedral. From that far, figures carrying torches are vaguely distinguished marching from the entrance down to the valley.



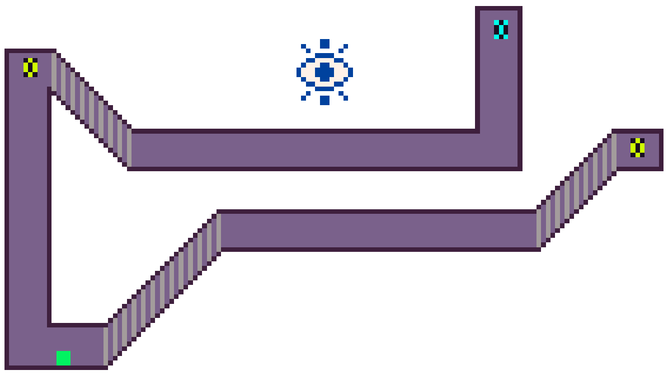
3. Lost village

After a descent towards the valley, the player will get introduced to new types of isolated enemies. This will serve as a preparation for the merciless ambush that awaits ahead. The player reaches a desolate village that was just ravaged by cultists. A considerable group of them is still there, ready for the assault.



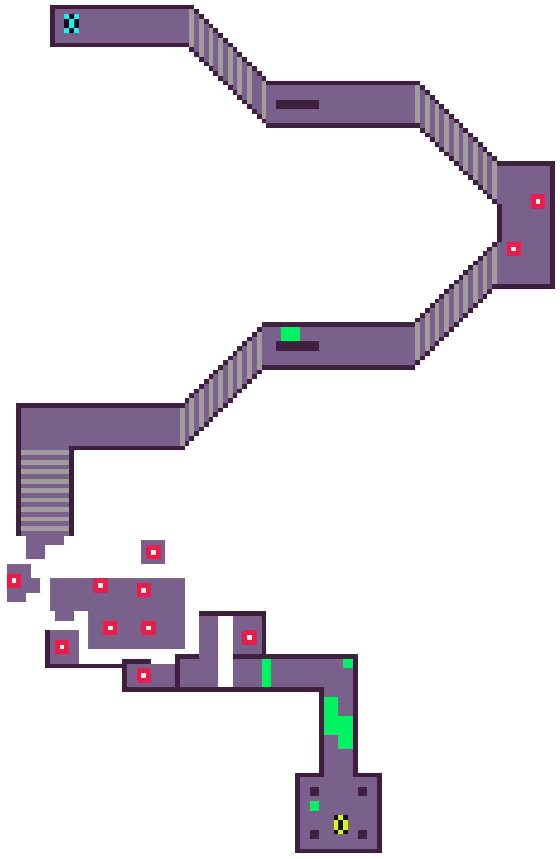
4. Pilgrim’s road

Following the path, the player can see a parade of cultist returning to the cathedral not so far ahead. The second stretch of the road is a bridge over with a frail foundation and the ground shatters as you step on it. This leads to a frenetic race to arrive safe to the other side. If the player falls and Pire dies, she will respawn half through the level, just before the bridge.



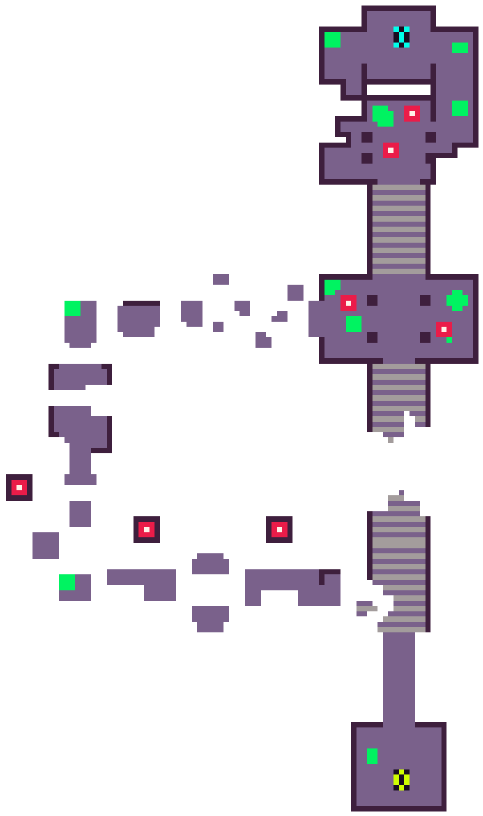
5. Holy Stair

This level opens with another tight combat that requires to maneuver over precipices. The second section of the level is a long walk up the stairs of the cathedral. The cultist are aware of your intrusion and will drop flaming wheels rolling down to kill Pira and prevent her advance. The player must internalize the timing of these hazards to overcome them.



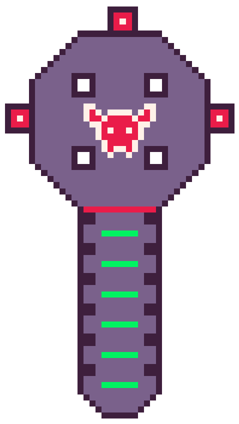
6. Ruined Path

Almost there, the cultists destroy before your eyes the path to the cathedral as a last resource to halt your pace. The player must find the hidden shortcut to reach the other side. This section has difficult platforming action without any cover from enemy projectiles.



7. The Cathedral

The end level where the boss fight takes place.



2.4.4 Universal mechanics

Breakable obstacles: These props function both as decoration and as a way to control the pace of the player blocking exits. Since they also block the enemies advance, they are useful as a tactical resource to cover from their attacks. This type of terrain is featured in three types:

- **Tall Grass** reduces movement speed but the player can easily cut through it.
- **Bushes** block movement and only take one hit to get destroyed.
- **Pillars** block movement too, however, they take several hits before breaking apart.

Precipices: Gaps in the terrain. If the player character falls into them, it respawns with less health. The player must use the Roll ability to overcome this gaps. There is also a variation of terrain called **Falling Ground**, which is terrain that disappears when the player steps on it, becoming precipices.

Firewall: When facing an enclosed enemy encounter, firewalls are spawned on the entrance and exit of the fight. The fire puts out when every enemy is defeated and the combat ends.

Firewheels: Fast moving objects that roll down steep terrain or stairs and burn on contact. The player can dodge them or run away since attacking it has no effect on this contraption.

2.5 Enemies

Enemies in the game are crazy zealots that belong to a fire obsessed cult. They follow the orders of a Supreme Priest.

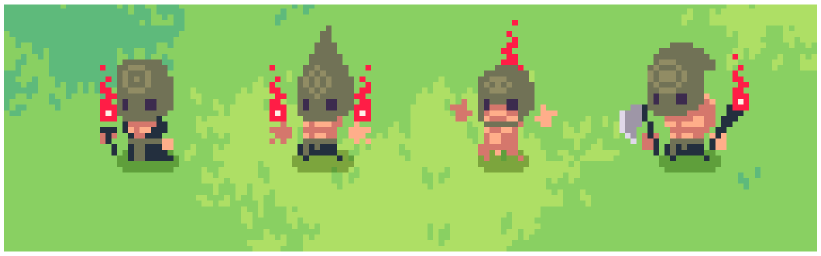


Figure 2.6. Concepts for the lesser enemy types

Pyromancer: The pyromancer is a long ranged enemy that casts fire barrages towards Pira and tries to keep a distance from her. They have 2 hit points and the ability to teleport themselves after a small channel time.

Pyromaniac: The pyromaniac is a medium ranged enemy that casts a fire cone in front of him. He also has the ability to suddenly dash forward to get in range for his attack. They are slightly tougher than the pyromancer, with 3 hit points.

Immolator: The immolator is an erratic unit that runs chaotically before setting himself on fire and running towards Pira. If he reaches her, he will explode and stun Pira with the blast. They have only one hit point.

Executor: The executor is a tough melee enemy that can deal high damage with his axe. They have 5 hit points and can stun Pira with their axe strikes.

2.6 Boss: The Supreme Priest

The Priest awaits the player inside the cathedral, the last level. The encounter is divided in three different phases where the boss displays a different behaviour and set of attacks. Each phase, the player has to deal 30 hit points to the Boss, for a total 100 hit points for the whole fight.

Phase 1: During this phase, the Priest acts elusive, keeping distance from the player with the ability to teleport anywhere in the room. His attack pattern is predictable. He casts barrages of fire projectiles that travel at slow speeds. Slowly they accumulate filling up the room. Alternatively he summons pyromancers to help him.

Phase 2: In his second form, the Priest uses two huge sickles and relays on melee combat. Charging forward to slice Pira in two. He only stops to catch his breath and channel a powerful beam of energy aimed directly at Pira. The only way to avoid it is to find cover behind a pillar. At the end of this phase, Pira beheads the Priest.

Phase 3: The cultists start eerie chants and immolate themselves. Their souls converge in the carcass of their deceased master and turn it into a demonic flesh aberration. The final phase starts and the pace of the combat gets frenetic. The monster is able to produce spontaneous combustions. Pira has to step outside of the hot areas before they explode. It can cast fire barrages like the ones in the first phase, but this time they are harder to predict to avoid the damage.

If the player beats this phase and gives a final death blow to the monster, the game will end.

3 Project development

This section is dedicated to explain the creation process of the project. The reasoning behind the implementation choices and a breakdown of its elements. The order in which each concept is presented follows the actual progression that the development had.

3.1 Character Controller

3.1.1 Movement

At the core of every great action game there is a great character controller. Character controllers are responsible for controlling the movement of the character: how they respond to player input and interact with the world.

In Unity, objects are usually moved in two different ways. Either by changing their position directly or by applying forces to a rigidbody[19] component and leaving the physics engine to deal with movement. Rigidbodies replicate quite well the behaviour of real world physics of simple objects. On the contrary, the kind of controls we require for our character are composed of complex actions and must be parameterized by hand. In this case, by directly modifying the position of our controller, we are able to finely tune how it moves around. This is essential to get a game that feels nice and plays fluidly.

This line of code on the Update method of the Controller2D class will translate the character a certain distance every frame, multiplied by deltaTime to ensure it remains framerate independent.

```
transform.Translate(velocity * Time.deltaTime);
```

3.1.2 Game feel

This method grants a great precision when interpreting the player input but it can feel stiff and rigid. This happens because we move from a full stop to the full acceleration instantly and vice-versa. If we move something with direct input from the player, there is simple trick but really effective trick to improve the feeling. When the input to move is received, we apply a constant acceleration to the speed up to a limit. When the input stops, we apply a constant deceleration [7]. The Figure 3.1 compares the two approaches.

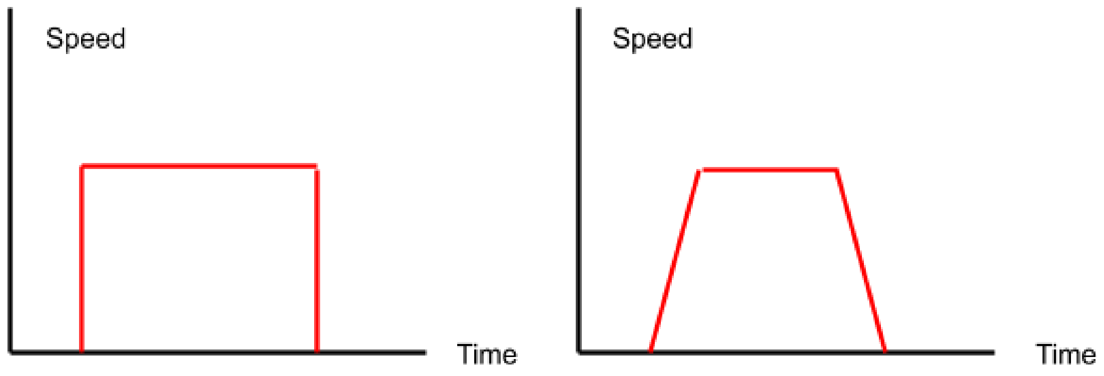


Figure 3.1. Comparative of the two methods to change movement

In Unity, we can just apply a *Vector2.SmoothDamp()* method to the speed to get this smooth transition between standing still and moving. Finding the right values takes trial and error, and depending on the parameters used we can arrive to different conclusions. If we keep short transitions, we are emulating a high acceleration and high friction that results in great responsiveness. The feeling is snappy, since it sacrifices precision for speed.

On the opposite, having slower transitions result in floaty movement that become unbearable to play with. With low acceleration and friction, there are no immediate changes in direction, so the player does not feel in control of the character.

Instead of the previous approaches, the best game feel comes from giving the player more agency over the character. We can get great precision by reducing acceleration and keeping a high friction. This way, we get in motion gradually and suddenly stop as required. As an exemption, when we want something to have impact, we should reverse the parameters to have instant acceleration and gradual deceleration. This will be really useful for the dash mechanics of the character.

Up to this point, the character can move freely since it does not take into account any limitation or boundary of the world. To be able to interact with the environment, we need some kind of collision detection that prevents the character to get through obstacles.

3.2 Collision detection

3.2.1 Alternative to Unity's physics simulation

Unity's physics engine was built for physics simulation. Unfortunately, simulations tend to behave in a non-deterministic way and produce inconsistent results in favor of realism. For the purpose of this game, realistic physics are not required. On the other hand, consistency in movement and collision detection is a must [6]. Collision detection is in charge of identifying the boundaries of the map and hits between the different entities that populate the game scene. The component rigidBody2D of the Unity 2Dphysics engine is perfectly capable of doing this task. However, relying on the physics engine for this specific calculation is a waste of performance. Even if we ignore the extra costs, there are still some edge cases where this type of collision detection lacks precision. One of these pitfalls is that unless physics are computed in the expensive continuous collisions mode, they may fail to detect a moving object intersecting others at great speeds. Even if we overlooked all these problems assuming our target platform is a powerful PC, using the Unity physics engine would not result in sharp and precise controls for the player character movement (I will expand on this on the 2D character controller section). Building our own system allows a fine tuning that we can not get from anywhere else.

For the former reasons, I opted to implement my own 2D collision detection system based on ray casting. Ray casting is the method of casting a ray towards a direction that then picks up information about what intersects the ray. Using this method, explained in the next subsection, we can determine how the character controller moves around the world.

3.2.2 Raycast system

Conveniently, Unity provides a Raycast[18] function which is an incredibly cheap operation. It is conceptually like a laser beam fired from a point in the space towards a particular direction. Any collider that makes contact is detected and reported. Moreover, the function returns a RaycastHit object with a reference to the collider that was hit. Their already fast performance can be further improved on using LayerMasks. This option allows to only apply the detection to certain types of objects, reducing the amount of overhead from unnecessary checks.

In this approach to collision detection these are the three main steps applied:
We have a moving object with a certain speed, direction of movement and position in a 2D space.

1. We break down the direction of movement in the “x” and “y” axis (we ignore the “z” component since we are working on a 2D space).

2. We cast rays in both “x” and “y” axis with a length equal to the exact distance that we attempt to cover in the next frame. This distance is obtained with the product of the speed and the delta time of the update rate of the screen.

3. If no hit is detected, then we move the object to the desired position and repeat the process. If the ray hits some collider, the hit point will define the maximum value the object will be able to reach on the axis.

This method [5] provides extremely precise collision detection, independent from frame rate and speed, because the length of the rays takes this parameters into account. For instance, if we had to move the character controller at really high speeds (or we had a low frame-rate due to a performance spike) the raycasts simply would stretch to cover a longer distance. This way, the collision detection check can not skip a collider passing through it.

In order to account for the total extent of our character collider we should at least cast 2 rays per axis, one from each corner. However, if there were obstacles smaller than the distance left between them, more rays should be used. To account for this scenario, I implemented a RaycastController class that takes the maximum gap that can be left between raycasts and dynamically casts more or less depending on the size of each side. Now, having several raycasts per side, we must iterate along them to get the one that returns the shortest distance to a collider and discard the others.

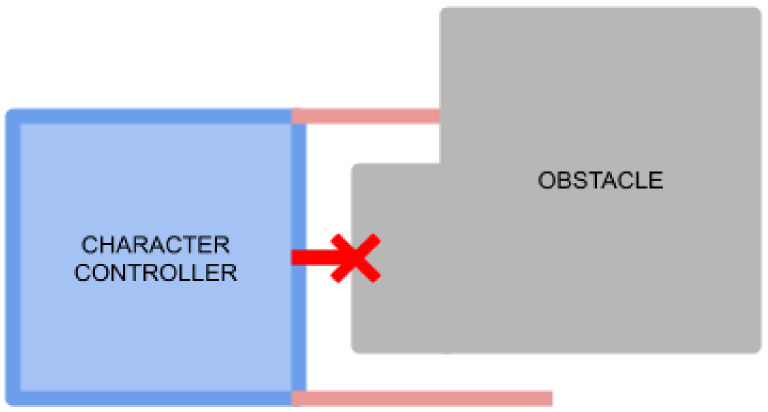


Figure 3.2. Character controller detecting an obstacle

I came across an important issue when casting rays from the edge of the character controller bounds. When the player controller is standing close to a collider, we still need a small amount of space from which we can effectively fire the rays. This was solved by adding an indent of small width to the point where rays are being cast.

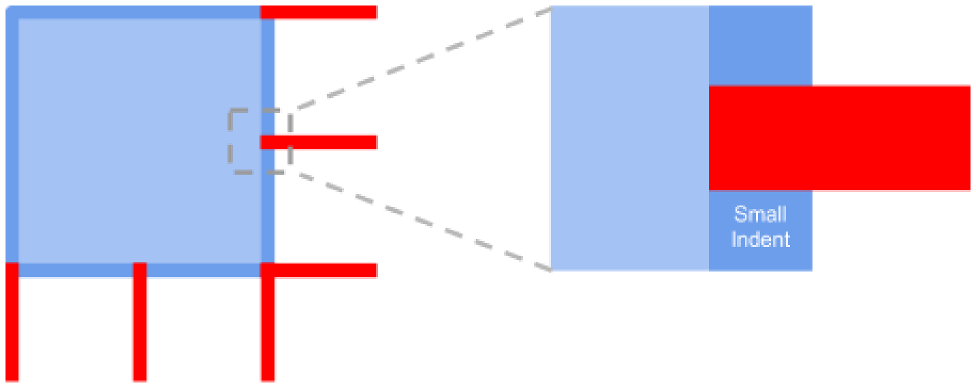


Figure 3.3. Highlight of the indent that offsets the origin of the raycasts

3.3 Input Manager

3.3.1 Processing the player input

Unity historically has lacked from a robust input manager that eased the addition of the numerous controller mappings that exist out there. This is going to change soon as the next version of the engine is going to make available a great input manager which simplifies the current workarounds.

For this project I had to implement an alternative solution to communicate the actual input of the player with the character actions in the game [8].

On one side we have the user, who has multiple ways to send the input. On the other, the in-game actions that must be triggered as a result. Since the game is built for PC, mouse & keyboard work by default. Although, I designed a gameplay that favors a console controller, so I focused on getting XBOX Controller support too.

If the user tries to perform an action, we must get a keypress from the keyboard or a button press from the XBOX controller. Regarding the code, we could implement a condition for each kind of input of every controller type and tie them directly to the action. However, with the introduction of a new controller we have to repeat the process. Likewise, when a different action is added, we need to add new conditions to check for each controller.

To take a better approach we can create a layer between the input and the game. A class able to read the input from all the controllers and convert it back into a single virtual input. In our code we can then use our virtual input to call the appropriate action function. This way, if we were to end up adding more controllers, we would only have to connect them to our new virtual input and no to every single piece of code triggered by these inputs.

This gives us more control of what is sent to our game. We then can block this input or alter it while it is actually being sent. In Figure 3.4 we can visualize the result of this configuration.

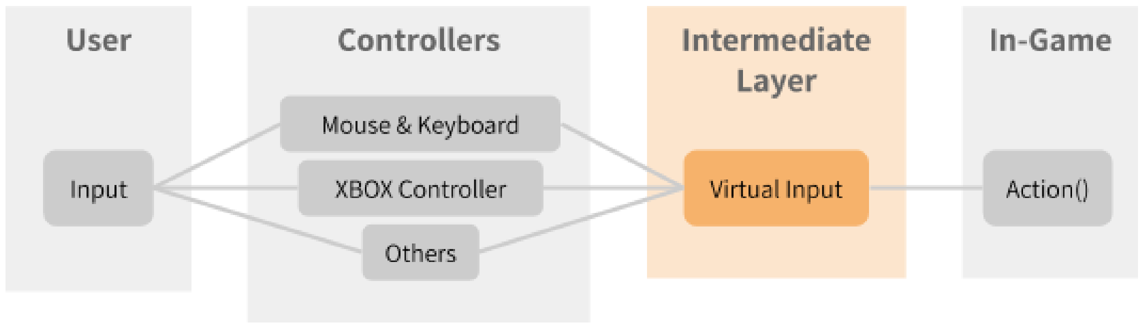


Figure 3.4. Diagram of the communication between the input related classes

3.4 Gameplay mechanics

Gameplay encompasses what the players can do and how the game responses to their actions. This interactivity with the game state relies in some elements we call mechanics. The interaction of different mechanics with synergy between them can create great depth for the gameplay. In this game, the player has access to three actions: running around, rolling and attacking.

Running is the most intuitive of them. It allows the player to traverse the space and position itself for further interaction. While running, the player character has great maneuverability but it is vulnerable to attacks. The implementation is really straightforward, building upon the 2D controller we simply update the direction of movement according to the players input and multiply it with the movement speed.

Rolling is a defensive action. It allows the player to reposition quickly to dodge incoming attacks. It requires some planification ahead, since it can not change direction halfway. In combination with the ability to run, it grants extra speed to travel faster. It also comes in handy to close gaps for an offensive approach. To implement it, we simply add a special case for running. In the direction the player is facing we apply a sudden burst of speed with a negative acceleration. While the speed is different to zero, we do not allow the direction to change and any further player input is ignored.

Attack has the more depth to it. When used, it covers a short range in front of the player character and damages anything caught inside. It can be used offensively at closeup range, just as a straight up damage dealing ability. However, and more interesting, it also deflects projectiles, sending them back in the facing direction of the enemy. This makes a great defense against projectiles. With good aim, the projectiles can even be turned against enemies.

To implement the attack action, we can create a method that calls Physics2D.OverlapCircleAll in a radius slightly offset to the direction the character is facing. This method returns a list with all the colliders inside the defined area. Then by iterating through the list we can call the method TakeDamage() with parameters for damage and direction of the hit. Only those objects that implement the interface IDamageable will take damage and/or get pushed in the direction.

There are a series of tweaks to balance things and prevent the player from mindlessly pushing the attack button. If the attack button is hold down or pressed in quick succession, the attack turns into a flurry of sword swings. It becomes impenetrable but deflected projectiles are weakened, losing accuracy and speed. Additionally it slows down the running and rotation speeds of the player character. These modifications result in a reliable tool to withstand a huge barrage of projectiles. However, they momentarily take away all the offensive power and open up a weak spot for non-projectile enemy attacks.

3.5 Camera

3.5.1 Achieve pixel-perfect camera

By default, Unity does not render pixel art properly. Both the camera and the sprite import settings must be adjusted to get pixel perfect style. By this, I mean getting the art style exactly as it was designed, without stretching, missing or duplicated lines and blurriness [9].

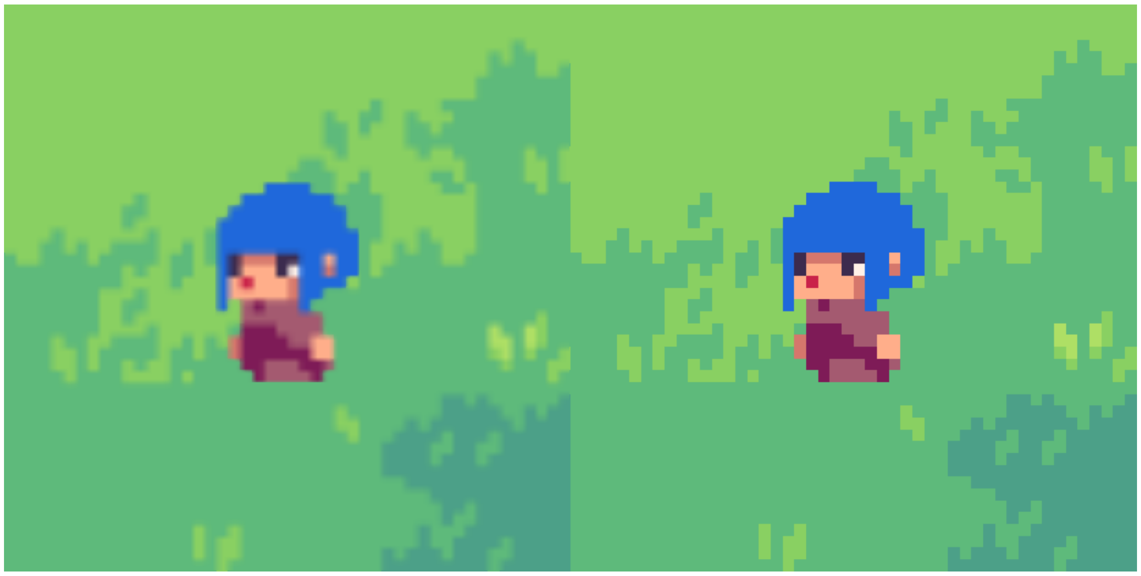


Figure 3.5. Comparative between blurred sprites and pixel perfect sprites

First, anti aliasing should be turned off. It is a useful rendering technique that blurs the jagged lines that appear when rasterizing 3D art to the screen. We need the opposite effect, clean cut pixels in the screen.

The import settings for sprites also try to hide pixels using a blur filter, this should be disabled. In addition, Unity applies an image compression by default, to prevent large textures from consuming too much memory in game and reducing loading times. Since our sprites are really small and must retain their original colors, compression is unnecessary and even counterproductive.

Now that the pixel art sprites look as intended, we need the camera to render it correctly to the screen. PC screens usually have a 16:9 aspect ratio, with resolutions of 1920x1080 pixels. However, there are screens with different resolutions and aspect ratios, and the game should look equally as good on all of them. The key to achieve this is to settle for a smaller resolution that favors pixel art. I chose 480x270 as the native resolution of the game because it keeps the common 16:9 ratio. The game will run on this size and then get upscaled to perfectly fit standard or atypical screen sizes.

Next, in the camera configuration, we set an orthographic size equal to the height of the native screen resolution divided in half. This way we get to match sprite pixels with our native screen pixels.

3.5.2 Camera controller logic

While designing a camera controller there are three main challenges to face: what the player wants to see, what the designer wants the player to focus on and how to achieve both in a comfortable and fluid manner for the player [10].

Interaction: The camera should give the players control over what is displayed and tie the changes to their controls so that they are predictable.

Attention: The camera needs to provide enough game info and feedback about what is happening at any given time by framing the action.

Comfort: Changes must be done with ease and contextualization to reconcile the two previous requirements.

Being a top-down game where threats can come from any angle of the screen, it makes sense to center the camera on the player character. This ensures a balanced amount of vision in any direction and keeps a clear point of reference even in chaotic situations. However, to give more agency to the player, we take the aiming input of the controller into account. Through it, the player can shift the camera to see further in that direction. This is done by averaging the focus of the camera between the player character and the aiming target.

To transition smoothly between both, the camera position is interpolated at a speed that feels quick and responsive. The speed should be capped to a reasonable value nevertheless, to prevent fast movements that may cause disorientation and nausea. This happens during dashes and fast impulses, where the camera does not follow the character instantly and gets dragged behind before catching up.

3.6 Projectile system

3.6.1 Object Pooling

The game's core design borrows the idea of a huge amount of proyectiles displayed at once from the bullet-hell genre. Each projectile is an instance of a Unity prefab. A naïve implementation would be to instance new projectiles each time and destroying them later as required.

This can cause some fundamental problems because of scalability. For small amount of objects, instancing and destruction operations are trivial. However, as the quantity of these operations increase, they start affecting performance. This is especially apparent in the case of destroy operations due to the nature of the programming language used in Unity. C# does not give direct control of memory management: when a Destroy() function is called on an object, it gets marked as pending to kill and remains in the memory instead of being destroyed right away. It is the job of a system called garbage collector to remove any objects marked as pending to kill from the memory. It does this task regularly on its own and as developer we do not have any control over when it will happen.

As a consequence of getting numerous pending to kill objects at once, the garbage collector can freeze the game while doing its job. This would result in stuttering and gameplay would be negatively affected.

Here is where object pooling becomes useful. Object pooling is an optimization technique that gives an extreme boost of performance in these cases. The idea is to instantiate a pool of objects and recycle them, rather than continually having to create and destroy new ones [11].

I created a PoolManager class that takes in a Unity prefab and a pool size integer. It instantiates the amount of objects specified and stores a reference of them in a queue. When we are reusing an object, we simply take the first one out of the queue, that being the oldest object, and then just add it back on to the end of the queue when we are done with it.

3.6.2 Further optimizations

With object pooling we can take care of instancing hundreds of objects in the game. But there are other expensive tasks that greatly escalate cost with the amount of projectiles in use. To start off we must update their positions on each frame; change speed, trajectory direction, etc. If we treat each projectile as an individual object with their own logic and collision detection in the scene, we are starting to get low performance.

Fortunately, Unity has lately come up with some huge improvements in terms of performance in writing optimized multithreaded code. At the center of this changes is the new Entity Component System (ECS) [12]. Using ECS allows us to write extremely efficient code in certain scenarios. ECS is a new way of writing code in Unity, moving away from object oriented programming to something called data oriented design.

Up until recently, Unity has been pretty much based on using gameObjects and monoBehaviours. With this approach, creating a projectile would mean making a gameObject and attaching monoBehaviour components to give the projectile functionality. This monoBehaviours are strips of code that would take care of rendering, physics and movement. With ECS we split this projectile gameObject into 3 parts. An entity, a component and a system. Entities group together components and components contain data. Unlike traditional monoBehaviours, these components do not have any logic in them. Instead we use a system to contain the logic that defines component based behaviour. In other words, the system is responsible to operate on all entities with a specific set of components.

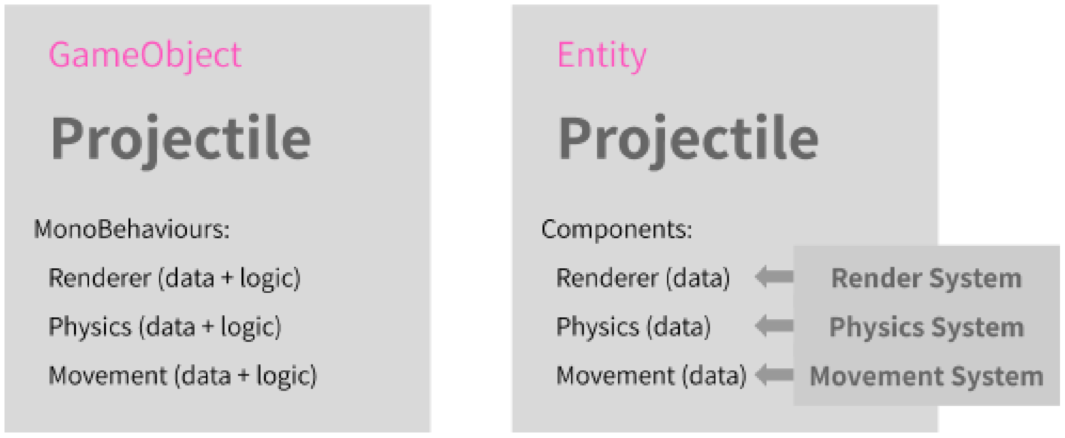


Figure 3.6. Comparative between traditional OOP and ECS paradigms

There are two ways to implement the new ECS in the game. The pure ECS approach is completely different from the old coding paradigm, scrapping gameObjects and monobehaviours altogether. On the other hand, hybrid ECS makes it easier to transition between the old system to ECS by combining them. This does not bring the full potential of ECS but it is more manageable as a starting point since it allows the use of monoBehaviours to store data. The standard scripting method had monoBehaviour scripts with data and a behaviour. The hybrid ECS still maintains monoBehaviours to hold data on each object, but relies on a Component System to handle the behaviour applied to those objects, thus becoming entities.

Our system will be called ProjectileSystem and will be tasked with updating entities with the component Projectile. This component holds the required information to display a projectile with its current speed, acceleration, layer mask, animation etc. Now, each projectile on the screen is updated from our streamlined ProjectileSystem instead of updating individually on its own. During my performance tests, this implementation allowed the game to increase the maximum count of projectiles from 300 to a 1000 keeping a steady framerate of 60 fps.

A 333% improvement seems great, but according to the performance profiler, physics calculations are restraining the game. Apparently, our raycast collision detection is not scalable for its use on more that a thousand entities. As explained before, raycast are one of the cheapest operations for collision detection so we need an even lighter solution. Carefully analyzing the projectiles behaviour, I noticed that all the projectiles in screen are checking for collisions even with trajectories facing away from the player character. This means that projectiles that will never collide with the player are doing pointless collision checks and that is a huge waste of resources.

In summary, we can assume some simplifications to strip the projectiles of all collision detection based on physics functions. We can consider that both the projectiles and player character have circle boundaries, determined by a point in space and a radius. At this point, we can infer a collision takes place if the squared distance of separation between them is smaller than the addition of both squared radius. This type of solution is viable only because the comparison happens between multiple projectiles to one player character, so the cost of the problem is O(n) and scales linearly.

After this optimizations, the maximum count of projectiles went from 1000 to a 10000 keeping a steady framerate of 60 fps. In addition to the pooling system, the result is virtually infinite bullets at our disposal.

3.6.3 Projectile patterns

Bullet hell games essential trait is the overwhelming number of enemy projectiles displayed on screen. However, their major appeal is the impressive formations and patterns emerging from the cloud of pellets moving around. The previous optimizations allowed this game to run with tens of thousands of bullets. At this point we need to take advantage of them with a highly controllable system that allows to script diverse and rich patterns.

The implementation a projectile emitter script that takes as many parameters as I could came up to set trajectories and get interesting patterns. For clarity the parameters were arranged into two types: emitter and projectile settings.

The emitter settings determine the direction and position where the projectiles will start from when spawned. They also cover the rate of fire, and either the duration of the barrage or number of shots fired. On the other hand, the projectile settings take care of the speed, acceleration, lifetime and special qualities, such as explosion radius upon collision or target seeking. The interesting part comes from the combination of all the parameters, which result in emerging patterns that would be really hard to obtain otherwise.

Through trial and error I got multitude of interesting and challenging projectile patterns. Although, in order to use them later, it became evident the need for a system to store them efficiently. To achieve this, I used Unity scriptableObjects. ScriptableObjects are data containers able to store data as an asset [13]. Those assets can then be accessed at runtime by other scripts. In this case, each scriptableObject stores different values for the parameters the projectile emitter will use. The projectile emitter has an array of scriptableObjects references and access them following a timed sequence or external command.

The list of the parameters is as follows:

Emitter settings:

Angle: The direction at which the emitter is aiming defined by 360º.
Spread: A ± random variation added to the previous angle on each shot.
Tracks: Number of simultaneous shots.
Arc: The arc defined by up to 360º that the previous tracks must cover.
Gap: A ± forward offset to the initial spawn of the shot.
Separation: The distance in which the tracks must separate from each other.
Tilt: The direction in which this separation takes place.
AimTarget: A boolean that controls whether the emitter should face its target automatically.
RotationSpeed: The speed at which the emitter changes its aiming direction.
RotationAcceleration: The acceleration of the previous rotation.
Oscillating: A boolean that controls whether a sine wave should offset the aim direction.
Frequency: The frequency of the sine wave.
Amplitude: The amplitude of the sine wave.
Rate: The time between shots.
Duration/Ammunition: The maximum of either time or shots the emitter must perform.

Projectile Settings:

Seeker: A boolean that controls if a projectile should steer automatically towards its target.
Speed and acceleration: Speed at which the projectile moves.
Angular speed and acceleration: The speed at which the projectile changes direction.
Lifetime: The time that each projectile lasts before being destroyed.
ExplosionRadius: The circular area that gets damaged when the projectile is destroyed.

3.7 Managing damage and health

Enemies and projectiles pose a challenge to the player because they are hazards that can damage and eventually kill the player's character. In action and combat video games, health is usually represented through an abstraction of discrete numbers or stats. Health points, for example, are convenient and easy to understand. They give instant feedback of the cause and effect of actions, and let the players plan effectively when they know exactly how much damage they can take before death. This abstraction of health works with a number of maximum points that decrease when damage is received until no more is left and the character dies.

The player character, enemies, the boss and destructible terrain share a similar reaction to damage, it depletes their health and destroys them when it reaches zero. Aside from that not much is shared between them, so it seems obvious that an interface could come in handy for this. IDamageable is the interface I created to call a method named TakeDamage(). This method takes care of updating the health of the bearer of said interface when it takes damage from any source. Aside from the amount of damage taken, it can also get the direction of the hit and the thrust applied.

Once health is depleted reaching a minimum of zero, a Death() function is triggered. In the case of simple enemies or proyectiles, this implies getting disabled and returning to their respective pool queue to be recycled. In the case of the player character, the whole scene gets restarted to its original state through a loading screen that dramatizes the game over condition.

3.8 Enemy AI and behaviour

3.8.1 AI in video games

Artificial intelligence is used broadly in video games to carry out behaviours and responses to the environment and player interactions. Within a game, implementing a traditional AI is not generally the goal. Nevertheless, many traditional AI techniques are borrowed to be applied in some way. However, the AI of games usually consist of a set of predetermined responses to limited inputs that, in consonance with the gameplay, manage to create the illusion of intelligent behaviour. In other words, its mission is to serve the gameplay restrictions and create a balanced challenge to the player.

The most relevant tasks of AI in games are pathfinding, decision making, and procedural-content generation. This last one is a method to create content algorithmically, which is not a goal of this project. On the other hand, pathfinding is used extensively through this project with two techniques: A* pathfinding and steering behaviours.

3.8.2 Pathfinding

A* pathfinding is a pathfinding algorithm, which can find a path between two nodes inside a graph [14]. It also happens to find the shortest path possible even when navigating through obstacles. It is quite efficient, only outperformed by algorithms that precompute the graph to avoid the runtime calculations. To use it appropriately, we need to translate our world into a grid of nodes. Such task is trivial in our case, since the maps are already built up inside a grid.

Inside of this grid we should have walkable nodes and obstacles. In order to know the shortest path between A to B we need establish some things. First, we decide the distance between each node. Generally we can assume it is a single unit, therefore, a diagonal move is the square root of two units. To simplify things, we define an orthogonal move as a distance of 10, and a diagonal move as a distance of 14.

With the grid of nodes set and ready, the algorithm begins by going to the starting node and checking all the surrounding nodes. Then it calculates three numbers for each node.

1. The G cost is the distance from this node to the starting node.
2. The H cost is the distance from this node to the end node.
3. The F cost is G cost plus H cost.

Finally, the node stores the neighbour node that led to it.

Once this step is completed, the algorithm will repeat the process selecting a new node with the lowest F cost. If two nodes have the same F costs, then it will select the one that is closest to the end node (smaller H cost). This also takes into account any previous nodes whose costs were calculated.

Eventually, the algorithm is guaranteed to find the shortest path. This is because as soon as it is not moving in an straight line, the F costs of consecutive nodes will keep increasing and that will force it to look for other routes. Once it reaches the target node, we can retrieve the path that led to it because each node stored the previous node.

Here is an A* algorithm pseudo code definition:

```
OPEN // set of nodes to be evaluated
CLOSED // set of nodes already evaluated
add the start node to OPEN

loop
    current = node in OPEN with the lowest f_cost
    remove current from OPEN
    add current to CLOSED

    if current is the target node // path has been found
        return

    foreach neighbour of the current node
        if neighbour is obstacle or neighbour is in CLOSED
            skip to the next neighbour

        if new path to neighbour is shorter OR neighbour is not in OPEN
            set f_cost of neighbour
            set parent of neighbour to current
            if neighbour is no in OPEN
                add neighbour to OPEN
```

Figure 3.7. A* pathfinding pseudo-code

3.8.3 Optimization of the A* algorithm

The main bottleneck of this algorithm happens in the first step taken at the start of the loop. To select the current node from the OPEN set we need to traverse this entire list each time. Such operation is required to find the node with the lowest F cost since they are in no particular order. This has a worst case cost of O(n) that grows linearly with the amount of nodes that populate the OPEN set.

A much better approach is to use a different data structure known as heap. A heap is a binary tree were each node has two child nodes. Then each one of those nodes has other two child nodes and so on. The heap structure also follows a strict rule that forces each parent node to have an F cost smaller than both of its child nodes. This setup is achieved with a really simple strategy. First, each new node that enters the heap is added at the end. Then, to find its proper location, we simply check its F cost with the parent node. If the cost is lower, we swap the positions between them. The process is repeated until the current parent of this new node has a lower F cost, or if we reach the root.

Using this method, we raised the new node to the appropriate location with only a few steps. Precisely, the cost of inserting or deleting a node using this binary heap is O(log n), which is much better than the previous linear cost. And since we always have the lower F cost node at the root, we can access it easily, removing the bottleneck we had.

At the end, we obtain a precise and optimal way to find the shortest path to a target through obstacles. However, if we were to interpolate the position of the enemies to move through the path of nodes, the result would appear as if they moved on rails instead of walking. Due to this, the following task is to get an organic and interesting way of traversal. This can be achieved using steering behaviours.

3.8.4 Steering behaviours

Steering behaviours were first proposed by Craig Reynold in 1986 as rules for a program called Boids. Boids is an artificial life simulation program whose complexity arises from the emergent behaviours of individual agents interacting with each other. These interactions follow a set of simple rules known as steering behaviours. They aim to move characters in a realistic manner, using simple forces that combine into life-like navigation around the environment [15].

The implementation of the forces involved in steering behaviours follow an additive approach. To start off, we need the agents to be able to move towards a target. This is called seek behaviour, and later will be used to follow the path computed by A* using subsequent points in the path as targets. For seek behaviour, two forces are involved: desired velocity and steering. Desired velocity is a force that pushes in a straight line to the target, whilst steering is the result of the desired velocity subtracted by the current velocity. This prevents our agent to change direction abruptly, and instead steer smoothly towards its target.

```
desiredVelocity = normalize(target - position) *maxVelocity
steering = desiredVelocity - velocity
steering = truncate (steering, maxForce)
velocity = truncate (velocity + steering, maxSpeed)
position = position + velocity
```

Figure 3.8. Implementation of seek behaviour

The steering force gets truncated, to avoid exceeding the forces that the agent can handle. The velocity gets truncated as well, since otherwise it would add up indefinitely when facing the target.

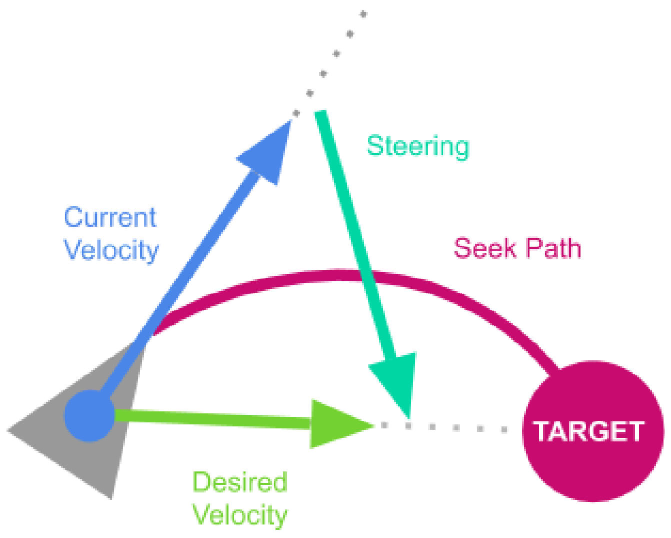


Figure 3.9. Visualization of the forces involved in seek behaviour

To get a fleeing behaviour we can just get the inverse vector of the desired velocity. This will cause the agent to flee from the target. Finally, to achieve the the behaviour of path following, we can simply iterate through an A* path updating the target to each node once we reach it.

At this point, when several agents are spawned and their trajectories cross over they do not react to each other. To get a natural movement from many agents we can apply flocking behaviours using three main rules: alignment, cohesion and separation.

Alignment is the behaviour that causes an individual agent to align itself with agents that are close by. In Unity we can use a Physics2D.OverlapCircle to get all neighbour agents in a radius. Then we add up all the velocities of the neighbours to the velocity of the agent divided by the total count of neighbours.

Cohesion is the behaviour that causes agents to steer towards the average point of their neighbours location. As previously, we use Physics2D.OverlapCircle to get all neighbour agents. However, instead of adding the velocities, we add up their locations to the agent's location divided by the total count of neighbours. As a result, we get the center of mass, and to get the direction towards it, we subtract the location of the agent.

Separation is the behaviour that causes agents to steer away from its neighbours. As before, we first get the neighbours. Then we need to add up the distance between each neighbour to the agent location and divide this by the neighbour count. Additionally, the computed vector must be negated to get the agent to steer away from the neighbour properly.

With all three vectors calculated for a particular agent, we just need to add them to the velocity. At this point is useful to add a multiplayer on each of this forces to leverage their contribution to the final velocity. The addition of weights can change drastically how the agents flock and interact with each other.

3.9 Boss behaviour

3.9.1 Finite state machines

Bosses in video games are powerful computer-controlled enemies encountered during a special event called boss fight. This confrontation tends to be the climax of the game and requires a particular strategy to be overcome. In particular, the boss of this game has three distinct phases. The boss behaviour is comprised of a sequence of actions, which, under some conditions may change to adapt to the situation. These conditions are mainly related to the players positioning in relation to the boss. For instance, on each phase the boss has an ability that provides a way to reposition itself. This is used in order to close distance or to keep a separation with the player when needed.

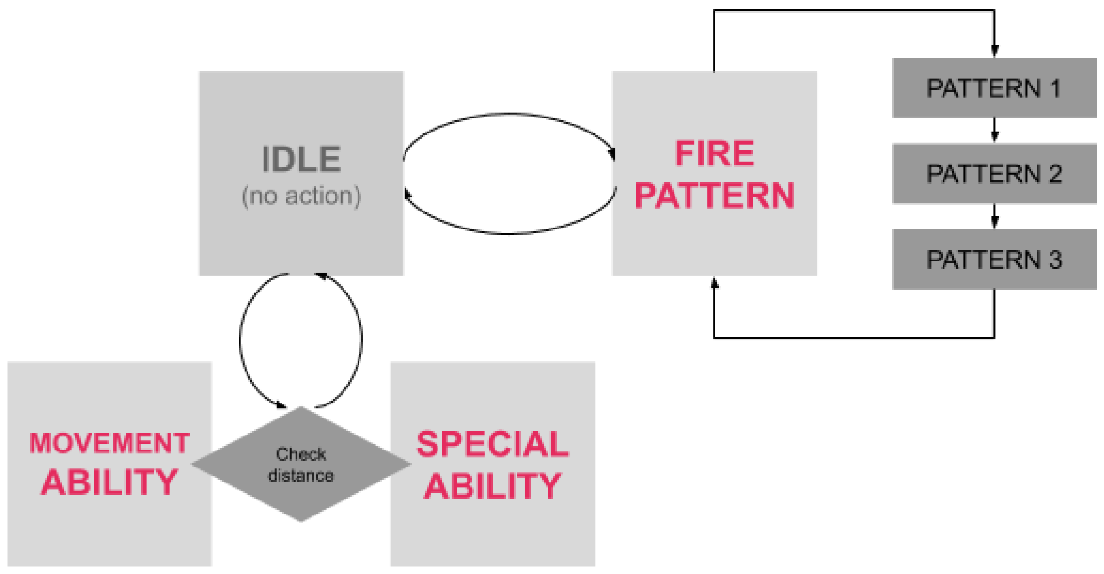


Figure 3.10. Finite State Machine diagram of the Boss behaviour

The implementation of this gameplay is done using a finite state machine [16]. The different states and their relations can be visualized as nodes contained inside of a graph. The nodes reference actions that the boss can execute. When the logic reaches a node, a certain state becomes active and its action performed. To transition between states, a certain condition must be evaluated as true. Some states last for a fixed amount of time while they keep executing an action, others happen instantly and transition to the next state immediately. Once an action is finished or a condition is met, the boss enters a new state. This new state is chosen based on different parameters such as the last action taken and the position of the player.

As previously stated, along the fight with the boss, there are three different phases that change its behaviour significantly. The change of phase is determined by a specific amount of damage received. The thresholds divide the total health into three equal parts for each phase. The boss is defeated once its health points reach zero.

3.9.2 Phases of the Boss fight

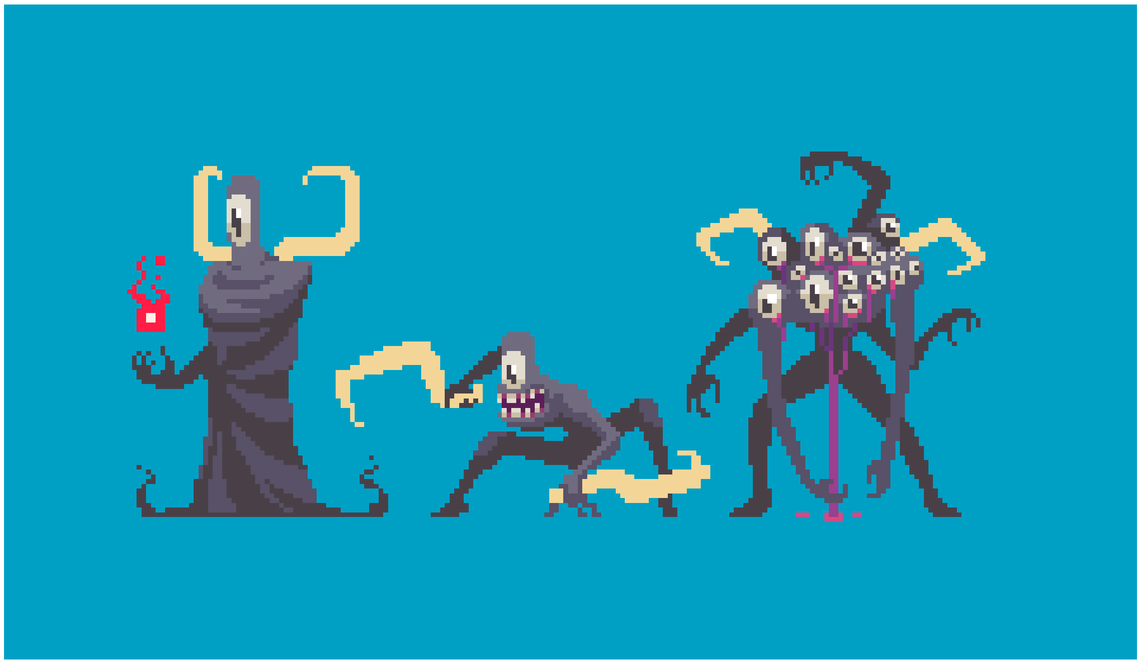


Figure 3.11. Concept art of the three Boss phases

Phase 1: Demented Priestess

During the first phase, the boss focuses on keeping a cautious distance with the player. To achieve this, it periodically teleports to a different point in the level. This action has more chance to happen the closer the player is. On the opposite hand, the farther the player is, the higher the chance for the boss to unleash a targeted laser beam. This precise ray takes a short delay to be casted, but once it does, it travels instantly and can not be deflected by the player. The only way to counter it is to hide behind an obstacle or dodging it.

In the meantime, the boss shots barrages of fire projectiles using the predesigned patterns. These are composed of slow traveling bullets that fly in all directions in distinct patterns. They also follow an organized sequence to enable the player to learn and predict the order.

Phase 2: Berserk

The second phase presents a change in the strategy of the boss. Instead of keeping distance, now it tries to get closer to the player. To achieve this, it periodically dashes forward in the direction of the player doing three slashing cuts. Between each cut, it can change direction slightly to face the player again. The strikes can not be parried and may only be dodged.

Periodically, the boss stands still and absorb the air around it creating a vacuum and pulling the player in the process. As a follow-up, it releases all the energy in a short-ranged explosion. The projectiles shot in this phase are short lived and of chaotic nature, so they are most dangerous at a short distance.

Phase 3: Abomination

The final phase is a callback to the first one, but much harder. This time, the teleports happen more often, with a chance of being used offensively as a surprise attack. Besides, instead of a laser, the boss can now conjure pillars of fire. These are squared areas that explode after a delay, pressuring the player to be in constant motion. Maneuvering the room becomes harder because there is no safe spot now.

Simultaneously, the boss shots fire projectiles echoing the first phase. However, the patterns have fewer but faster bullets, also aimed at the player. The player must plan ahead when to dodge to avoid stepping into a fire pillar, while displaying fast reactions to deflect the projectiles.

3.10 Environments

3.10.1 Tile-based design

The tile-map model is a remnant of older times where memory storage was the main bottleneck for game development. Early video game consoles had to fit all the sprites of art for characters and background in small cartridges of 4K in size. This left developers with a huge challenge to overcome; so they came up with the idea of building backgrounds with a minimal set of reusable tiles. These square tiles, of 8x8 pixels, could then be set on a grid as building blocks to create maps much bigger than the current size of memory would have allowed otherwise.

Nowadays, the memory savings that this method offers is no longer relevant. However, it is still quite helpful to reduce the amount of artwork required to create maps. After all, new maps can be created using the same tileset. It also shortens the iteration times to make changes without having to redraw, speeding up the process of level design.

A hindrance of this method is the observable pattern that arises from the combination of repeated tiles and the grid they are displayed on. This could be claimed as an artistic trait of the style, but I wanted to go further and resorted to Wang tiles [17].

Wang tiles were first proposed by mathematician Hao Wang in 1961. This type of tile-map uses a set of square tiles, with each side of a fixed color that can be arranged side by side in a rectangular grid, matching the color of their adjacent tile sides. The method ensures that even a small but complete set of hand-made source tiles can be assembled easily without obvious repetition or visual breaks between them.

There are many variations of tilesets that can apply this rules. I chose one that allows for a transition between two different types of terrain.

4 Conclusions

The resulting product is an action-adventure game that showcases the design and systems developed for this project. It is not in a finished state as a fully fledged game since content like art and sound exceeded the available time. However it is a proof of concept for the design and its underlying systems.

The following link gives access to a Google Drive folder that stores an executable game:

https://drive.google.com/open?id=1di5LIlbj4-iPkCbvsMzu73eVlsgn3Kms

4.1 Achieved goals

The goals described in the technical proposal have been met with different degrees of success:

Goal 1: *To design an engaging gameplay that is entertaining and challenging (easy to learn and hard to master). It combines the effortless action packed aspect of the hack'n'slash genre with the frenetic and precise controls that require the bullet hell genre.*

To develop a robust core for the gameplay, there was a huge investment in crafting systems such as the *character controller* with collision detection created from scratch. This served as the foundation for the implementation of the gameplay mechanics. These mechanics were in turn fine tuned around the projectile system to offer the best experience possible. Since projectiles are the main feature of bullet hell games, this projectile system received special attention and ended up being really powerful and flexible.

Goal 2: *To implement efficient artificial intelligence for the Priest (final boss) and its minions to produce an interesting behaviour. It is designed to pose a real threat, while avoiding erratic and unpredictable actions in order to be understandable by the player.*

AI for the game was implemented using pathfinding with steering behaviours for mobility and finite state machines for the decision making. The boss had more depth to its design than the enemies, however, its behaviour is more scripted to ensure the challenge it poses follows more strict rules. This is done to achieve a more balanced result that avoids arbitrary actions giving structure and pace to the encounter.

Goal 3: *To create an original setting and a variety of characters consistent with the world rendered in pixel art style. It requires frame by frame hand drawn art for the animations.*

Many of the designs that were planned could not be completed in time. Pixel art is a craft that takes lots of time to animate, and since the main focus of the project was design and programming, this section ended up being the weakest in comparison. Nevertheless, the ones that made the cut have a distinct style and polished look for high quality standards.

4.2 Planning deviation

The figure 4.1 shows the deviation that took place during the development of the project. I prioritised the parts of the project related to code and gameplay systems over the art. This is a significant drawback because the visual aspect of a game it the main barrier to be able to appreciate the work that went into the design and programming behind it. However, this was a compromise I had to take in order to reach a functional version that could be later polished to its full potential.

Task	Estimated time	Real time
Game Design Document	15	15
Implement the game mechanics and core systems	80	**100**
AI, behaviour and pacing of the encounter's phases	40	**60**
Character animations and VFX	50	**40**
Environmental art and world building	30	**10**
SFX and music	10	**3**
Implement the non-gameplay systems and miscellanea	10	**2**
Final Memory	60	**70**
Total	300	300

Figure 4.1. Table comparing the estimated times and real times for each task (in hours).

4.3 Future lines

The systems present in the game reached a fairly finished state and ended up working as planned with efficiency. However, there is still lots of content to make in the art section to polish the appearance of the game. Also, with proper time, more user testing should be carried out to balance the game difficulty and pace.

I intend to continue developing the game with the addition of all the art that is lacking right now, such as animations, FX and sound. I also plan to continue designing proper levels and extra content to reach a state of full game in the future. The foundation of the gameplay and the systems are already working flawlessly.

4.4 Personal reflection

Working on this project was quite satisfactory as it has allowed me to experiment and improve as a game developer in general. I spent a large amount of time coming up with the concept by trying different innovative twists to already existing ideas until I settled down. This gave me time to layout all the requirements and steps I needed to implement the final result, which was really useful for productivity. I had previous experience with Unity, but this project broadened my knowledge of the engine and C#, which additionally, made me more confident in my programming skills.

On the downside, I overestimated the actual time for the scope I aimed for. However, instead of rushing things, I worked hard on the foundations of the gameplay. Because of this, the project is actually valuable and can be expanded upon in the future.

5 Bibliography

[1] Vambleer (2013) Nuclear Throne | Presskit [online] Available at: https://www.vlambeer.com/press/sheet.php?p=Nuclear_Throne [Accessed 9 Apr. 2019]

[2] Heart Machine (2016) Hyper Light Drifter | IGDB [online] Available at: https://www.igdb.com/games/hyper-light-drifter/presskit [Accessed 9 Apr. 2019]

[3] Dodge Roll (2016) Enter the Gungeon | IGDB [online] Available at: https://www.igdb.com/games/enter-the-gungeon/presskit [Accessed 10 Apr. 2019]

[4] Acid Nerve (2016) Titan Souls | Devolver Digital [online] Available at: https://www.devolverdigital.com/games/titan-souls [Accessed 9 Apr. 2019]

[5] Pignole, Y. (2013) The hobbyist coder #1: 2D platformer controller | Gamasutra [online] Available at: https://www.gamasutra.com/blogs/YoannPignole/20131010/202080/The_hobbyist_coder_1_2D_platformer_controller.php [Accessed 27 Apr. 2019]

[6] Lague, S. (2015) Creating a 2D Platformer (Video Series) | Youtube [online] Available at: https://www.youtube.com/watch?v=MbWK8bCAU2w&list=PLFt_AvWsXl0f0hqURlhyIoAabKPgRsqjz [Accessed 15 May 2019]

[7] Venturelli, M. (2014) Game Feel Tips III: More on smooth movement | Gamasutra [online] Available at: https://gamasutra.com/blogs/MarkVenturelli/20140904/224866/Game_Feel_Tips_III_More_On_Smooth_Movement.php [Accessed 8 Mar. 2019]

[8] N3K EN (2016) Input Manager (Multiple Inputs) | Youtube [online] Available at : https://www.youtube.com/watch?v=NYZoLOpYp2k [Accessed 12 Apr. 2019]

[9] Tham, P. (2015) Pixel Perfect 2D | Unity3D [online] Available at: https://blogs.unity3d.com/es/2015/06/19/pixel-perfect-2d/ [Accessed 2 Jun. 2019]

[10] Keren, I. (2015) Scroll Back: The theory and practice of cameras in side-scrollers | Gamasutra [online] Available at: https://www.gamasutra.com/blogs/ItayKeren/20150511/243083/Scroll_Back_The_Theory_and_Practice_of_Cameras_in_SideScrollers.php [Accessed 20 May 2019]

[11] Placzek, M. (2016) Object Pooling in Unity | Raywenderlich [online] Available at: https://www.raywenderlich.com/847-object-pooling-in-unity [Accessed 2 May 2019]

[12] Shekhar, G. (2018) Entity Component System in Unity | Gyanendu Shekhar's Blog [online] Available at: http://gyanendushekhar.com/2018/08/01/getting-started-entity-component-system-ecs-unity-tutorial/ [Accessed 7 May 2019]

[13] Fisher J. (2018) Scriptable Objects in Unity | Raywenderlich [online] Available at: https://www.raywenderlich.com/6183-scriptable-objects-tutorial-getting-started [Accessed 13 May. 2019]

[14] Lague, S. (2014) A* Pathfinding (Video Series)| Youtube [online] Available at: https://www.youtube.com/watch?v=-L-WgKMFuhE&list=PLFt_AvWsXl0cq5Umv3pMC9SPnKjfp9eGW [Accessed 25 May 2019]

[15] Bevilacqua, F. (2013) Understanding Steering Behaviours | EnvatoTuts+ [online] Available at: https://gamedevelopment.tutsplus.com/series/understanding-steering-behaviors--gamedev-12732 [Accessed 25 May 2019]

[16] Sobolewski, P. (2016) Finite State Machines | The Knight of Unity [online] Available at: https://blog.theknightsofunity.com/finite-state-machine-part-1/ [Accessed 10 Jun. 2019]

[17] cr31 (2018) Wang Tiles | cr31 [online] Available at: http://www.cr31.co.uk/stagecast/wang/intro.html [Accessed 22 Apr. 2019]

[18] Unity Technologies (2019) Physics2D.Raycast | Unity3D Documentation [online] Available at: https://docs.unity3d.com/ScriptReference/Physics.Raycast.html [Accessed 2 Mar. 2019]

[19] Unity Technologies (2019) Physics2D.Rigidbody2D | Unity3D Documentation [online] Available at: https://docs.unity3d.com/ScriptReference/Rigidbody2D.html [Accessed 16 Mar. 2019]