Work Report

By Adriel Dias, Gabriel Schuenker, Kauan Farias and João Victor Resende

December 2024

About this Report

.

This report aims to thoroughly document the development process of the game "Cosmic Survivor". This game project was undertaken as part of the second evaluation for the Programming Languages discipline, a key component of the Data Science and Artificial Intelligence Course at EMAp.

The primary objective of the project was to enhance our understanding of Object-Oriented Programming (OOP) principles, focusing specifically on foundational concepts such as polymorphism, class inheritance, and encapsulation. By applying these concepts in a practical and creative way, we were able to deepen our grasp of how object-oriented methodologies can be leveraged to structure and organize complex systems like video games.

This report not only serves as a detailed record of our journey through the development of Cosmic Survivor but also highlights the skills acquired and the lessons learned along the way. It is our hope that this documentation can provide valuable insights into the process of applying OOP concepts to real-world applications.

First Game's concepts

.

For this project, the group's initial idea was to create a game in the "bullet hell" style, where the player must navigate through a barrage of projectiles while strategically eliminating enemies. In this type of game, the player takes control of a character who is placed in a dynamic and increasingly challenging environment, requiring quick reflexes, precision, and effective use of resources to survive.

The team decided to embrace the idea of a foreign planet environment. The main characters were envisioned as combatants equipped with sci-fi technological skills, relying on medium-to-long-range weapons.

In addition to creating a visually engaging scenario, our goal was to integrate a progression system where the player would encounter increasingly difficult enemy waves.

Ultimately, the "bullet hell" style was chosen because of its potential to deliver an exciting and high-energy experience, while also providing a solid foundation for implementing the programming principles we aimed to learn and apply.

First lines of code

.

Our project was developed using the powerful **pygame-ce** library, which served as the foundation for the game's development. The sprite object acts as the protagonist, serving as the main character that the player controls. To start, we designed the **Player** class as a subclass of the sprite class, which allowed us to define various attributes and behaviors specific to the player character.

The initial challenge was to implement the movement logic. At first, we developed a method that directly altered the position (x, y) of the player object based on the keys pressed (WASD). However, we quickly encountered an issue: when two non-opposing keys were pressed simultaneously (for instance, W and A), the movement speed was unnaturally increased, resulting in a faster and less controlled motion.

To address this, we implemented a more sophisticated solution using a direction vector. This vector receives values in both the x and y dimensions based on the keys pressed by the player and is subsequently normalized. The normalized direction vector is then multiplied by the player's speed attribute, yielding a smooth and consistent movement across the game world. This method ensures that the player's character moves in a more fluid, controlled manner, regardless of the number of keys pressed at once.

This vector-based approach to movement wasn't limited to the player character alone; we also applied this logic to other in-game objects such as enemies, projectiles, and special abilities. By using the same system to manage the positions of these objects, we ensured consistency and a more coherent game environment, where all elements moved in a unified and predictable manner.

Crafting the Scene

.

To create the game scenario, the team utilized the software Tiled, which is a widely used map builder that generates maps in the **TMX** format. This format is compatible with the pygame library, making it a natural choice for the project. The tileset used in the development was purchased from the website https://craftpix.net, a platform offering a wide range of free domain sprites suitable for game development. This tileset was chosen to provide a visual foundation for the alien environment, aligning with the game's theme and enhancing the overall aesthetic.

In the initial stages of development, the original plan was to convert each block and object on the map into a separate sprite in the game. However, this approach proved to be impractical as it only worked for very small and limited maps. The tileset blocks, although rich in detail, were relatively small in size, which led to an overwhelming number of sprites being generated, even for relatively short maps. This resulted in significant performance issues, as the game struggled to handle the high volume of objects, causing severe slowdowns. These performance drops occurred even before enemies or projectiles were introduced into the scene, which was a major concern.

To address this problem, we restructured the map generation process. Instead of creating a sprite for each block on the map, we decided to generate the entire map in **.tmx** format and then convert it into a **PNG** file. This process allowed us to merge the individual tiles into a large sprite representing the entire background of the game. This method reduced the number of sprites being loaded into memory and significantly improved the performance of the game. However, this change also presented a new challenge: collision detection, particularly for map boundaries and obstacles.

To solve the issue of collisions, we implemented a system of boundary restrictions based on the obstacles present on the map. Rather than associating each object with a collision area, we created rectangular regions that acted as invisible barriers, limiting the player's movement to certain areas. These collision areas were carefully designed to align with the obstacles in the game world, ensuring that the player could interact with the environment in a natural way, while still maintaining performance. This implementation was a much lighter solution, as it did not require complex object-based collision detection, which can be computationally expensive. As a result, this system proved to be highly efficient and did not incur any significant performance costs, which is why it was retained in the final version of the game.

Camera logic

.

To enable the use of a game map larger than the visible area of the computer's screen, we needed to implement a system for rendering game images. The primary goal was to center the player and adjust the position where other sprites are drawn based on the character's current position. To achieve this, we created an "offset" vector, which is calculated by subtracting the player's position from the vector corresponding to the center of the screen. This offset vector is then applied to the "drawing" position of all sprites. It's important to note that this method does not change the actual position of the objects in the game, but rather the position in which they are rendered on the screen. This approach ensures that the collision logic works correctly, as without this offset, the player's position would remain "fixed," which would disrupt interactions with the game environment. Additionally, by adopting this solution, we were able to maintain stable game performance even with large maps, as rendering objects and collision calculations became more efficient.

About the player and first combat logic

.

The initial concept for the game was to create a single, customizable character for the player, with the only change being the weapon they would wield. However, as development progressed, we decided to broaden this approach and introduce a variety of characters, each with unique attributes and appearances. But still in the early phase, we shifted our focus to developing the weapon systems and combat mechanics. To achieve this, we defined key character attributes such as health, damage, and speed, while also creating essential game classes, including the **Enemy** class, which represents the foes the player must face, and the **Gun** class, which serves as the protagonist's primary combat tool.

The mechanics behind combat were designed to be intuitive and simple, focusing on fun and fast-paced gameplay. Enemies are modeled as sprites, each defined by specific images and attributes that dictate their behavior. The core of enemy logic lies in the "behavior" method, which governs how enemies interact with the player. In the early stages, the first enemy types simply pursued the player relentlessly, creating a straightforward challenge. As we expanded the game, more diverse enemy behaviors were introduced to provide a wider range of challenges for the player.

Weapons, on the other hand, are dynamic objects that follow the player's movements on the screen. The primary functionality of the weapon is encapsulated in its "Shoot" method, which fires projectiles in the specified direction, factoring in a "cooldown" reload time defined within the class attribute. Bullets, which are derived from the Sprite subclass, are created with two key points: the starting position and the destination. The position of each bullet is updated every frame, using a collinear vector that calculates the difference between the initial and final positions. This allows for smooth movement of the bullets across the screen. Upon collision with an enemy, the bullet is destroyed and inflicts damage equal to its ”damage” attribute.

Collision detection in the game is handled using the **Sprite.rect** attribute, a built-in feature of Pygame that provides an easy way to detect overlaps between sprites. This system is highly efficient for determining when bullets hit enemies or when the player collides with objects in the game environment. By relying on Pygame’s built-in methods for sprite collision detection, we were able to keep the logic straightforward while maintaining the performance and responsiveness required for a fast-paced action game.

Enemies development

.

As our player’s enemies, we created various types of monsters, each with unique traits and behaviors that would challenge the player in different ways. However, only four types of enemies made it into the final version of the game. They are:

- Goblin: The Goblin is the most basic enemy in the game, designed to be a common yet persistent threat that players will encounter throughout their journey. It has fast movement speed, making it agile and difficult to avoid, and possesses average damage and health, which makes it a balanced opponent for the early stages of the game. This enemy type serves as the foundation for many of the game’s combat scenarios, offering a crucial challenge for the player to hone their skills and strategies. The Goblin also has a versatile sprite sheet with images for all directions, which enabled us to animate it effectively based on its movement. This is particularly useful as it allows for a more immersive and dynamic gameplay experience, as the enemy’s movements feel more lifelike. To achieve this, we used an **Enum** object, which streamlines the ”choice” algorithm by providing a faster and more efficient implementation for determining the Goblin’s current direction. This approach not only enhances the performance of the game but also adds a layer of animation complexity, as the Goblin’s behavior can change dynamically in response to the player’s actions.
- Alienbat: The Alienbat is a replicator enemy, meaning that every time it attacks the player, it creates a copy of itself. This feature makes the Alienbat particularly dangerous, as it can quickly overwhelm the player with a growing number of enemies if not dealt with swiftly. The Alienbat forces the player to strike quickly and with precision, as failing to eliminate it fast enough will result in an exponentially more difficult battle. Lately, we decided that it would be nicer if it only duplicates in the first attack, avoiding bat clusters and performance and gameplay issues.
- Slime: The Slime is the “hydra” type enemy, meaning that when killed, it splits into multiple smaller Slimes. This mechanic makes it a tricky adversary, as the player needs to eliminate not just one Slime, but potentially several smaller ones after the main one is defeated. The Slime serves as a test of the player’s ability to manage

multiple threats at once and requires careful positioning and timing to handle effectively. The slime mob ended to be a little problem in game's performance, which we'll see later in the optimization tab.

- Andromaluis: The Andromaluis is a stationary, summoning-type enemy. Unlike most other enemies, it does not move but has the ability to summon additional Goblins when the player gets too close. This creates a dynamic, multi-enemy encounter that forces the player to think strategically, as they must deal with waves of Goblins while also managing the threat posed by the Andromaluis itself. The Andromaluis adds an element of unpredictability, as it can rapidly escalate the difficulty of an encounter. It was indeed to be a boss, but we changed the concept to be just a harder to defeat mob, thus, we could add multiple enemies like it in the game.

Characters

.

As mentioned previously, after some time in development, we decided that it would be interesting to create characters with unique abilities for the game to have greater replayability. We tried to create different personalities for each of our heroes, reflecting the play style of each one, namely:

- **Cyborg**: The Cyborg is the initial character, designed to serve as an all-around hero with long-range capabilities and high area damage. His primary weapon is a **machine gun**, which, in the basic attack, fires a burst of bullets in a straight line, dealing small amounts of damage repeatedly. This weapon is ideal for taking out large groups of enemies quickly and efficiently. The Cyborg's first ability enhances his rate of fire and the damage of each bullet, enabling him to cut through waves of enemies with ease. This ability is especially useful during intense moments in the game when the player is overwhelmed by multiple enemies at once. The Cyborg's second ability, **Missile Rain**, launches a barrage of missiles that cause massive area-of-effect damage. This ability is arguably the most powerful in the game, as the missiles explode upon impact, wiping out any enemies in their blast radius. The sheer destructive power of Missile Rain makes it a game-changer, allowing the player to clear large sections of the battlefield in one fell swoop.

- **BladeMaster**: The first character to be unlocked, the BladeMaster, is an agile assassin who specializes in throwing knives at enemies. His attacks are focused on single targets, dealing increased damage with each strike. The BladeMaster's first ability boosts the damage of his knives and increases the frequency of his throws, turning him into a fast and deadly threat. Additionally, killing enemies with this ability restores a portion of the BladeMaster's lost health, creating a self-sustaining gameplay loop. Recognizing the strength of this ability, we introduced a balancing mechanic: while the BladeMaster is active, his self.armor attribute is reduced, making him more vulnerable to enemy attacks. This vulnerability forces the player to carefully consider positioning and timing, adding an extra layer of strategy to the gameplay. The BladeMaster's second ability is **Time Manipulation**, initially conceived as a movement speed buff for the character. However, after further consideration, we decided that this could disrupt the flow of the game, as it would make it harder for the player to avoid enemies at such high speeds. Instead, we reworked the ability to slow down all living enemies, simulating the BladeMaster's control over time. This ability not only affects enemy movement but also alters their animation speed, further enhancing the sensation of temporal manipulation and adding a unique tactical advantage.

- **Berserker**: The second unlockable character, the Berserker, is designed for players who prefer a tanky, slow-moving playstyle. His weapon of choice is a **shotgun**, which was one of the most enjoyable weapons to design. The shotgun features a unique mechanic where its bullets deal significant damage to enemies in close proximity, but the damage decreases as the bullets lose speed over distance. This design creates a "point-blank" combat style for the Berserker, making him highly effective at close range but less potent at longer distances. His first ability, **Iron Will**, enhances his resilience by temporarily increasing his armor to an extremely high value. This ability allows the Berserker to charge into the midst of enemies, withstanding substantial damage while dealing heavy blows to nearby foes. As his ultimate ability, the Berserker can activate an area of increased gravity, drawing enemies toward the center of the area and causing them to be shot down if they cross a certain threshold. This powerful ability serves as a devastating crowd control tool, turning the tide of battle by clustering enemies together before eliminating them in one fell swoop.

Optimization

.

After playing the game for some time, we noticed significant performance issues related to the loading of images for enemies and characters. Specifically, when the number of enemies on the screen increased, the game's performance would dramatically degrade, resulting in lag and a less enjoyable experience. To resolve this, we developed a more efficient image loading system. We created a function that would load all frequently used images into memory at the start of the game, which helped avoid the need to reload images repetitively. This approach greatly reduced the computational overhead associated with image handling during gameplay.

To further optimize this process, we incorporated an Enum object to manage the loading and storage of these images. By using an Enum, we could map each type of image to a specific constant, allowing us to centralize the image management process. This not only ensured a more streamlined approach to loading assets but also prevented redundant image loading operations, resulting in a notable improvement in the game's overall performance. With this optimization in place, the game was now capable of handling far larger swarms of enemies without causing performance bottlenecks.

However, we still faced challenges with area-of-effect (AoE) damage abilities, particularly when there were many slimes on the screen. The slime enemies had a unique mechanic where, upon death, they split into smaller slimes, a behavior reminiscent of the "hydra" concept in mythology. This mechanic caused a significant issue because, as multiple slimes were killed simultaneously, their images were resized numerous times, leading to a spike in memory usage and further degrading the game's performance.

To address this issue, we reworked the mob multiplication mechanic. Rather than allowing the slimes to multiply immediately upon death, we introduced a restriction where mob multiplication would only occur in areas where no active area damage was being applied. This adjustment ensured that the number of new slimes generated would be limited, thus preventing excessive image resizing and maintaining the game's performance even during intense combat situations. This change allowed us to keep the fun, chaotic nature of the slimes' behavior intact while ensuring the game remained optimized and playable.

More Animations

.

To enhance the overall user experience in the game, we focused on adding a series of animations that would make interactions feel more dynamic and engaging. Some of the key additions include:

- **Weapon Synchronization with Mouse Cursor**: One of the first improvements we implemented was a method that synchronized the player's weapon with the mouse cursor. This was done by calculating the angle of the vector between the weapon and the cursor, and then rotating the weapon's sprite accordingly. This animation not only added a sense of fluidity and realism to the gameplay, but it also gave the player a greater sense of control over their character, making the interaction feel more intuitive. We believed this would serve as an interesting and engaging way to attract players, ensuring the game felt responsive and visually appealing.
- **Character Movement Animations**: Initially, the sprite sheets for the player character lacked animations for vertical movement (up and down). To address this, we took the existing character sprites and manually drew additional frames that captured the movement in those directions. By doing this, we created a smoother and more cohesive animation experience for the player. The added animations not only ensured the player's movement was fluid, but also contributed to a more immersive experience. We were able to maintain the overall visual style of the character, while still enriching the gameplay with a complete set of movement animations, which we believe significantly enhanced the player's interaction with the game world.
- **Menu Button Animations**: We recognized that the main menu and interface interactions could benefit from some additional flair. To improve this, we added subtle animations and sound effects that were triggered when the player hovered

over or clicked on various buttons. These animations made the interface feel more responsive and lively, giving immediate visual feedback to the user, which is crucial for a satisfying interaction. By incorporating sound effects, we also added an auditory dimension to the experience, making the interface feel more polished and interactive. This not only improved the flow of navigating the menu but also created a sense of achievement and progress when the player interacted with the UI elements.

Each of these improvements was carefully designed to make the game feel more polished and user-friendly, enhancing the immersion and ensuring that players remained engaged throughout their experience. Through these animations, we were able to create a seamless, visually cohesive experience that rewarded user interaction and made the game environment feel more alive.

Credits to Resource Authors

.

The images and audio used in the game development were sourced from external providers and carefully adapted to align with the game's aesthetic and functional requirements. We would like to express our gratitude to the following websites and their corresponding authors for generously providing the resources that contributed significantly to the creation of the game:

- CraftPix.net: Provided a comprehensive tileset that was essential for creating the map and various environmental elements. This included character sprites, background features, and enemy designs, all of which added depth and immersion to the game world.
- OpenGameArt.org: Contributed a wide range of valuable resources, including environmental assets and enemy sprites, which were crucial for expanding the diversity of in-game elements and enriching the player's experience.
- Freesound.org: Served as the primary source for many of the sound effects used in the game. This platform provided a variety of audio files that enhanced the gameplay atmosphere, including ambient sounds and action effects.
- The Spriters Resource: Some of the sprites used in the game were sourced from this excellent database, which offers a collection of video game art assets. These sprites were adapted and integrated into our game to maintain visual consistency.

We deeply appreciate the effort and creativity of these contributors, whose resources have played an indispensable role in bringing our project to life.

Team Members and their Contributions

.

Below is how the group's general task division occurred.

Adriel Dias: Responsible for the majority of the core game mechanics and logic, programming and part of the background design.

Gabriel Schuenker: Responsible for the game difficulty implementation, most of the menu interface and debugging various issues

João Victor Resende: Responsible for the Docstring and unittests implementation in addition to the implementation of the character unlock system

Kauan Farias: Responsible for the games aesthetics by adapting sprites, improving interface, character selection system, and fixed bugs for a polished user experience.

.

Finally, we are deeply grateful to Professor **Rafael Pinho André** for his dedication in teaching this subject and for providing us with extensive knowledge that greatly contributed to our academic growth. His support and passion for education have been fundamental in guiding us throughout this journey.