E.T.S. de Ingeniería Industrial, Informática
y de Telecomunicación

The Cost of Lies

Grado en Ingeniería Informática

Trabajo Fin de Grado

Íñigo Domínguez Jalle

Jesús Villadangos Alonso

Pamplona, 07/09/2021

upna
Universidad Pública de Navarra
Nafarroako Unibertsitate Publikoa

"What is the cost of lies? It's not that we'll mistake them for the truth. The real danger is that if we hear enough lies, then we no longer recognize the truth at all. What can we do then? What else is left but to abandon even the hope of truth and content ourselves instead with stories? In these stories, it doesn't matter who the heroes are. All we want to know is who is to blame."

Valery Legasov. 1:23:45, Chernobyl **[1]**

Contents

Acknowledgements

I would like to thank Jesús Villadangos Alonso for his interest, dedication and passion for the world of video games and his crucial representation in the university community through his involvement in the *"Diploma de Especialización en Desarrollo de Videojuegos y Aplicaciones de Realidad Virtual".*

Thanks to all my professors throughout my university career, my always supportive family and my closest friends I have met here.

To my future me that he will know this as his early steps in his videogame industry career.

Abstract

The aim of this Final Degree Project is to develop a video game technical demo to study, compare and implement features from fields such as Physics, Procedural Random Generation, Performance improvements and Artificial Intelligence among others, using an Object-oriented programming language.

To accomplish this, the author has developed a playable 3D Videogame demo showing his skills on the field, algorithms studied, implemented and some additional comparations between techniques with their results and final conclusions.

When it is built, the demo allows the player to play and experience a First-Person Shooter with a simple story and a totally procedural random level system where the user can progress and advance until completion.

The project has been tested in different platforms and systems in order to analyze and implement performance improvements with the goal to reach 144 Frames per second, a smooth gameplay and overall good experience.

The engine used in the project is Unity Engine 2020.3.8f1 and the language C#.

Keywords: Videogame, Procedural Generation, Artificial Intelligence, Physics, Programming.

Introduction

The inspiration to do this videogame technical demo came from the fact that I aspire to become a videogames programmer in the future so I thought this could be the perfect opportunity to interconnect my passion for videogames with my current degree at the university being this work my first steps into becoming a videogames developer.

In this project I pretend to illustrate the knowledge I have acquired through all these years as a Computer Science student and videogames lover creating a playable demo putting to test my programming and design skills.

To do so I have created a videogame in which I explore some fields of the videogame development industry, comparing different algorithms and techniques and putting them into test, addressing its advantages and disadvantages as well as its implementations problems and how I solved them.

On the first hand, I explore the world of design in which I create a basic storyline inspired by the Chernobyl HBO miniseries. **[1]**
You play the role of a soviet KGB agent that has been called to help in a top-secret important rescue mission. Some scientists have been trapped in a strange dimension and you are the chosen one to save them.

On the other hand, we encounter a classic first-person shooter with a three main statistic-based progression: Strength, Dexterity and Intelligence. Each of one will upgrade the characteristics of the player through the progression of the story and will transition from carefully slow gameplay into a fast and extremely fun experience.

Regarding programming, we find a totally procedural random dungeon generation, where different dungeon layouts generate each time, we play adding monsters that we should fight in order to save the soldiers. These enemies have different behaviors guided by a State's machine and a pathfinding AI which allows them to target the player among the dark corridors of the different dungeons and worlds. Other systems like a dialogue AI, achievement or quest system can be found which give more depth to the gameplay and story to give an overall better experience to the player.

This is mainly a technical demonstration of my skills on programming so the game assets regarding aesthetics, sound and other animations come from third-party people. We can see the credits to this people in the credits section at the end.
The art is low poly based which allows us to focus the insight of the project in the gameplay, physics and other playable elements rather than the artist aspect of videogame development.

Chapter 1. “General concepts”

State of Art

The current videogame industry is a constantly changing world surrounded with innovation, technology, art and lots of new challenges to overcome. Finding new algorithms, methods of optimization or the improvement for the latest AI available are some of the topics being talk about in the current state of the market.

PAST

Because we cannot conceive videogames without technology, from the very begging the videogame world has been constantly growing as the technology advance. The original Pong put in comparison with the latest AAA title is the best example. The nonstop improvement of game developers creating new engines, algorithms and methods to create are the things that allowed this crucial progress.

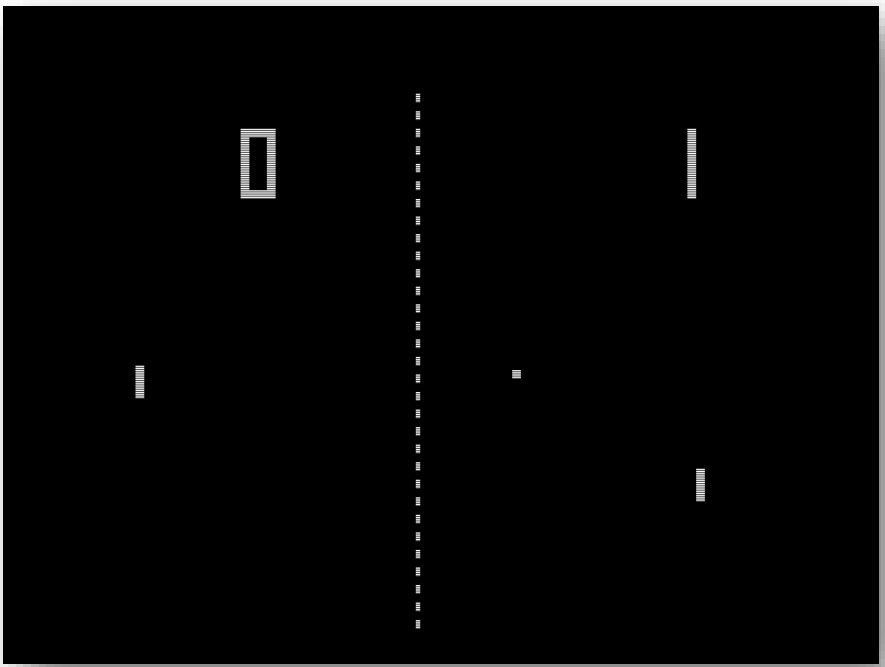


Figure 1.1: Pong. Bumm13, en.wikipedia.org, Public domain [2]

PRESENT

Not only videogames have become a symbol of progress during the course of time but also a way to express feelings, political concern, a way to communicate and interact with people around the world or even teaching at schools.

Figure 1.2: GRIS. Nomada Studio, Devolver Digital. [3]

During the current pandemic situation around the world the consume of videogames has been increased everywhere leading to a bigger than ever profit. One of the reasons if the isolation we are suffering with the constant restrictions to travel and limited mobility. These videogames are a way to experience something new and "travel" to another world for a few hours putting your mind somewhere else and focusing on having a good time.

It is known that videogames are a big business, and this will not change in short period. To put things in perspective in this current year 2021, video game industry in U.S grew 27% and made up to 57 billion in revenue surpassing movies and music combined.

FUTURE

With these advances in technology new ways to develop videogames are being develop by technological companies. One of them that is talked about recently is Virtual Reality. This new technique allows the player to experience an immersive world that tricks the brain into thinking you are actually inside the videogame.

Figure 1.3: Current generation of Valve Index VR Kit. Valve. [4]

It is hard to predict where these nonstop advances will end but for now, it is secure to say that we have an exciting future ahead for the players and developers around the planet!

Objectives

The objectives in these kinds of project can be endless, so I chose some fields that match my interests or attention and some that would connect with the purpose of this final degree project studying and implementing the most technical part of the videogame's development such as AI or high-end procedural algorithms. The objectives for this project are the following:

- Study the different fields of the videogame industry comparing techniques, methods and algorithms as well as the implementation of those:
    - **Storyline:** Create a basic storyline for the gameplay.
    - **Gameplay:** Design, study and implement an attractive gameplay mainly focused on a first-person shooter.
    - **Level design:** Design, study and implement the level design of the scenery focused on the use of a procedural random generation.
    - **Camera and movement:** Create a fluent and smooth movement based on physics with a first-person camera.
    - **Physics:** Study and implement a physics system for the videogame.
    - **Logic:** Study and implement all the logics necessary for the videogame.
    - **Procedural Random Generation:** Study the different methods for a procedural random generation and the current situation in the videogame industry. Propose a comparative between different algorithms checking advantages and disadvantages. Finally choose the best fit for the project and implement it.
    - **Artificial Intelligence:** Study the AI behaviors in videogames, select the best strategy, create an AI for the enemies and make some improvements until enemies feel real.
    - **Other systems:** Implement other systems and methods which help interact with the demo and make the videogame playable such as controls, interactions, buttons and settings.
- Create a final **stable build** with the objectives proposed.
- Implement different methods to improve **performance** in different platforms and systems.

Overview (Technical perspective)

Description

Videogame technical demo in which the author pretends to illustrate his technical skills in some fields such as Procedural Random Generation, Artificial Intelligence, Physics and Logic. Moreover, study and compare the different methods proposed and discarded addressing the advantages and disadvantages between them given the current industry state.

Theme

You are a soviet soldier in a distant dystopic future in which you are being hired by the Russian government to perform a rescue mission on a parallel universe infested with dangerous creatures and the strangest infinite dungeons.

Genre

First-Person Shooter, Story driven, Single player.

Platform

UWP (Universal Windows Platform), Linux, Mac OS.

Target audience

FPS players and Indie videogames developers.

Latest version

Demo pre-presentation 1.0. (Last update: 20/08/2021)

Elevator Pitch

"A classic low-poly FPS Indie game with zombies where your biggest preoccupation is whether kill them soft and gently or blow everything up."

Chapter 2. “Design”

Storyline

In a dystopic future where has been discovered parallel universes, the race to conquer and dominate these worlds is the current preoccupation and goal for most of the powers that reign over the earth. Russia, one of these biggest powers has been trying to access one of these parallel universes but has lost communication with its soldiers inside.

You, as a government hired soldier must help those trapped comrades and go back alive to earth. The first step is to recover your equipment from the shooting range in a laboratory lost in the vast Russian continent. A comrade called Yago is the head of the laboratory. It tells you to take your equipment from the shooting range and practice a little bit to refresh your abilities on the field.

Figure 2.1: “Laboratory”, The Cost of Lies.

After talking with him, you will go to the shooting range where your equipment is. After picking it you can practice shooting some targets.

Figure 2.2: ”Shooting Range”, The Cost of Lies.

When you are ready to go, Yago opens the portal to the other dimension with the help of Stefan, another scientist there, here is where your adventure truly begins: Dangerous random generated dungeons, beautiful sceneries and an amazing gameplay is left for the player to discover the secrets these new worlds have to offer and how he can save his comrades.

Figure 2.3: “Encampment”, The Cost of Lies.

Characters

All characters have their own specific reason to appear in the game, from giving you quest to sell you ammunition as well as its own personalized dialogue system the player can interact with. Here are all the main ones that are introduced in this demo, its characteristics and a summary of their role on it:

Main character

- **Name:** <UNKOWN>
- **Photo:** <UNKOWN>

- **Occupation:** KGB Specialist Agent (KGB)

  Summary: This is the main character of the story, a KGB agent that has been hired by the Russian government to accomplish a difficult mission involving parallel universes and lots of mysteries.

Yago

- **Name:** Vladimir, Yago.
- **Photo:**

Figure 2.4: "Yago". The Cost of Lies.

- **Occupation:** Head Scientist (KGB)

  Summary:
  Head of the main laboratory for parallel studies in Russia. The guy who contacted you regarding the incident.

Stefan

- **Name:** Petkov, Stefan.
- **Photo:**

- **Occupation:** Scientist (KGB)

Summary:
Scientist at main laboratory for parallel studies in Russia.

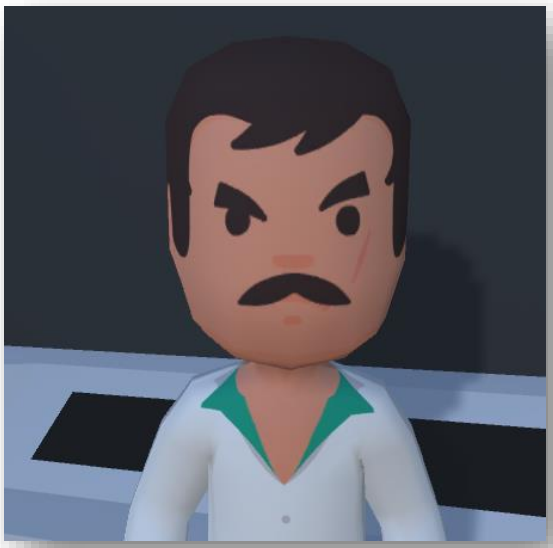


Figure 2.5: "Stefan", The Cost of Lies.

Rubén

- **Name:** Toutanov, Rubén.
- **Photo:**

- **Occupation:** Captain (Russia Army)

Summary:
Captain of the second brigade at the Russian army.
Currently in a mission on an Unknown parallel world.

Figure 2.6: "Rubén", The Cost of Lies.

Álvaro

- **Name:** Lergov, Álvaro.
- **Photo:**

- **Occupation:** Soldier (Russia Army)

Summary:
Weapon specialist of the second brigade at the Russian army.
Currently in a mission on an unknown parallel world.

Figure 2.7: "Álvaro", The Cost of Lies.

Asier

- **Name:** Kiprotich, Asier.
- **Photo:**

- **Occupation:** Soldier (Russia Army)

Summary:
After a tragic mission on a parallel universe.
Asier is out of combat and you must take his position in this difficult mission while he recovers.

Figure 2.8: "Asier", The Cost of Lies.

Enemies

The enemies introduced in the demo are zombies. These types of enemies are located inside the different dungeons and have different behaviors depending the situation they are in. We have two differentiable ones:

Standard Zombie

- Photo:

- Damage: *15 - 25*.
- Speed: *6*.

Figure 2.9: "Zombie", The Cost of Lies.

Mini-Zombie

- Photo:

- Damage: *10 - 20*.
- Speed: *8*.

Figure 2.10: "Mini-Zombie", The Cost of Lies.

Characteristics and behaviors

One of the key aspects of a PVE game is how it feels to fight versus the AI or enemies in this case, a good design of enemy's movements and attacks is crucial to get the player engaged and challenged at the same time. Here are some characteristics and behaviors of the previously mention zombies:

Unique enemies The first point when designing a variety of enemies is to provide unique features for each one of them. In this case we encounter two different types of zombies. The standard zombie: A classic slow zombie with a damage range from 20 to 30 that patrols the dungeon corridors at 6 speed and another smaller zombie which is up to 8 speed faster than the previous one but has the downside of a lower damage in return to balance it.
This would make the player try to evade the second due to its dangerous speed and take care of the first and its incredible damage, both make a combination between force and speed which will induce a sense of challenge when the player is fighting them inside the dungeons.

Random patrolling Another interesting point is the randomness on the enemies when it comes to patrolling. If the enemy does not have the player in range for him to chase, he will patrol through the dungeon corridors randomly. This will create for example, extremely unusual situations in this dungeon-based kind of videogame, usually it is expected for the player to enter a room and encounter some monsters to engage and fight, however the patrolling implemented in these dungeons is random, which means the player can be looting in a room and a monster can randomly appear and start chasing him.

Player chasing The zombies will chase the player until one of these conditions is fulfilled, either it reaches the player and its able to attack or the player has run out of his range and it is not worth it for him to chase. In this second case the zombie will stop chasing the player and will return to patrol the current area he is in. The zombie will always choose the optimal path to chase the player as it has an A* algorithm for path finding. If the zombie encounters an unreachable zone in which the player is in, for example a tall obstacle where it cannot climb, then the zombie will wait for the player in the closest available position and continue to chase the player.

Gameplay

The exploration of the gameplay field in videogames has led to an incredible, smart ways to propose each year new ways to play, new mechanics to explore and better forms for the player to interact.

Therefore, this is considered by most of the industry one of the key aspects when developing a videogame. Because it can have an incredible story, awesome new-gen graphics and an insanely complex AI but in the end when it comes to playability, the gameplay must feel smooth and fun otherwise is a waste of time and money as the player will notice it in the first 5 min of play.

As a gamer myself this was the most difficult part in this project to be satisfied with. I have played a lot of videogames in the past and I know how it feels when a camera does not move exactly how you expect it to or how the player movement is not as accurate as you would think. That is why in this project, I developed and design a first-person gameplay involving a smooth raw camera movement, a quick fun and easy to understand combat system with some physics and some QoL features for the player and an interactive dialogue and quest system between characters.

The goal was to implement a gameplay in which my years of experience as a consumer in this industry would be satisfied and did not feel awful when I test it. In the end I was very satisfied with the final result and smoothness of it.

Gameplay Mechanics

When we talk about game mechanics, we refer everything related to the core gameplay of the game itself and what it is possible to do in game. For example, in an ARPG genre it is expected to be able to equip gear like a heavy armor or a sword, a puzzle game is expected to have different levels of difficulty and interactive elements on screen. In the case of first-person shooter this core mechanic will obviously be the weapons or guns, the shooting experience.

Figure 2.11: “AK-47 Rifle”, The Cost of Lies.

Unlike other videogame shooters, this weapon is not modifiable at any level. In this case I have created a progression system based in three statistics which provide different bonusses for the player which affect directly to the gameplay itself, we will talk about this in the next section. Now let me introduce the basic gameplay controls implemented:

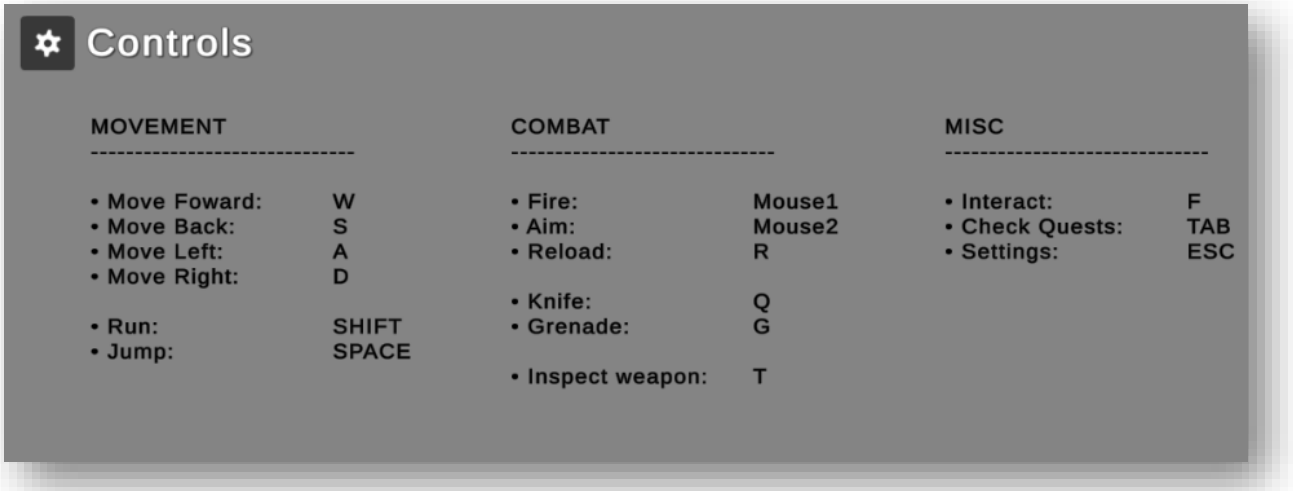


Figure 2.12: "Controls panel", The Cost of Lies.

Movement

The basic movement through the game. It is performed by the classic AWSD controls which allow us for example to get the input and control the camera rotation as well as the player's transform. The basic movement's speed can be increased by purchasing a dexterity point on Alvaro's shop, each point gives 5% increased speed.

- Keycode: **A, W, S, D**

Run

In relation to basic movement, we have the running ability, it allows the player to increase the speed of the character by 75%. It disables the ability of shooting when moving and can be improved by purchasing a dexterity point at Alvaro's shop, increasing the running speed 5% for each dexterity point available.

- Keycode: **LEFT SHIFT**

Jump

Finally, as the first set of controls we have the jump, this ability allows the player to jump up to 6 units and can be upgraded by strength at points 25 and 50 with a final hard capped value 10.

- Keycode: **SPACE**

Fire

The main ability for combat in the videogame. When pressed the player releases projectiles that come out of his weapon dealing 25 of damage to enemies. It has a random patron in which each bullet has a random spawn and a cadency that can be increased by the intelligence.

- Keycode: **Mouse 1**

Aim

The second main ability for combat. When hold pressed the view of the current equipped weapon changes into a more versatile non-recoil version in which the player has the ability to have perfect recoil control and a gun sight for better aiming at targets.

- Keycode: **Mouse 2**

Reload

When the current weapon runs out of ammo the reload comes into action, reloading the weapon adding to the current magazine ammo the total amount of ammo per magazine and reducing the total ammo available by the same quantity. The reload speed can be improved by 3% each time a dexterity point is bought.

- Keycode: **R**

Knife attack

The basic attack of the main character. It throws a powerful attack to the enemies in front dealing up to 15 damage to 3 concurrent enemies. The player becomes vulnerable while performing the attack and it cannot be stopped once started.

- Keycode: **Q**

Grenade

The player throws a devastating explosive which causes an explosion and deals 100 damage to the enemies in range. The player is only permitted to carry one grenade per time and can be bought as many times as he wants at Alvaro's shop on the ammunition package.

- Keycode: **G**

Inspect weapon

This special interaction between the player and the weapon allows the user to inspect the current weapon performing an animation. This is purely esthetic and has barely any implications in the gameplay, it cannot be stopped once started.

- Keycode: **T**

Interact

The basic button to interact with other character and the environment in the videogame. It allows the player to pick up ammunition, interact with other player or for example open Alvaro's shop at the encampment.

- Keycode: **F**

Check quests

This UI interaction grants the player with the ability to check the current quest status every time he wants by pressing a button. It shows the current quests available for 3s and after that it automatically closes by itself.

- Keycode: **TAB**

Interactive Elements

We can find diverse interactive elements around the different sceneries and worlds, some of these elements are purely cosmetic elements, others have the purpose of giving gold to the player or unlock certain secrets when touched, we called these interactive elements. In this case we have two different types:

Easter Eggs

The Easter eggs are cosmetic collectibles that are hidden one per area in the videogame. They gave the player 500 gold per unit and are accompanied by a classic quote from other videogames such as Bioshock [5] or Path of Exile [6]. The mission for these secrets can be consulted on the quest UI interface pressing TAB and it is automatically updated when the player touch one of these secrets.

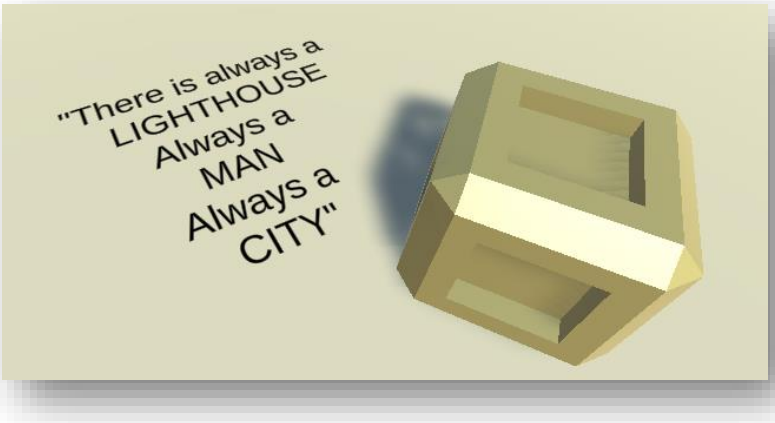


Figure 2.13: "Golden collectible", The Cost of Lies.

Progression and upgrades

A good gameplay with incredible mechanics and a fun gun system will feel empty for the player if he has no upgrade system or a way to feel he is progressing somehow. That is why I created a statistic-based system where the player can interact buying upgrades with gold, a currency obtained killing monsters and discovering secrets through the map. In this progression there are three elements to take into consideration:

Figure 2.14: "Alvaro's Shop", The Cost of Lies.

The Strength A buyable stat in Alvaro's shop which allows the player to increase in-game health ten points per time bought, it has a cost of a thousand in-game gold currency, and it can be bought up to 50 times which will result in six hundred total health for the player in comparison of 100 health in the begging. This will make the player feel safer when exploring the dungeon and fighting monsters.

The Dexterity A buyable stat in Alvaro's shop which allows the player to increase in-game movement speed by 5% and reload speed by 3% per time bought, it has a cost of a thousand in-game gold currency, and it can be bought up to fifty times which will result in 250% increased movement speed and 150% increased reload speed in total for the player. Dexterity stat is incredibly important in making the gameplay smooth and fun, the lower cooldown when reloading and a faster movement speed allows the player to feel smooth control of the character and an overall better experience.

The Intelligence A buyable stat in Alvaro's shop which allows the player to increase in-game cadency of fire of all weapons by 5% per time bought and upgrades all magazines by adding one bullet of capacity, it has a cost of a thousand in-game gold currency, and it can be bought up to fifty times which will result in 250% increased cadency for all weapons the player owns. The increased cadency and upgraded magazines will increase the speed in which the player kills monsters, so it will in the end means faster gain of gold and progress through the levels.

Faster reload and movement, more cadency of fire and a bigger health pool, all this statistic makes the gameplay to feel incredible better as the player progress through the story. It also induces the player a sense of risk every time he enters a dungeon, as he might be saving money for the next upgrade but if he dies inside the dungeon his gold will be reduced to half. Also, this system makes the different golden collectibles hidden around the worlds an attractive for the player, as they give each one five hundred gold on discover.

Level Design

Level design is one the biggest challenges of developing a videogame due to its complexity and the number of hours you must spend design each one of the level layouts. The goal for this field was to be able to create an algorithm that does this for us automatically and generates a random level each time we cross a specific trigger for example a portal to another dimension or a door.

Random generation with pre-built rooms

The first approach I contemplated was a completely random dungeon level system with pre-built rooms to choose. It was a good method for small games with almost cero complexity in programming, I rejected the idea because it also implied building each room with decoration, enemies and all kind of other elements. Here is a photo of what the early results looked like and the problems and downsides I have encountered when I first tested it:

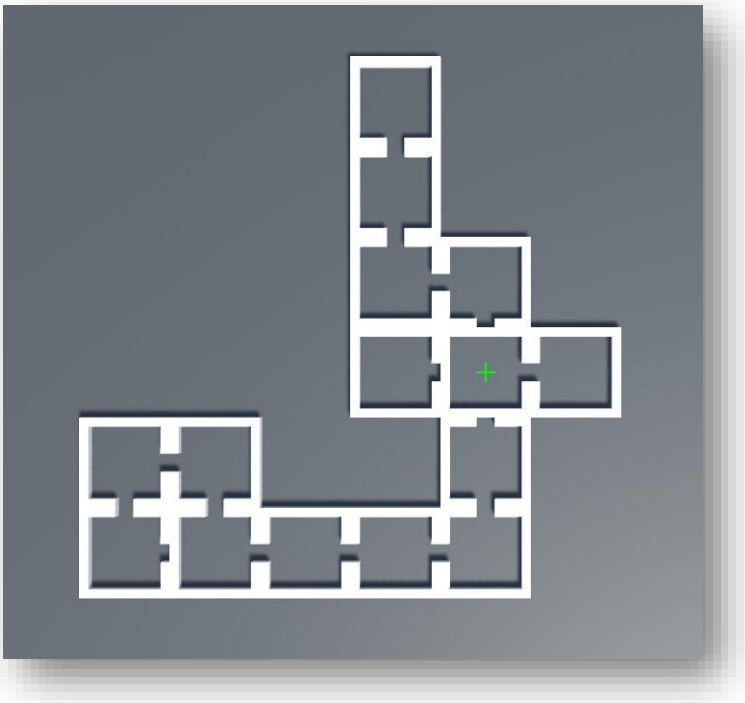


Figure 2.15: “Random Algorithm v1”, The Cost of Lies.

At first sight it may look convincing, well design dungeon but if we look closer, we can see for example, in the room 3 exits are blocked due to the randomness of the generation and it blocks the entrance resulting in a 2 rooms dungeon. After fixing a couple bugs and errors like that one, I came up with a second version where everything looked promising.

Here is a photo with the results of this second version where we can see the entrance bug fixed:

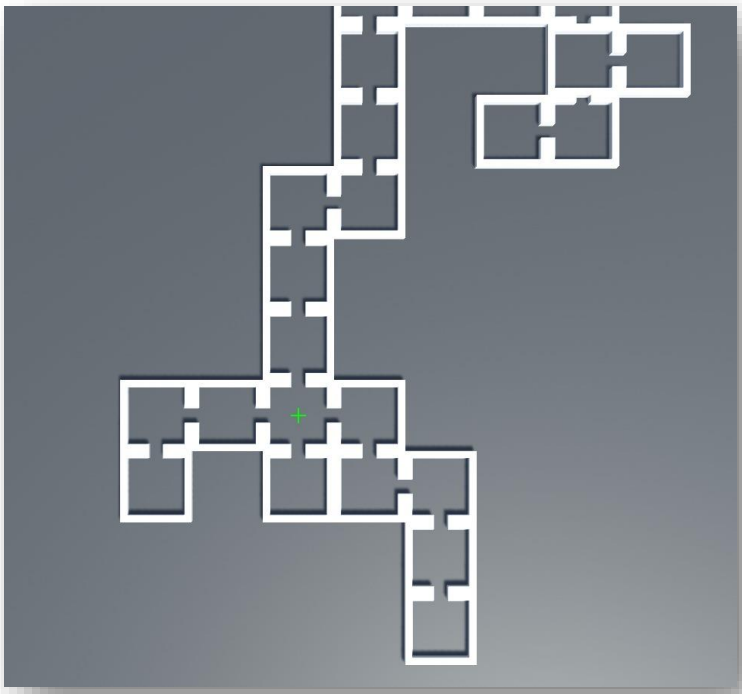


Figure 2.16: "Random Algorithm Fix", The Cost of Lies.

Once again looking at the result we can clearly see another problem in this random generation, the length of the dungeon and the unbalanced routes. Some results show a dungeon with 10-12 rooms which are the ideal result expected, but some iterations are giving us around 25-30 rooms or 5.

The disparity is not great for our demo, as we want an equal number of rooms for each level especially on the first levels where the player is still getting used to the gameplay.

The solution to this was the implementation of **Binary Space Partitioning**. We will talk about it in the section about programming and how I solved this level design problem, check the final result:

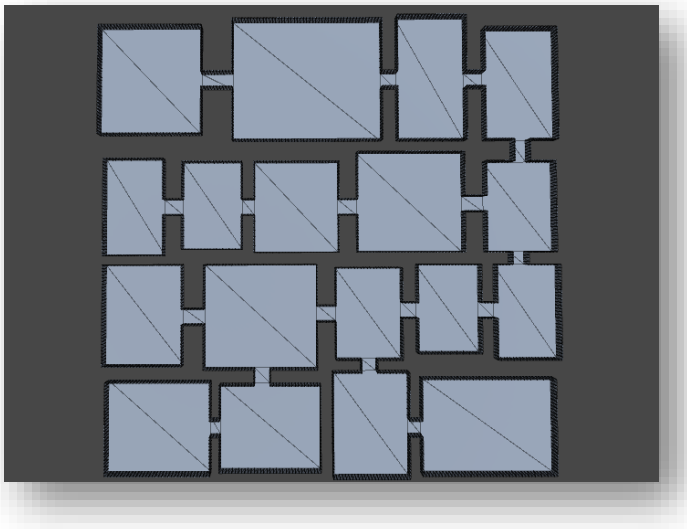


Figure 2.16: "BSP", The Cost of Lies.

Game Aesthetics

The selection of game assets for this project was a challenge rather than something simple because I knew this was not a doable aspect of my project and I had to take some assets from third parties in order to put some decoration and color to the demo. The final decision was to pick low-poly assets which I think they fit perfectly for this purpose.

Low-Poly Art

The **low-poly art** is a technique used mainly in videogames to generate textures and models with lower number of polygons this allows the game to run smoother and optimize for performance. Nowadays this has been a mode between indie game developer that does not have that many resources and need something colorful and different.

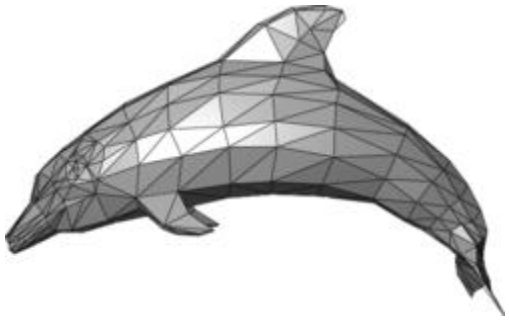


Figure 2.16: "Delphin low-poly", Wikipedia [7]

Screenshots

Here is a compilation of screenshots showing the art in-game:

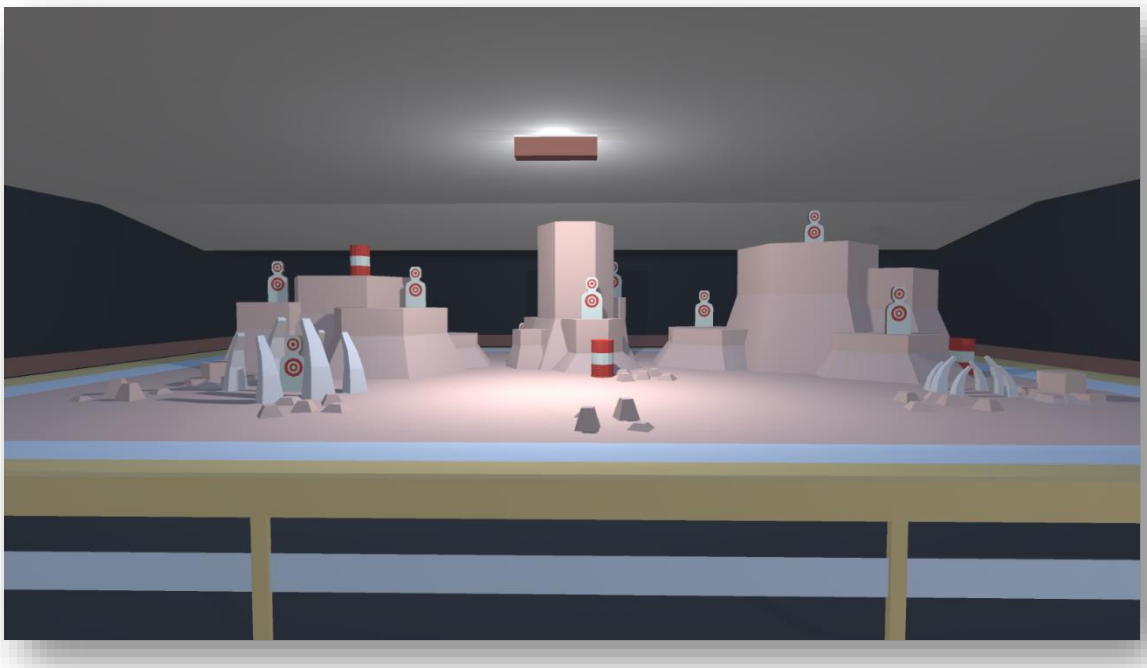


Figure 2.17: "Shooting range", The Cost of Lies

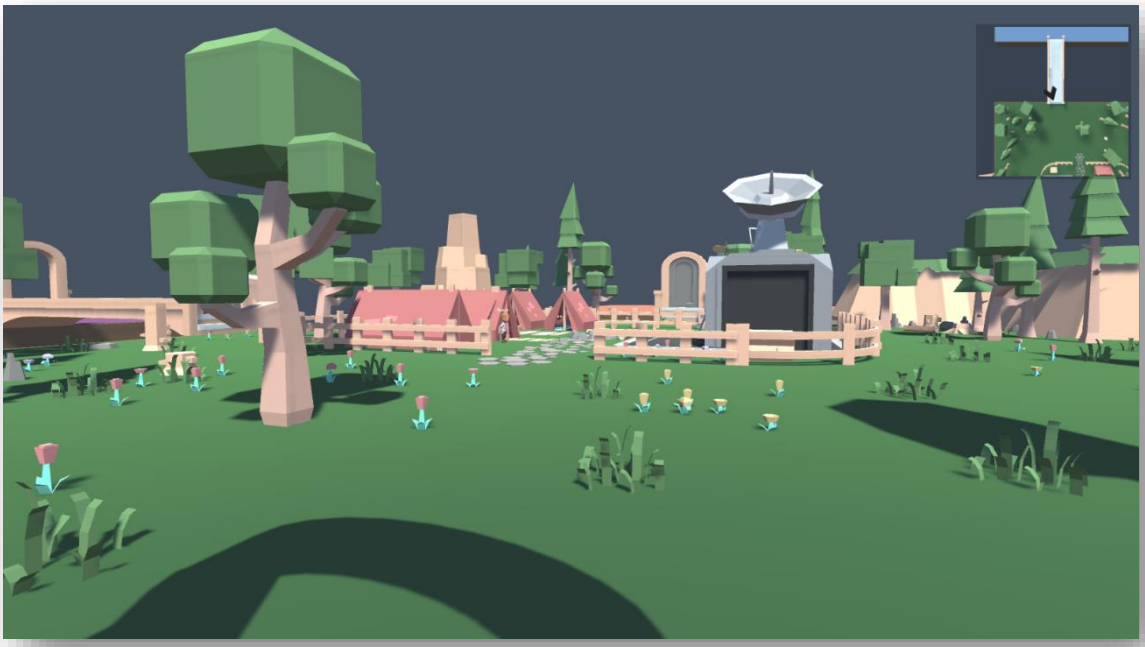


Figure 2.18: “Encampment”, The Cost of Lies

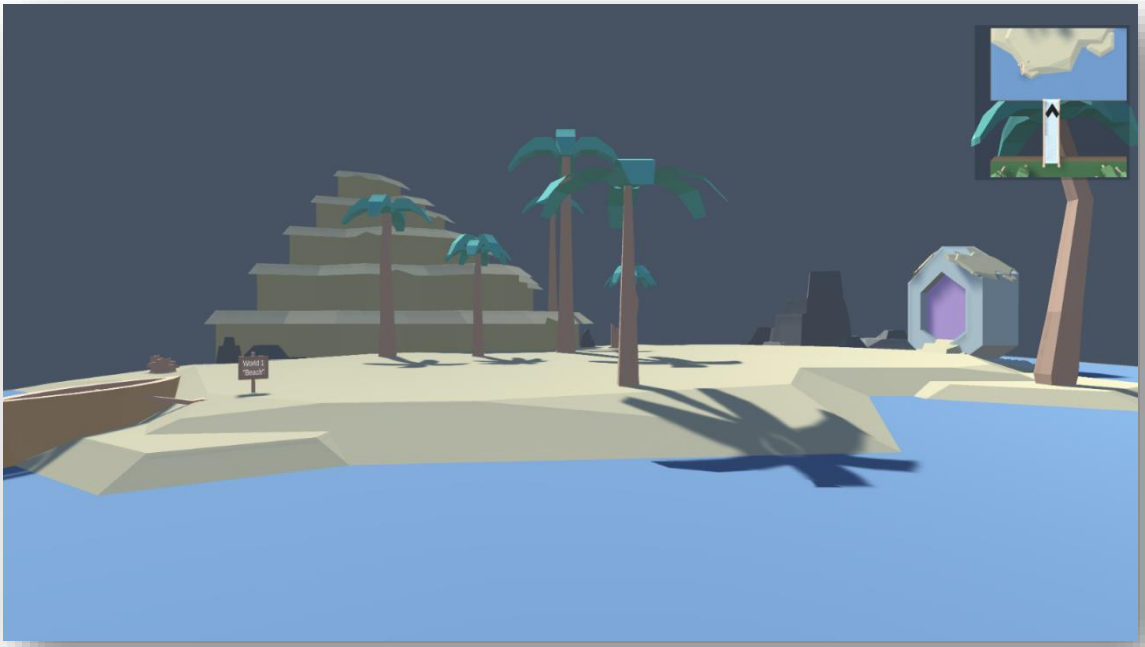


Figure 2.19: “Beach”, The Cost of Lies

Figure 2.20: "Gunplay version 0.5", The Cost of Lies

Figure 2.21: "Gunplay final version", The Cost of Lies

Chapter 3. “Programming”

Camera and Movement

In this section we are going to talk about the camera and movement implementation in the game, the programming structure and methods used to create a camera and the character movement. As I commented in the last chapter, I am a consumer of this industry and I guided myself by how I expect the controls and feelings to react.

Camera

The camera controller is the game object that controls the view following the player depending on the mouse input and has the objective of rendering all the scenery and game objects. Here are the three key elements used in the demo:

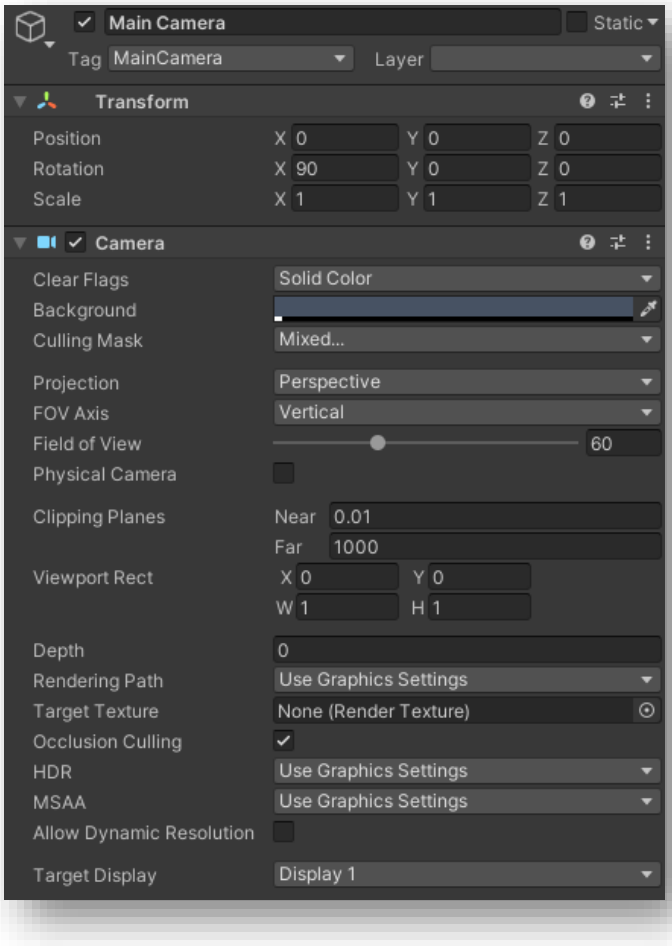


Figure 3.1: “Main Camera”, The Cost of Lies.

Culling Mask

List that give us the power to avoid or exclude some types of game objects from rendering. This is used for minimap elements like the position of the player or some elements that we do not want to render.

FOV

Allow us to control how much angle of vision the player has. By default, it is set on 60º but can be changed in game so the player has a better customized experience, some user can experience headaches from low field of view, so it is important.

Clipping Planes

The minimum and maximum distance the camera is rendering objects. This is very important for the performance as we do not want to render all objects in the scene all the time due to it increase in cost. This is one of the methods studied and implemented after testing the performance of the application, we will talk in depth later.

The camera by itself will not do anything rather than render some elements in a static range, to follow the player and render the real view the player is expected to see we need a script to control how the view is rotating depending the mouse input and the character itself:

```
private void RotateCameraAndCharacter()
{
    // Get the raw input from the player.
    float rotationX = inputMapping.RotateX * mouseSensitivity;
    float rotationY = inputMapping.RotateY * mouseSensitivity;

    // Rotate camera, horizontal axes.
    transform.Rotate(0f, rotationX, 0f);

    // Rotate camera, vertical axes.
    if (rotationY > 0)
    {
        verticalAngle = new Vector3(Mathf.MoveTowards(verticalAngle.x, -90, rotationY), 0);
    }
    else
    {
        verticalAngle = new Vector3(Mathf.MoveTowards(verticalAngle.x, 90, -rotationY), 0);
    }
    player.transform.localEulerAngles = verticalAngle;
}
```

In this script we calculate the raw input rotation we are getting from the player's mouse and we multiply it by the mouse sensitivity to get a float value for the X axis angle. The rotation on X axis can be directly applied using the method Rotate() which allow us to rotate the current game object (In this case the camera) given a vector.

Next, we have the rotation on Y which is applied taking into consideration that we cannot go beyond 90 degrees when looking upwards or downwards, so we have to limit that camera behavior. We calculate the vertical vector angle given the input and the restriction and after that we apply the calculated angle to the player character body local angles.

Character movement

For the character movement we have to think about the physics that implies the movement of a given character. We have the force of gravity and the input the user is giving us through the keyboard. To take control of the character we have the character controller a unity object that allow us to take the control and works as a collider as well. The first thing to do as usual is receive the player's input from the keyboard:

```
// Get the raw input from the player.
float speed = inputMapping.Run ? speedRunning : speedWalking;
float hInput = inputMapping.MoveH;
float vInput = inputMapping.MoveV;
```

After taking this input we must check wether the player is touching the ground or not:

```
// Touch the ground.
if (charController.isGrounded && charVelocity.y < 0)
{
    charVelocity = new Vector3(0f, 0f, 0f);
    onGround = true;
}
```

Finally, if the character is on the ground and its moving (We receive some input) we proceed with the character controller and apply the movement. We must think we are working with physics here so we multiply the speed by Time.deltaTime which determine per video frame the current speed of the player, if we do not take care of this the systems where the application runs at higher fps, we will get higher speed and that is not the output we want.

```
// Player is on the ground and moving.
if (hInput != 0 || vInput != 0)
{
    // Movement settings.
    moveInput = transform.right * hInput + transform.forward * vInput;
    charController.Move(speed * Time.deltaTime * moveInput.normalized);
    onMove = true;
}
// Player is not moving.
else
{
    onMove = false;
}

// Apply Gravity.
charVelocity.y += gravity * Time.deltaTime;
charController.Move(charVelocity * Time.deltaTime);
```

At the end we apply the gravity to cover the case where the player is not on ground.

Movement through scenery and obstacles

Related to movement we have the topic of the scenery and the obstacles. Whether this could be seen as something trivial or simple this implies a lot of things when we talk about movement because it is essential another factor to take into consideration when programming or designing the players and AI systems of movement.

Two different perspectives

From the **player** point of view, we do not have too many concerns regarding collisions and pathing as the user will interact with the environment and take care of the obstacles he encounters, the colliders the character controller has are enough for it to interact and give feedback to the player about where to go. The only thing we should take care here is the physics involving those objects and the rigid body unity provides solves this issue, the problem comes from the enemy perspective.

The **AI** has a different approach when we talk about obstacles, because itself by default cannot avoid those obstacles, we must take care of those manually. The solution is path-finding algorithms such as A* and Dijkstra. In the next sections we will see the problems these algorithms have while implementing them and why it is extremely difficult to implement them correctly on a procedural generated world.

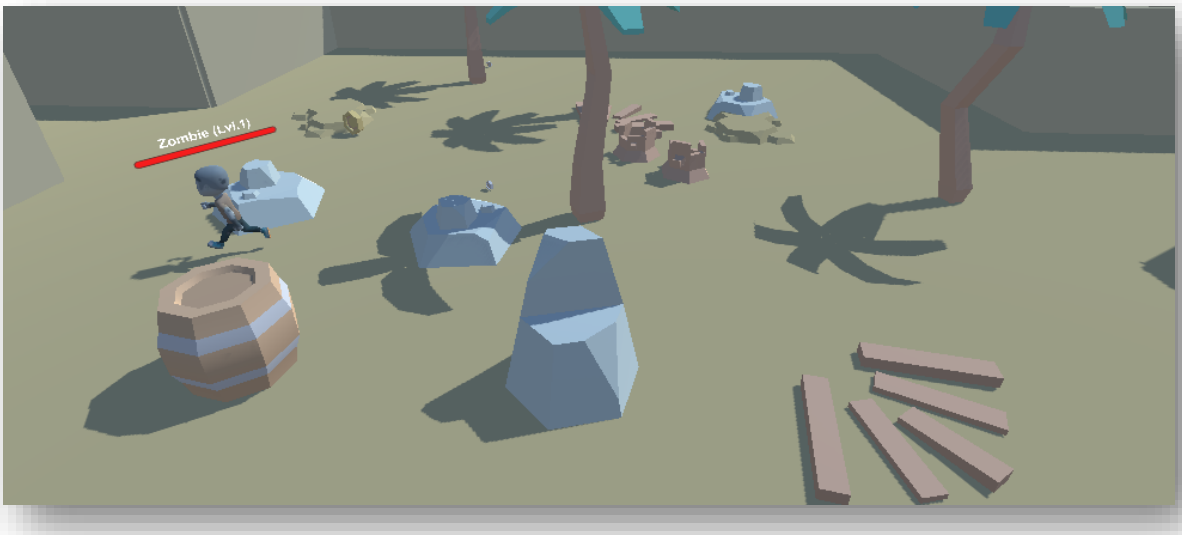


Figure 3.3: “Obstacles”, The Cost of Lies

Procedural Generation

Nowadays procedural generation in videogames is one the biggest challenges to overcome due to complexity and the problems it causes during runtime when taking into consideration the physics, colliders and other parallel systems running in the background. But what is in essence this technique?

Procedural Generation is a computing technique of creating some data automatically between the combination of pre-built assets, for example, some textures or game objects with the computer-generated randomness and processing power.
It can be used for texture generation, 3D models or even a complete layout which is what we are using them here for. The advantages it provides over other techniques are the randomness for less predictable gameplay, solves the problem talked earlier about level design and generation and overall gives the player the possibility of replayability as the player will encounter different layouts, enemies and loot each time the game is turned on.

In the following points, we are about to see three different implementations of a procedural the generation, the advantages and disadvantages of each one and what it is best for our context and type of videogame and why.

Random Algorithm

The first type of algorithm we are about to study is a complete random algorithm. In this first case we apply the basic concept of procedural random generation and construct X different rooms to generate a random layout with the randomness and computer power:

Algorithm	Random Generation

1. Pre-create X random rooms with the decoration and elements we desire.
2. Create the points mesh for each pre-created room, considering which other rooms can be generated depending on entrances and exits of each room and its positions.
3. Generate the first room.
4. For each possible point of generation, generate a room with the orientation matching entrance with exit.
5. Repeat step 4 until there is no more points available in the mesh.
6. Check each generated room for blocked entrances and nonsense exit points.

This will create an automatically generated layout with the pre-built rooms we give it through parameter allowing us to avoid most of the runtime problems regarding random generation, collisions and bake routes for AI. We will see why this is a huge advantage later when we encounter problems in other algorithms and methods.

Here is a photo result of a random generated layout:

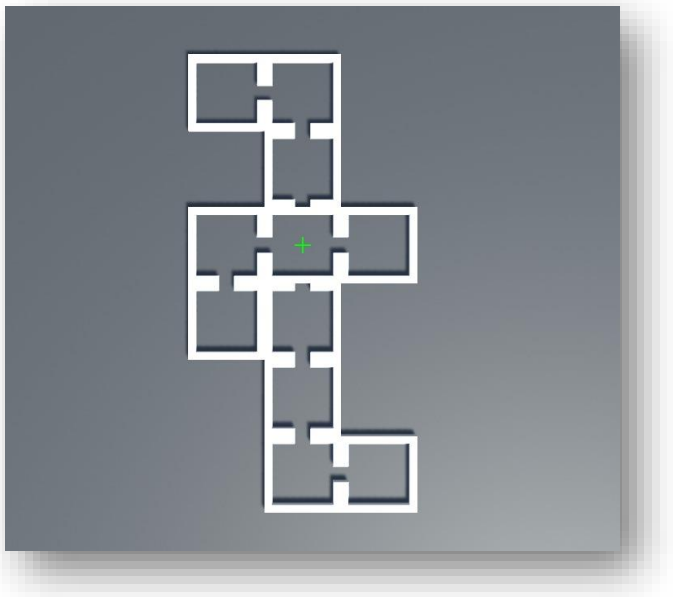


Figure 3.2: “Random Algorithm”, The Cost of Lies.

The problem with this algorithm comes when we talk about its cost, unwanted randomness and volatility of the generated layouts. As we can see in the picture, the generated layout is not 100% well generated, as some room’s walls are obstructing other entrances.

Another big problem is the lack of control we have on the layout itself due to the complete randomness. We cannot control how many rooms will be generated and if we try to do it, the complexity increases and varies a lot. We can encounter layouts with 10 rooms on it, 5 or even 35 which is not what we are looking for in our game as we pretend to increase the difficulty and complexity of the layouts each time the player completes a level to create a sense of progress and an increase of difficulty on the game.

Binary Space Partitioning

The second approach to the random generation is the use of a well-known algorithm used on optimization, the Binary Space Partitioning. But first let me introduce a little bit of history of why this algorithm and its importance in the videogames industry when it comes to 3D rendering.

Brief History

This algorithm was used as a 3D rendering resource in a one of the most influential and important games in videogames history the original 1993, DOOM.

Figure 3.3: "DOOM" (1993), id Software LLC, ZeniMax Media Company. [3]

Here is an article written by **Two-Bit History** about it:

"In 1993, id Software released the first-person shooter Doom, which quickly became a phenomenon. The game is now considered one of the most influential games of all time.

A decade after Doom's release, in 2003, journalist David Kushner published a book about id Software called Masters of Doom, which has since become the canonical account of Doom's creation. I read Masters of Doom a few years ago and don't remember much of it now, but there was one story in the book about lead programmer John Carmack that has stuck with me. This is a loose gloss of the story (see below for the full details), but essentially, early in the development of Doom, Carmack realized that the 3D renderer he had written for the game slowed to a crawl when trying to render certain levels. This was unacceptable because Doom was supposed to be action-packed and frenetic. So Carmack, realizing the problem with his renderer was fundamental enough that he would need to find a better rendering algorithm, started reading research papers. He eventually implemented a technique called "binary space partitioning," never before used in a video game, that dramatically sped up the Doom engine.

(...)

Initially, Carmack tried to solve this problem by relying on the layout of Doom*'s levels. His renderer started by drawing the walls of the room currently occupied by the player, then flooded out into neighboring rooms to draw the walls in those rooms that could be seen from the current room. Provided that every room was convex, this solved the VSD issue. Rooms that were not convex could be split into convex "sectors." You can see how this rendering technique might have looked if run at extra-slow speed in this video, where YouTuber Bisqwit demonstrates a renderer of his own that works according to the same general algorithm. This algorithm was successfully used in Duke Nukem 3D, released three years after* Doom*, when CPUs were more powerful. But, in 1993, running on the hardware then available, the* Doom *renderer that used this algorithm struggled with complicated levels—particularly when sectors were nested inside of each other, which was the only way to create something like a circular pit of stairs. A circular pit of stairs led to lots of repeated recursive descents into a sector that had already been drawn, strangling the game engine's speed."*

Two-Bit History, "How Much of a Genius-Level Move Was Using Binary Space Partitioning in Doom?" **[8]**

The algorithm consists in the process of dividing a given space in two given a random position in the space until some conditions are met, we store each partition in a BSP (Binary Space Partitioning) tree which will be useful later when we create the floor for dungeon.

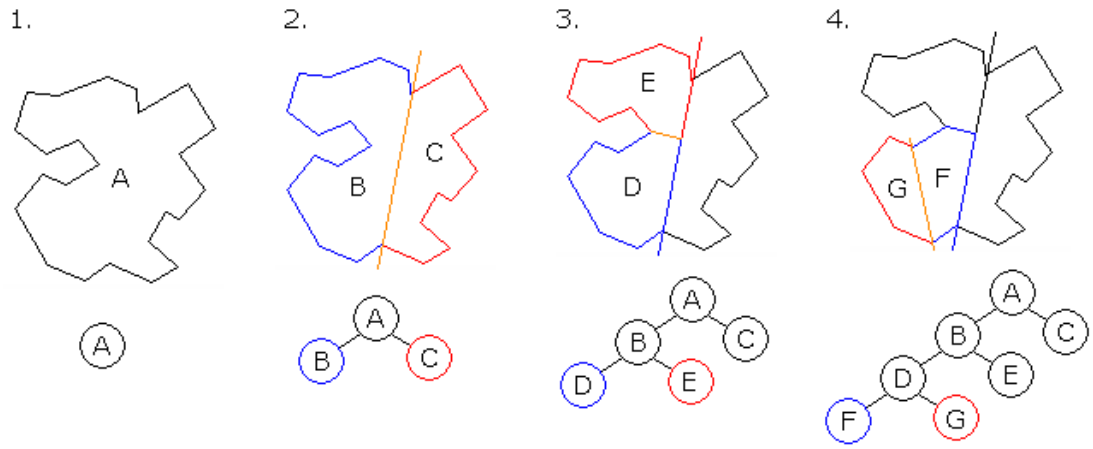


Figure 3.4: "Binary Space Partitioning", Wikipedia [9]

In this example we can see how given a space A, we divide it in two different sections B and C and store the division in a BSP tree as children of A. In the next iteration we do the same with the children B dividing it in half and creating their children E and D and storing them in the tree. Finally, in the 4th iteration we create F and G and store them in the tree.

If we would apply our algorithm to this example we would keep dividing until a minimum room width and length is met, so we do not keep diving indefinitely. This would result in the division of C rather than continue diving D and we would have 4 different zones.

After that we would create the corridors between the nodes of the same parent, this is when the tree becomes a powerful resource to use as we can perform a tree traversal in order to extract the lowest leaves and their parents and connect them as we advance through it.

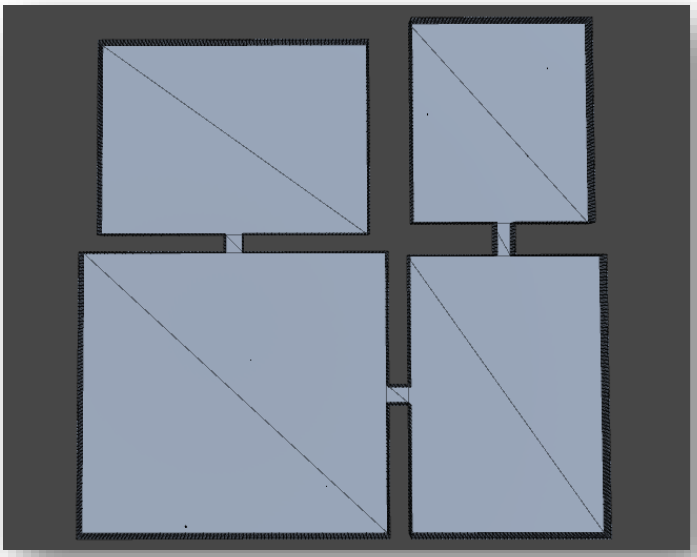


Figure 3.5: "BSP simulation", The Cost of Lies.

The algorithm implemented follows these steps:

Algorithm: Binary Space Partitioning

1. Define a space with a specific width and length.
2. Divide the space using a horizontal or vertical line at a random point.
3. Add the partition to a BSP tree, being the two resulting spaces children of the original.
4. Check if the new created spaces can be divided any further.
5. Repeat steps 2-4 until there are no more spaces that can be divided.
6. For every stored space in the tree create a room within the boundaries.
7. Perform an in-order traversal creating corridors between nodes of the same parent.
8. Choose the next youngest branch.
9. Repeat steps 8-9 until finished.

Once the algorithm has finished, we have a well-constructed empty random dungeon, but I wanted to push this algorithm as far as possible, so I added some steps to the algorithm to create the decoration and enemies of each level automatically being it random and dependent on how the player has been performing in previous levels. This last addition would finish the level completely without pre-designing a single room:

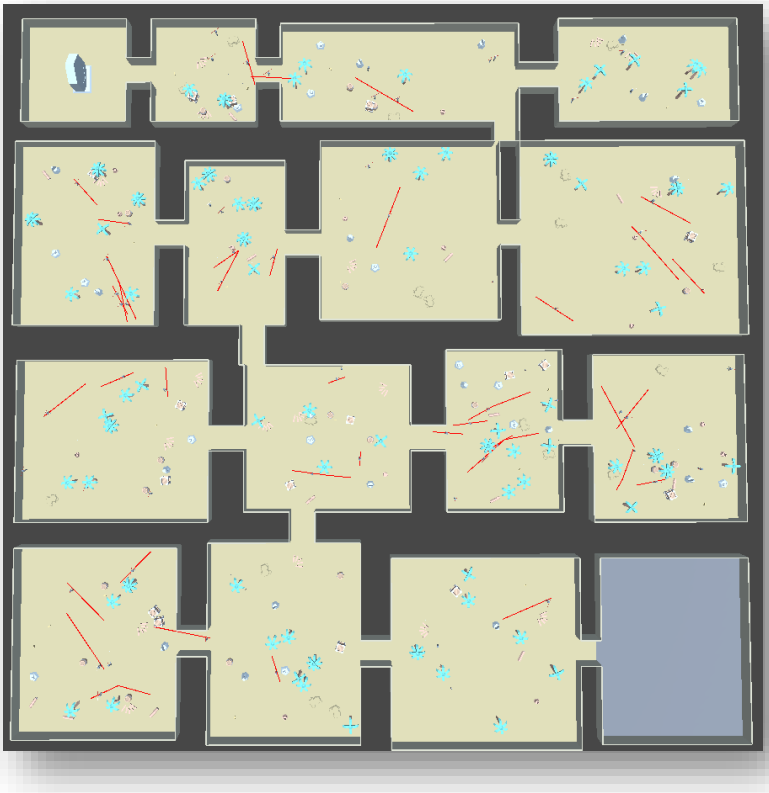


Figure 3.6: "BSP final result", The Cost of Lies.

Results and conclusion

The final results in the simulations when implemented the algorithm looked promising. A well-built dungeon that automatically creates the levels and their decoration, spawns the enemies and player as well as the portal to next level and some interactive objects to collect:

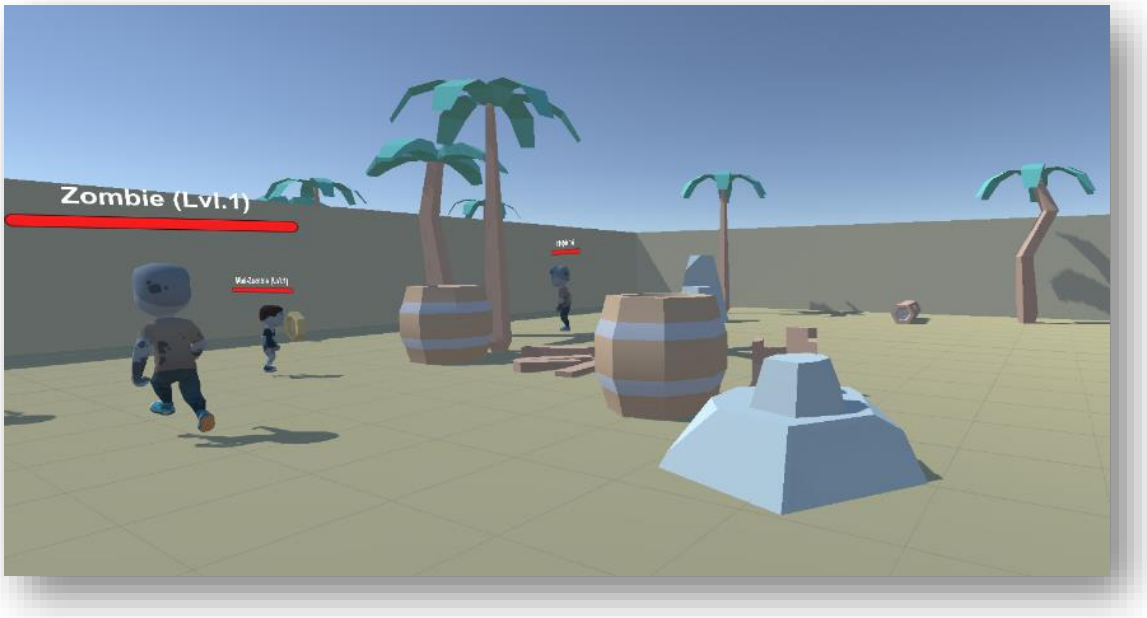


Figure 3.7: "Dungeon Room", The Cost of Lies.

However, there were still some problems that I was not aware in the begging such as baking the level at runtime and the implementation of a pathfinding algorithm in a randomly generated world.

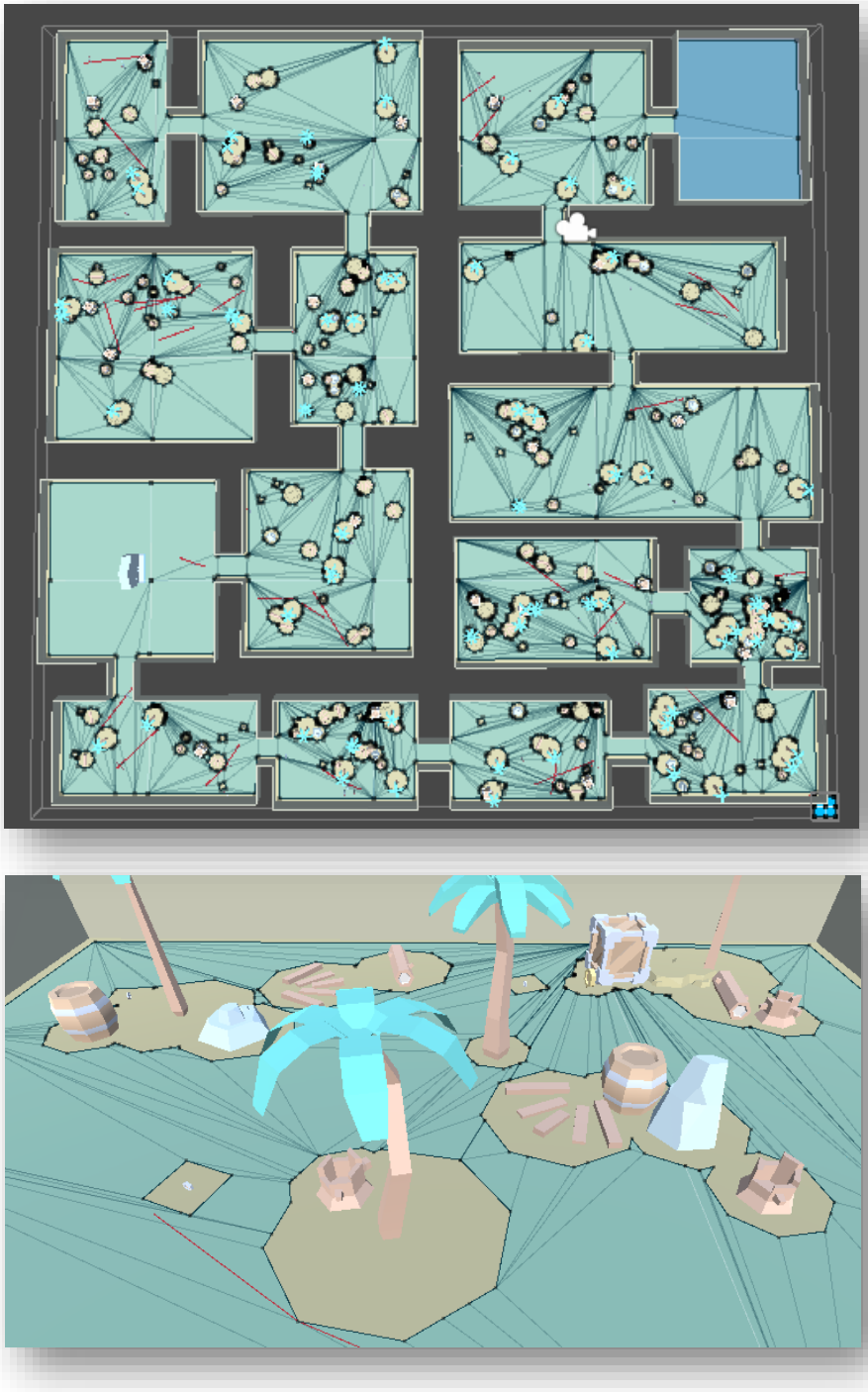
This is a well-known problem in Unity about generating Navmesh in runtime that has not been solved yet. You cannot directly bake the surfaces in which your AI is able to walk using a pathfinding algorithm at runtime it must be done in a pre-rendered scene and we are basically creating a labyrinth while we play.

This was not an easy problem to fix and after a lot of hours of testing and research, I came up with an extern experimental plugin provided by Unity. This plugin gives us the ability to use experimental surface scripts that automatically updates themselves at runtime which allows us to bake the level once it is created. Here is a link to a unity tutorial in which they talk about this:

https://learn.unity.com/tutorial/runtime-navmesh-generation

After implementing these scripts into the algorithm, the prefabs used to create the level and some intense testing, the level was baking properly, we finally have a walkable dungeon for our Navmesh agents.

In the following images we can see the baked areas and the different vertices to calculate the optimal path using a A* algorithm and avoiding all obstacles:



Figures 3.8 and 3.9: "Baked Areas", The Cost of Lies.

Artificial Intelligence

For this project I also studied the field of artificial intelligence in order to create a behavior for the enemies and their movement. In this section we are about to see the pathfinding algorithm for the navigation system implemented in the enemies and the state's machine that controls the enemy's behaviors towards the player.

After that we are going to see the results implemented in the dungeon that we have created before and which problems I have encountered referring real-time area baking and the obstacles in the dungeon.

Enemy Navigation system

When creating a moveable enemy, it is important for it to be able to chase the player and move around the scenery at its pace, the problem most of the times is the complexity of the scene or the lack of freedom these enemies have.
The solution is to implement a pathfinding algorithm that is capable of both moving the enemy creating a sensation of conscience and a real obstacle avoidance system that allows the AI to evade the different object in its path.

This navigation system consists of:

Walkable Areas

The navigation system collects data form the scenery in form of walkable area in which the different agents in the scene can walk, in our case these would be the enemies.

The NavMesh stores these areas as convex polygons to posteriorly calculate the path around the vertex of those polygons.

The walkable areas are built automatically from the geometry in the scene and the different obstacles in it. It basically tests the position of the agent on each position and stores that position if it fits. Once we have collected this information, we can find a path.

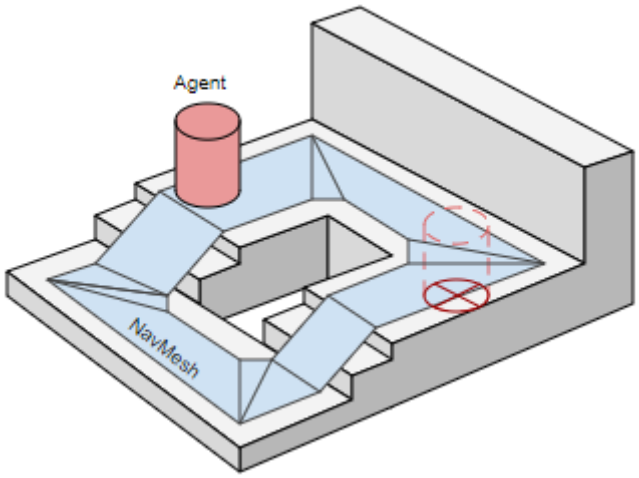
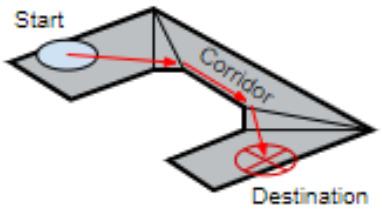


Figure 3.10: "Walkable areas, Unity. **[10]**

Path Finding Algorithm

To create movement, we give the agent two points, point A and point B and processing the data from the walkable areas it will create a path using the A* Algorithm creating the shortest path.



If we encounter any object or obstacle in the path, the agent will automatically avoid it as it has already calculated a route around that object vertices.

(We have obtained that data from the walkable areas before)

In the case where the agent can not find the route, it will go to the closest available point and reset the path. In our implementation the zombie stays in the closest point to the player to avoid it from scape when he is up in a tree or a barrel, for example.

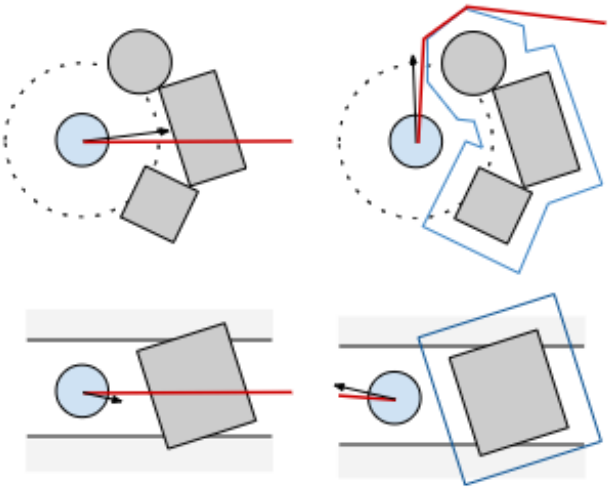


Figure 3.11: "Walkable areas, Unity. [10]

Example

Here we can see a real example during a testing simulation where we can see how the zombie uses the data from the walkable areas to create a path using a palm vertex to avoid collision with the object and reach the final point.

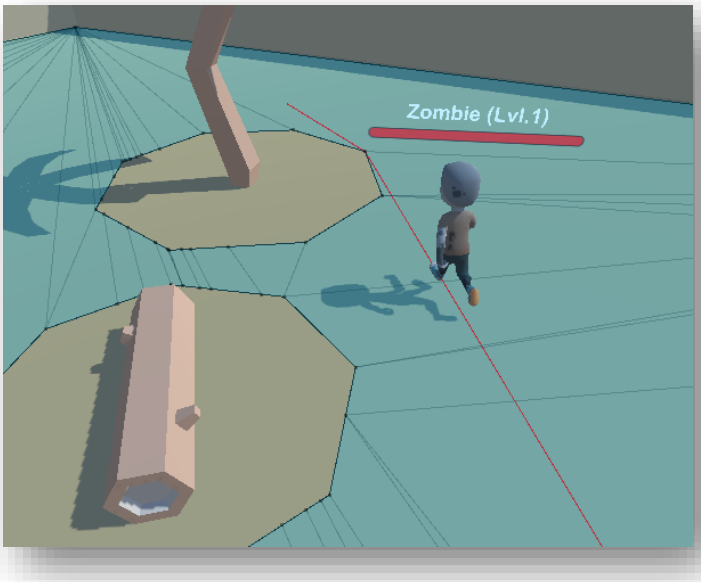


Figure 3.11: "Walkable areas", The Cost of Lies.

State's machine

The final goal of having an enemy AI is to make the player feel oppressed and challenged while feeling the enemies have conscience and they have some predicable behaviors the player can learn through his experience in the game.

The best way to achieve such thing is to create a state's machine for the enemies AI. In this case I have created a state's machine in which the zombie has 4 states: Idle, Patrolling, Chasing, Attacking.

```
        // Check for Sight and Attack range.
        playerInSightRange = Physics.CheckSphere(transform.position, sightRange, playerTag);
        PlayerInAttackRange = Physics.CheckSphere(transform.position, attackRange, playerTag);

        // States
        if (!playerInSightRange && !PlayerInAttackRange && !isWaitting)
        {
            // PATROLLING
            PatrolArea();
        }
        else if (playerInSightRange && !PlayerInAttackRange)
        {
            // CHASING
            ChasePlayer();
        }
        else if (playerInSightRange && PlayerInAttackRange)
        {
            // ATTACKING
            AttackPlayer();
        }
```

For the states Patrolling and Chasing we must calculate the path to set it to the agent:

```
    private void SearchWalkPoint()
    {
        // Calculate random point in range
        float randomZ = Random.Range(-walkPointRange, walkPointRange);
        float randomX = Random.Range(-walkPointRange, walkPointRange);
        path = new NavMeshPath();

        // Set a posible walkPoint
        walkPoint = new Vector3(transform.position.x + randomX, transform.position.y,
                                transform.position.z + randomZ);

        // Check if the path is complete
        if (NavMesh.CalculatePath(transform.position, walkPoint, NavMesh.AllAreas, path) &&
            path.status.Equals(NavMeshPathStatus.PathComplete))
        {
            walkPointSet = true;
        }
    }
```

Other systems

There are other systems that are working simultaneous to those previously mentioned, dialogue system, achievements system or the quest system. We are going to go through all of them doing a brief summary.

Dialogue's system

This system is in charge of the story and the interaction between the player and the other characters in the worlds. It provides the player with some texts to read and quest to complete as well as some other options and notifications.

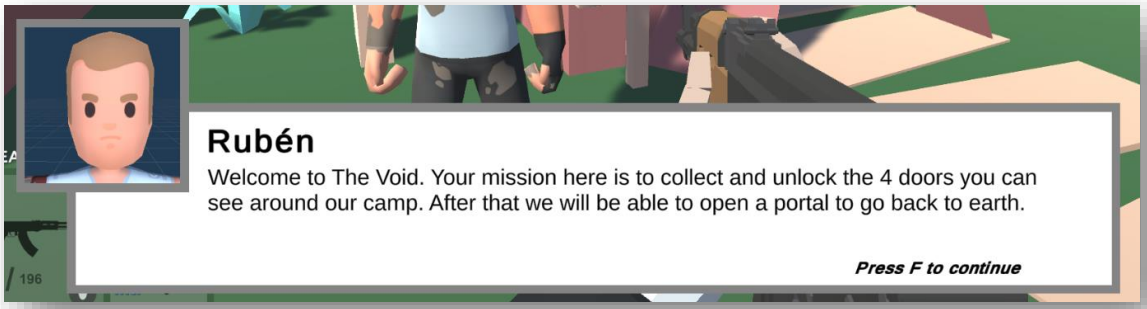


Figure 3.12: "Dialogue's system", The Cost of Lies.

Quest's system

With this system we allow the player to complete a follow the track of some of the quests in the game such as the main storyline and the collectibles around the map. In the case of the last, it adds an indicator of whether the collectible is collected or not.

Figure 3.13: "Quest's system", The Cost of Lies.

Results and comparative

Advantages and disadvantages

Here are the advantages and disadvantages of all methods and algorithms studied and presented above:

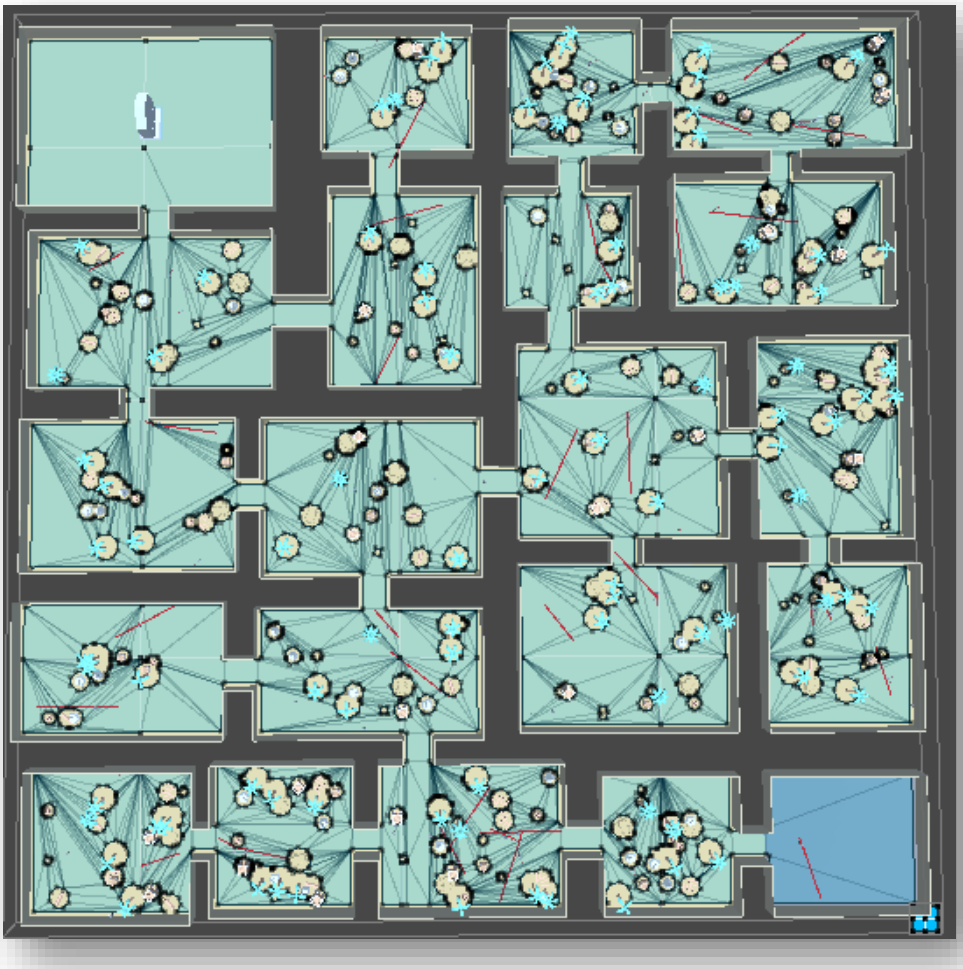
RANDOM GENERATION	
ADVANTAGES	DISADVANTAGES
Easy Implementation.	Unbalanced random number of rooms.
Avoid runtime problems.	Uncontrollable generation.
Pre-built textures and decoration.	Forced equal size rooms.
	Possible blocks and random errors.
	High computing cost. O(n^2)

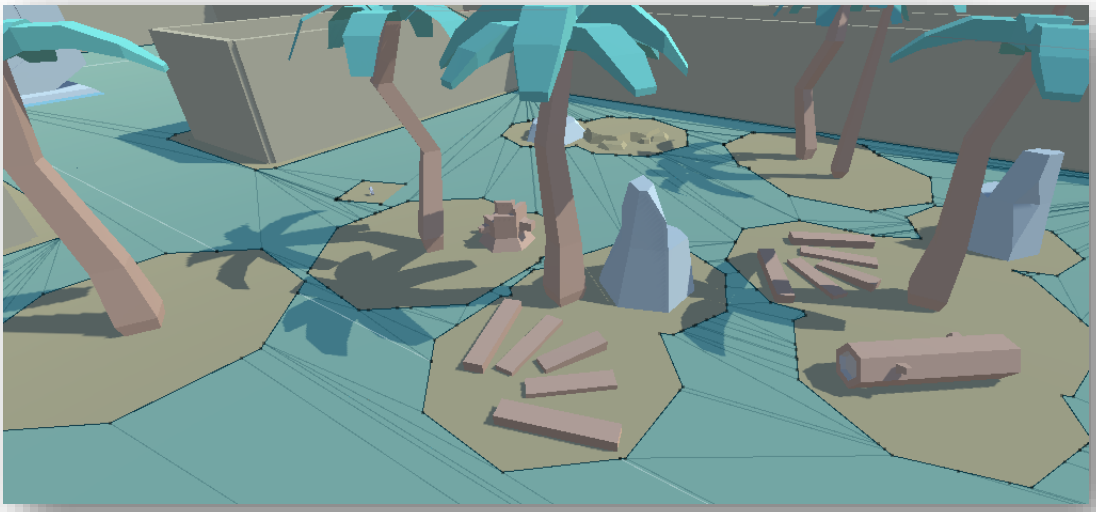
BINARY SPACE PARTITIONING	
ADVANTAGES	DISADVANTAGES
Controllable size and number of rooms.	Complex implementation.
Well-built random generation.	Runtime problems.
Lower computing cost. O($n * log(n)$)	
Customizable random decoration.	
Avoid designing hundreds of rooms.	

Results

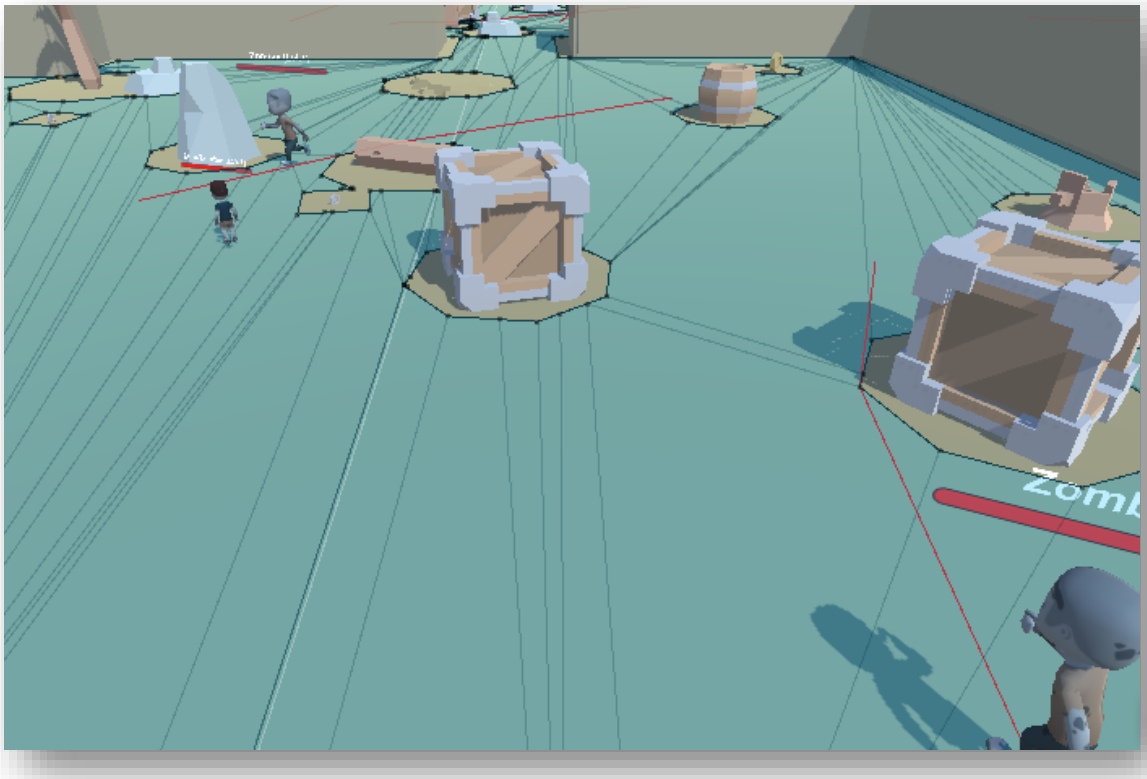
Here are the results of the algorithms and methods studied above after implementation:

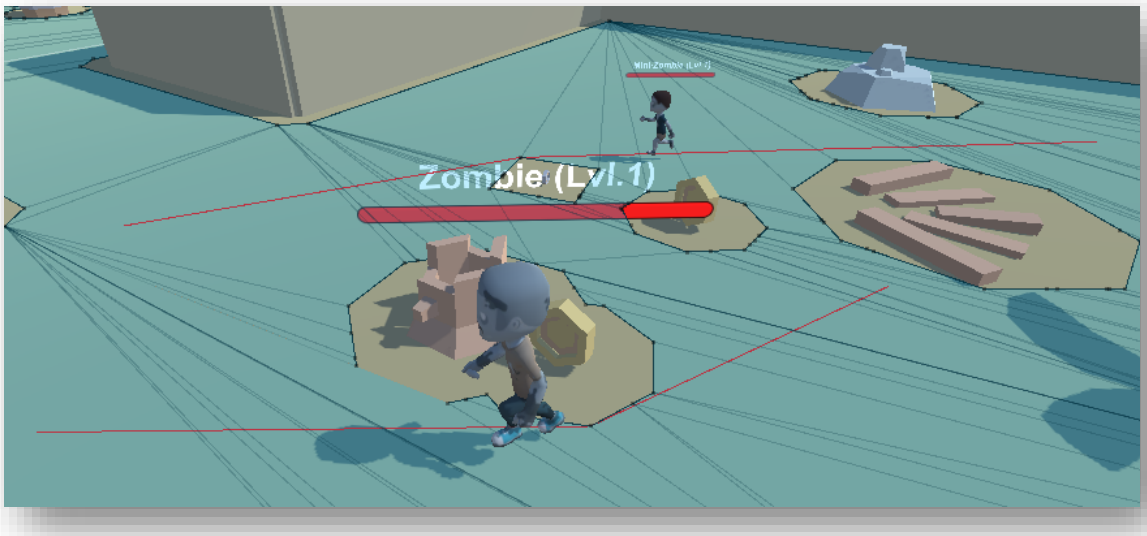
Binary Partitioning Algorithm (Runtime baked areas)





Enemy Navigation system (A* Algorithm)





Future Work

The project could be considered to be finished as it is because as mentioned above this is a demo to show some skills and algorithms on the field. The whole purpose of this demo is to develop a showcase of those algorithms recreating a real videogame and its implementations and problems while coding them.

Finish the development

However, if we would like to continue with the development and finally create a real videogame out of this demo, we could. There are some features which I could not implement on time for this project that could be good possible additions such as more weapons available to try, more worlds to explore and more challenging levels.

New game modes and fields to study

Multiplier would be a good addition to the game too. While designing the project in the early days I have considered to create a multiplier game mode to it but due to the little time I had to develop it I rejected the idea. It is interesting how connections between multiple clients works and I think this would be an incredible field to study to create a good infrastructure for a multiplier game mode.

Improvements and additions

With the decision to continue with the project comes the obligation to make some improvements to the game and some additions too. Right now, the demo has very little and simple sound effects and low poly graphics, these would be two fields to improve if we would like to improve the final quality of the product. Ambient music, dialogue voices, OST (Original Soundtrack), SFX (Special Effects) are other aspect to take into consideration as well.

Credits

Here are some of the assets I have used in the creation of the technical demo and its authors:

David Stenfors

- *3D Weapons*
  License: *Standard Unity Asset Store EULA.*

Kenny

- *Nature Kit*
  License: *CC0 1.0 Universal (CC0 1.0).*
- *Platformer Kit*
  License: *CC0 1.0 Universal (CC0 1.0).*
- *Animated Characters 1*
  License: *CC0 1.0 Universal (CC0 1.0).*
- *Animated Characters 2*
  License: *CC0 1.0 Universal (CC0 1.0).*
- *Onscreen controls*
  License: *CC0 1.0 Universal (CC0 1.0).*

Acronyms

- **AAA (Videogame)** Videogame high development budget.
- **API** Application Programming Interface.
- **AR** Augmented Reality.
- **CPU** Central Processing Unit.
- **FPS (Refresh frequency)** Frames per second.
- **FPS (Videogame Genre)** First Person Shooter.
- **OS** Operative System.
- **OST** Original Soundtrack.
- **SFX** Special Effects.
- **UI** User Interface.
- **UWP** Universal Windows Platform.
- **VR** Virtual Reality.

Bibliography

[1] Renck J. (Director), Mazin C. (Writer). (June 3, 2019). "*1:23:45*" (Episode 1) HBO, *Chernobyl* [TV Miniseries].

[2] "Pong" by user Bumm13, en.wikipedia.org, Public domain, Available: https://commons.wikimedia.org/w/index.php?curid=799667

[3] Gris (Videogame), Nomada Studio. (2019) https://twitter.com/nomadastudiobcn

[4] Valve Index, Valve (2021). https://store.steampowered.com/valveindex

[5] Bioshock (Videogame), Irrational Games, 2K Games.

[6] Path of Exile (Videogame), Grinding Gear Games.

[7] "Dolphin triangle mesh" by user Chrschn, Public Domain, Available: https://commons.wikimedia.org/w/index.php?curid=10626667

[8] "How Much of a Genius-Level Move Was Using Binary Space Partitioning in Doom?" by Two-Bit History, 06 Nov 2019. [Article about BSP algorithm] Available: https://twobithistory.org/2019/11/06/doom-bsp.html. [Accessed: 29-August-2021]

[9] "Binary Space Partitioning", en.wikimedia.org, Public domain, Available: https://commons.wikimedia.org/w/index.php?curid=641368

[10] "Inner Workings of the Navigation System", Unity, Unity Docs, Available: https://docs.unity3d.com/Manual/nav-InnerWorkings.html