Kangaroo Planet

Final defense report

Project made by :

Helios Bringuet, Eliott Caquelot,
Nicolas Delisle, Tanguy de Jerphanion

EPITA - A3
Long Story Short

Contents

Introduction

As we approach the culmination of our Kangaroo Planet project, this third and final defense report marks a significant milestone in our development journey. It provides a comprehensive overview of the project's current state, details the final features and refinements introduced since the second defense, and reflects on the evolution of our development process.

From the outset, we established clear goals and structured our collaboration around the strengths of each team member. This approach has continued to guide our progress, allowing us to efficiently enhance both the technical foundation and gameplay experience of Kangaroo Planet. Since the previous defense, we have focused on polishing the game's core mechanics, resolving outstanding issues, and optimizing the overall user experience.

This report is not only a summary of our progress—it also showcases the dedication, adaptability, and teamwork that have driven the project forward. As our final opportunity for formal evaluation, it aims to present a clear picture of the game's readiness and the thoughtful process behind its development.

Ultimately, this report stands as a testament to our team's commitment to delivering a high-quality, engaging final product. It concludes our defense milestones and sets the stage for the official launch of Kangaroo Planet.

Chapter 1

Objectives, tasks, and technologies used

1.1 Game Overview

Kangaroo Planet is an action-survival video game in which the player controls a kangaroo defending its home planet from a human invasion. The core gameplay revolves around surviving waves of enemies while navigating increasingly difficult challenges. The game features three distinct modes:

- A **main mode** designed in a die-and-retry style, consisting of 5 handcrafted levels with progressively complex enemy waves and mechanics.
- A **training mode**, offering a single infinite procedurally generated level to help players practice and improve their skills in a dynamic and ever-changing environment.
- A **multiplayer 1v1 mode**, where two players compete in real-time, each controlling a kangaroo and battling for dominance through strategic movement and combat.

Its gameplay focuses on fast-paced action, reactive decision-making, and replayability through both solo and competitive experiences.

1.2 Project objectives

The main objectives of the project are as follows:

- Provide an accessible and entertaining gaming experience with solid mechanics.
- Develop a unique stylization, where the player embodies a kangaroo.
- Implement diverse and well-balanced levels.
- Ensure optimal performance and a smooth user experience.

1.3 Task distribution

Tasks	Lead	Backup
Training Mode	Nicolas Delisle	Eliott Caquelot
Main Mode	Helios Bringuet	Tanguy de Jerphanion
Kangaroo Modeling	Eliott Caquelot	Tanguy de Jerphanion
Enemy Modeling	Eliott Caquelot	Nicolas Delisle
Weapon Development	Tanguy de Jerphanion	Helios Bringuet
AI Development	Tanguy de Jerphanion	Eliott Caquelot
Shooting Mechanics	Helios Bringuet	Nicolas Delisle
Audio Integration	Nicolas Delisle	Eliott Caquelot
Menu Design	Tanguy de Jerphanion	Helios Bringuet
Multiplayer	Eliott Caquelot	Tanguy de Jerphanion
Damage System	Helios Bringuet	Nicolas Delisle
Statistics System	Tanguy de Jerphanion	Eliott Caquelot

1.4 Technologies used

The development of *Kangaroo Planet* relies on the following tools and technologies:

- **Programming language**: C#, used for implementing the game mechanics.
- **Game engine**: Unity, for managing scenes, objects, and interactions.
- **Project management**: Trello and Discord, used to organize tasks and track project progress.

Chapter 2

Weeks 1 and 2 - Development (Nicolas)

2.1 Introduction

During the first weeks of the project, the team focused on establishing the functional foundation of **Kangaroo Planet**'s gameplay. The scripts developed during this period include essential mechanics such as player control, enemy AI, weapon and projectile handling, and the wave-based enemy spawning system. These elements form the core upon which the rest of the game is built.

2.2 Player Control System: ***Player.cs***

The `Player.cs` script handles the player's main actions, including movement, aiming, and shooting.

Player movement is controlled using the keyboard's directional keys. The script reads user input via `Input.GetAxisRaw("Horizontal")` and `Input.GetAxisRaw("Vertical")` to create a movement vector. This vector is normalized and multiplied by a defined speed (`moveSpeed`) to produce a consistent velocity. This velocity is then passed to the `PlayerController` component, which physically applies the movement to the character.

Aiming is controlled via the mouse position. A ray is projected from the camera towards the mouse cursor (`Input.mousePosition`), and the point where it intersects a virtual plane (the ground) is calculated using a `Plane`. The player character then rotates to face this point, which is essential for accurate directional shooting.

When the left mouse button is pressed (`Input.GetMouseButton(0)`), the script calls the `Shoot()` method from the `GunController`, activating the equipped weapon to fire a projectile in the desired direction.

2.3 Enemy AI: *Enemy.cs*

The `Enemy.cs` script controls enemy movement and pursuit behavior.

Each enemy uses a `NavMeshAgent` to navigate realistically through the environment. The enemy continuously targets the player, with the player's position being updated via a coroutine called `UpdatePath()`. This method recalculates the path to the player every 0.25 seconds to maintain responsiveness while optimizing performance.

The script inherits from the `LivingEntity` class, enabling vital state management, such as handling death. When an enemy dies, it triggers the `OnDeath()` event, which initiates actions like decreasing the count of active enemies.

2.4 Weapon Handling: *Gun.cs*

The `Gun.cs` script manages weapons and their core functionalities.

When the `Shoot()` method is called, a projectile is instantiated at the muzzle's position (`muzzle`). The script checks the time since the last shot (`nextShotTime`) to enforce a firing rate defined by `msBetweenShots`.

The instantiated projectile is launched with an initial speed (`muzzleVelocity`), ensuring a consistent trajectory that allows players to aim and hit targets accurately.

2.5 Wave System: *Spawner.cs*

An enemy wave spawning system has been implemented using the `Spawner.cs` script.

Waves are defined using an internal `Wave` class, which specifies the number of enemies and the delay between their spawns (`timeBetweenSpawns`). Each wave starts with a call to the `NextWave()` method, which initializes the counters for enemies to spawn (`enemiesRemainingToSpawn`) and to defeat (`enemiesRemainingAlive`).

At regular intervals, enemies are instantiated and placed in the scene. Each enemy is linked to the `OnDeath()` event, allowing the spawner to monitor the number of remaining enemies and trigger the next wave when the current one is cleared.

2.6 Damage Interface: *IDamageable.cs*

The `IDamageable.cs` script defines a common interface for all entities that can receive damage. This interface includes the `TakeHit()` method, which is called whenever an entity is attacked. It allows for consistent damage handling across different entity types, such as enemies and the player.

2.7 Conclusion

During these first two weeks, the team laid down the core gameplay systems. The implemented features allow the player to interact with the environment, engage in combat, and progress through successive enemy waves. These systems provide a solid framework for integrating more advanced features in the upcoming phases of development.

Chapter 3

Weeks 3 and 4 - Procedural Generation and the Fisher-Yates Shuffle Algorithm (Helios)

3.1 Procedural Map Generation

Procedural generation refers to the algorithmic creation of game content with limited or no human input. In our project, procedural map generation is utilized to construct dynamic and varied levels for each play session, which contributes significantly to the replayability and unpredictability of the gameplay. The engine responsible for this is defined in the `MapGenerator.cs` script. This system dynamically creates a tile-based map, populates it with obstacles, and establishes boundaries to guide AI and player movement.

3.1.1 Overview of the Map Generation Process

The procedural generation pipeline is triggered each time a new enemy wave is initiated. The `MapGenerator` component listens for the `OnNewWave` event dispatched by the `Spawner` object. When this event is caught, the map generator resets the current environment and proceeds to generate a new one based on the parameters defined in the current `Map` object.

This process includes the following key stages:

1. Initializing tile coordinates and shuffling them.
2. Creating the tile grid and adjusting tile spacing.
3. Placing obstacles using a constrained random distribution.
4. Validating the accessibility of the map.
5. Applying NavMesh masks and adjusting floor geometry.

3.1.2 Coordinate Initialization and Shuffling

A foundational step in the generation pipeline is building a list of all valid tile coordinates. These are stored in the `allTileCoords` list. Each coordinate is a `Coord` struct representing a specific location on the grid (X, Y). Once the list is populated, it is shuffled using the Fisher-Yates algorithm (discussed in detail later), ensuring randomness in the selection of tiles for obstacle and spawn point placement.

The shuffled coordinates are stored in a queue, `shuffledTileCoords`, allowing the script to pull coordinates in a randomized but deterministic order. A second queue, `shuffledOpenTileCoords`, is similarly created to track tiles that remain walkable after obstacle placement.

3.1.3 Tile Grid Construction

Tile creation is handled through the instantiation of a predefined prefab called `tilePrefab`. Each tile is placed according to a grid centered around the world origin. The coordinate conversion from logical grid coordinates to Unity world coordinates is handled by the `CoordToPosition(x, y)` method.

To avoid the visual monotony of a perfectly uniform grid, an `outlinePercent` is applied to each tile's local scale. This creates a thin gap between adjacent tiles, increasing visual readability and contributing to the stylized appearance of the environment. All tile objects are parented to a `mapHolder` GameObject, enabling easy cleanup and regeneration.

3.1.4 Obstacle Placement and Visual Variation

Obstacle generation is governed by the `obstaclePercent` parameter of the `Map` object. The total number of obstacles is computed as a percentage of the total number of tiles. For each obstacle, a coordinate is drawn from the `shuffledTileCoords` queue and tested for suitability.

Visual variety is introduced through two primary methods:

- **Randomized Height:** Each obstacle is assigned a height value chosen via linear interpolation between `minObstacleHeight` and `maxObstacleHeight`. This prevents flat and featureless environments.
- **Gradient Coloring:** Obstacle materials are tinted using a color gradient interpolating between `foregroundColour` and `backgroundColour`, providing visual cues that correlate with vertical height.

These visual techniques ensure that even randomly generated maps maintain aesthetic appeal and visual clarity, crucial for both user experience and gameplay.

3.1.5 Accessibility Validation with Breadth-First Search

To ensure that the generated map remains playable, a Breadth-First Search (BFS) algorithm is employed to validate that all accessible tiles (i.e., those not covered by obstacles) form a single contiguous region. This is particularly important in preventing isolated "islands" that would be unreachable to the player or AI.

The BFS is initialized from the central tile (computed via `mapCentre`) and expands outward to explore connected walkable tiles. The algorithm maintains a visited matrix to track tiles it has processed. If the number of visited tiles matches the expected number of walkable tiles, the map is deemed valid. Otherwise, the last placed obstacle is removed and a new coordinate is tested.

This validation loop ensures both fairness and gameplay integrity, as it guarantees that every non-obstacle tile is reachable from the center of the map.

3.1.6 NavMesh Masking and Boundaries

After tiles and obstacles are placed and validated, the generator configures navigation boundaries. This includes four NavMesh mask objects placed around the periphery of the tile grid: left, right, top, and bottom. These are instantiated using the `navmeshMaskPrefab` and scaled dynamically based on the difference between the current map size and the maximum allowed size.

Additionally, two ground objects—`navmeshFloor` and `mapFloor`—are resized to accommodate the current tile grid. The former defines the area for Unity's pathfinding system (NavMesh), while the latter provides the visible ground plane rendered in the scene.

3.1.7 Map Data Encapsulation

Map configuration is encapsulated in the `Map` class. This allows for each wave or level to define its own unique map settings. The parameters defined in the `Map` class include:

- **mapSize (Coord):** Grid width and height.
- **obstaclePercent:** Density of obstacles.

- **seed:** Integer for consistent random generation.
- **minObstacleHeight / maxObstacleHeight:** Range for obstacle elevation.
- **foregroundColour / backgroundColour:** Color gradient applied to obstacles.

Using this structure, designers can easily define multiple unique map profiles and experiment with different gameplay dynamics, such as more crowded or open environments, varying color schemes, or extreme elevation changes.

3.2 Fisher-Yates Shuffle Algorithm

Randomization plays a critical role in procedural generation, and a poor randomization method can lead to biased, repetitive, or unbalanced results. The Fisher-Yates shuffle is a classic, proven algorithm for producing an unbiased permutation of a list.

3.2.1 Algorithm Description

The Fisher-Yates algorithm iteratively swaps elements in a list to achieve uniform randomness. The process is:

```
for i from n - 1 downto 1 do
    j ← random integer such that 0  j  i
    exchange a[i] and a[j]
```

This algorithm ensures that each of the n! permutations is equally likely, provided the random number generator is fair.

3.2.2 Why Fisher-Yates?

Compared to naive methods like sorting with random weights (e.g., `Random.Sort()`), Fisher-Yates provides several advantages:

- **Time Complexity:** O(n), compared to O(n log n) for sorting-based methods.
- **Unbiased Output:** Every permutation is equally probable.
- **Determinism with Seed:** Produces reproducible results when initialized with the same seed.

- **Simplicity:** Easy to implement and understand.

These properties make Fisher-Yates particularly well-suited for applications like level generation, where fairness and performance are essential.

3.2.3 Applications in the Project

In our procedural system, Fisher-Yates is used in two critical places:

1. **Tile Coordinate Shuffling:** The list `allTileCoords` is shuffled to determine the order in which tiles are selected for obstacle placement.
2. **Open Tile Coordinate Shuffling:** After obstacle placement, the list of remaining walkable tiles is shuffled and stored in `shuffledOpenTileCoords`, which is later used to determine enemy spawn locations.

These uses of Fisher-Yates guarantee that obstacle and spawn point selection is both efficient and statistically fair, contributing to balanced and engaging level designs.

3.2.4 Random Seed and Determinism

The inclusion of a user-defined seed in the `Map` class allows for deterministic procedural generation. When the same seed is supplied, the map will always generate identically. This is especially valuable for:

- **Debugging:** Developers can reproduce specific map instances.
- **Replayability:** Players can share seeds to recreate favorite maps.
- **Testing:** Controlled testing environments can be created consistently.

Unity's `Random.InitState(seed)` is used to ensure that all randomized operations in map generation are affected by this seed, including Fisher-Yates shuffling.

3.3 Conclusion

The procedural generation system implemented in `MapGenerator.cs` demonstrates how data-driven, algorithmic approaches can be combined with visual and gameplay design principles to create dynamic, replayable environments. By leveraging a tile-based system, controlled randomization, and accessibility validation, we are able to produce maps that are both unique and fair.

The Fisher-Yates shuffle plays a vital supporting role by ensuring unbiased selection in a variety of contexts, from tile placement to enemy spawning. Together, these systems exemplify how algorithmic logic and creative game design can coexist to enhance player experience.

Future improvements may include the introduction of multi-zone maps, biome-based tile sets, procedural scenery objects, or AI-aware terrain features such as chokepoints and cover zones, all of which can build upon the foundation laid in this early procedural generation system.

Chapter 4

Weeks 5 and 6 - Refining Core Mechanics in Gun.cs, Enemy.cs, and Spawner.cs (Tanguy)

During these two weeks, the team focused on revisiting and improving the core scripts of the project: `Enemy.cs`, `Gun.cs`, and `Spawner.cs`. These scripts, initially implemented in weeks 1 and 2, were revisited to bring them to a level of technical excellence appropriate for the project's needs. Below, we detail the specific improvements made to each script.

4.1 Refining `Enemy.cs`

The `Enemy.cs` script is essential for managing enemy behavior, including movement, interaction with the player, and responses to various events such as attacks and damage. The main improvements this week were the following.

The state management system (`Idle`, `Chasing`, `Attacking`) was restructured to improve efficiency. For example, the logic in the `Update()` method was optimized to avoid unnecessary calculations.

```
void Update() {
    if (hasTarget) {
        if (Time.time > nextAttackTime) {
            float sqrDstToTarget = (target.position - transform.position).sqrMagnitude;
            if (sqrDstToTarget < Mathf.Pow(attackDistanceThreshold
                + myCollisionRadius + targetCollisionRadius, 2)) {
                nextAttackTime = Time.time + timeBetweenAttacks;
                StartCoroutine(Attack());
            }
        }
    }
}
```

This change ensures that distance calculations are only performed when the attack cooldown has elapsed.

The attack system was fully reworked to synchronize both visually and mechanically the enemy animation with the damage applied.

```
IEnumerator Attack() {
    bool hasAppliedDamage = false;

    while (percent <= 1) {
        if (percent >= 0.5f && !hasAppliedDamage) {
            hasAppliedDamage = true;
            targetEntity.TakeDamage(damage);
        }
        percent += Time.deltaTime * attackSpeed;
        yield return null;
    }
}
```

The `SetCharacteristics()` method was expanded to allow full customization of enemy behavior. In addition to speed and color, the script now automatically calculates damage based on the player's starting health.

```
public void SetCharacteristics(float moveSpeed, int hitsToKillPlayer,
                               float enemyHealth, Color skinColour) {
    pathfinder.speed = moveSpeed;
    damage = Mathf.Ceil(targetEntity.startingHealth / hitsToKillPlayer);
    startingHealth = enemyHealth;
    skinMaterial.color = skinColour;
}
```

These changes ensure smoother and more realistic enemy management while improving code readability.

4.2 Refining `Gun.cs`

The `Gun.cs` script, which governs the shooting mechanics, also received several key updates.

Projectile trajectory management was adjusted to better reflect the player's orientation.

```
Vector3 shootDirection = (targetPoint - gunBarrelEnd.position).normalized;
Projectile newProjectile = Instantiate(projectile,
                                       gunBarrelEnd.position, Quaternion.identity);
newProjectile.SetSpeed(projectileSpeed, shootDirection);
```

The handling of visual and sound effects upon impact was revised to prevent duplicates and excessive function calls.

```
if (Physics.Raycast(ray, out hit, range)) {
    AudioManager.instance.PlaySound("Gunshot", transform.position);
    Instantiate(hitEffect, hit.point, Quaternion.LookRotation(hit.normal));
}
```

These refinements make the shooting mechanics more intuitive for the player while optimizing overall performance.

4.3 Refining `Spawner.cs`

The `Spawner.cs` script was reworked to enhance the management of enemy waves.

An anti-camping system was implemented. It detects whether the player stays in the same location for too long (`isCamping`) and adjusts enemy spawn positions accordingly.

```
if (Vector3.Distance(playerT.position, campPositionOld) < campThresholdDistance) {
    isCamping = true;
    spawnTile = map.GetTileFromPosition(playerT.position);
} else {
    isCamping = false;
    spawnTile = map.GetRandomOpenTile();
}
campPositionOld = playerT.position;
```

The `NextWave()` method was improved to include a sound cue and a visual effect indicating the end of a wave.

```
if (currentWaveNumber > 0) {
    AudioManager.instance.PlaySound2D("Level Complete");
}
```

4.4 Conclusion on Weeks 5 and 6

These improvements significantly strengthened the structure of the project's core scripts, making the code more efficient, modular, and scalable. The game now benefits from more responsive enemy behavior, accurate shooting mechanics, and a more dynamic enemy wave system. These upgrades mark a crucial step in the project's development, establishing a solid foundation for upcoming features.

Chapter 5

Weeks 7 and 8 - Audio and Visual Effects Integration (Eliott)

5.1 Audio Management and Implementation

During weeks 7 and 8, our primary focus was the integration of high-quality audio and visual effects. These features are critical for transforming the gameplay from a functional prototype into a polished, immersive experience. Audio, in particular, affects player perception and emotion on a deep level—it is not merely functional, but atmospheric. Through layered sound design and intelligent management scripts, our goal was to create an audio space that enhances tension, reinforces game feedback, and complements the visual style.

The core system for sound control is handled by the `AudioManager` script, which acts as a centralized hub for managing all audio playback in the game. It provides separate controls for three major audio categories:

- **Master Volume:** Controls all audio output globally.
- **SFX Volume:** Manages the volume of sound effects such as gunfire, explosions, footsteps, and ambient actions.
- **Music Volume:** Independent control of background music levels.

This tri-channel approach provides flexibility for both developers and players. For example, players can mute the music while keeping sound effects active, or vice versa. Each channel is controlled through Unity UI sliders and persisted using Unity's `PlayerPrefs` for seamless user preferences across sessions.

Sound triggers are implemented in response to key gameplay events—player shooting, enemy spawning, collisions, etc. Effects can be played in either 2D (for UI and non-spatial sounds) or 3D (for environment-specific sounds). The spatialized sound allows players to detect the source direction and distance of sounds, adding depth and realism to gameplay.

```
void PlaySound(AudioClip clip, Vector3 position) {
    AudioSource.PlayClipAtPoint(clip, position, sfxVolume);
}
```

5.2 Sound Organization and Randomization

To avoid repetitiveness, sound playback is randomized using the `SoundLibrary` system. Instead of assigning a single clip to each action, clips are grouped by category (e.g., “Gunshot”, “Explosion”, “Footstep”). When a sound is needed, a random clip is selected from the corresponding group.

This design increases auditory variation and prevents player fatigue caused by hearing the same exact sound repeatedly. The `SoundLibrary` uses a dictionary of string keys and lists of `AudioClip` objects for fast lookup and random selection.

```
AudioClip GetRandomSound(string groupName) {
    List<AudioClip> soundList = soundGroups[groupName];
    return soundList[Random.Range(0, soundList.Count)];
}
```

Additionally, cooldown systems are sometimes applied to prevent audio spam (e.g., when the fire button is held), especially for sounds that might otherwise overlap too frequently and create auditory clutter.

5.3 Visual Feedback and VFX Integration

In parallel with audio, visual effects were developed and fine-tuned to reinforce player actions and increase visual clarity. The core visual feedback systems implemented during this phase include muzzle flashes, shell ejections, and impact sparks. These are designed to be short-lived but noticeable, providing visual reinforcement for rapid combat interactions.

5.3.1 Muzzle Flash Effects

The `MuzzleFlash` script handles the brief flash of light that appears when a gun is fired. This effect, while minor in code, is crucial for giving visual weight to gunfire. Each flash is randomly chosen from a set of sprite variations and displayed for just 0.1 seconds using a coroutine.

```
void CreateFlash() {
    flashSprite.SetActive(true);
    StartCoroutine(FlashDuration());
}

IEnumerator FlashDuration() {
    yield return new WaitForSeconds(0.1f);
    flashSprite.SetActive(false);
}
```

This visual cue reinforces the timing and position of shots, and works in tandem with sound effects and recoil animations to create a satisfying shooting experience.

5.3.2 Shell Ejection

To further enhance realism, a shell casing is spawned and ejected each time the player fires a weapon. The `Shell` script instantiates a small rigidbody object, applies randomized force vectors, and optionally triggers metallic bounce sounds as the casing hits the ground. Casings are destroyed after a delay to prevent performance issues.

```
void SpawnShell() {
    GameObject shell = Instantiate(shellPrefab, shellSpawnPoint.positi
    Rigidbody rb = shell.GetComponent<Rigidbody>();
    rb.AddForce(Random.insideUnitSphere * 5f, ForceMode.Impulse);
    Destroy(shell, 3f);
}
```

The result is a subtle but effective immersion layer. Shells create short-lived motion on the screen and reinforce a sense of physicality and action.

5.4 Dynamic Music Transitions and Ambient Soundscapes

The game uses a modular approach to background music management through the `MusicManager` script. This manager handles automatic track switching between game scenes (e.g., menu, loading, gameplay) and supports smooth transitions using fading techniques.

```
IEnumerator FadeVolume(float targetVolume, float duration) {
    float start = musicSource.volume;
    for (float t = 0; t < duration; t += Time.deltaTime) {
        musicSource.volume = Mathf.Lerp(start, targetVolume, t / durat
        yield return null;
    }
    musicSource.volume = targetVolume;
}
```

This attention to transition quality prevents jarring audio cuts and enhances emotional consistency. For example, transitioning from an intense combat track to a quiet menu ambiance should feel natural and paced.

5.5 A Soothing Musical Contrast: Daft Punk's "Veridis Quo"

An interesting and intentional artistic decision was the inclusion of Daft Punk's track *"Veridis Quo"* during specific in-game moments. This track is characterized by its soothing, melancholic synth progression and slow pacing—quite atypical for a fast-paced shooter environment.

Why choose a soothing piece in a combat-focused game?

- **Emotional Contrast:** After a wave of high-intensity combat, transitioning to a calm, reflective track creates emotional contrast. This enhances the perceived impact of each gameplay phase by emphasizing variety.
- **Tension by Calmness:** Paradoxically, calm music can build tension more effectively than aggressive music, particularly when players are expecting danger. The absence of heavy drums or aggressive melodies can make players more alert and uneasy.
- **Pacing and Recovery:** A game with constant intensity becomes exhausting. Music like "Veridis Quo" gives players moments to breathe, think, and appreciate the atmosphere before tension ramps back up.
- **Atmospheric Depth:** The track evokes a sense of mystery and introspection, aligning with the idea of the player exploring deeper truths or uncovering hidden parts of the world.

This unconventional use of ambient music illustrates how audio design can subvert expectations and support narrative or thematic intent even in minimalist contexts.

5.6 Conclusion and Future Extensions

Weeks 7 and 8 marked a significant transformation in the game's feel and polish. By integrating both auditory and visual effects, the experience became more responsive, immersive, and emotionally engaging. Sound design through the `AudioManager` and `SoundLibrary` introduced layered interactivity, while visual systems like muzzle flashes and shells supported clarity and realism.

Perhaps most critically, the music system introduced the ability to create emotional nuance through auditory contrast. Tracks like *"Veridis Quo"* offered a new dimension of meaning and pacing, revealing how deliberate sound choices can shape player perception even in mechanically driven games.

Looking ahead, the audio/visual system could be expanded in the following ways:

- **Dynamic Layering:** Adaptive music that changes intensity based on in-game variables (e.g., player health, enemy proximity).
- **Surface-Dependent SFX:** Different footstep and bullet impact sounds depending on the surface type (metal, wood, dirt).
- **Environmental Audio:** Adding wind, mechanical drones, or distant explosions to create a more vibrant soundscape.
- **Reverb Zones:** Using Unity's audio zones to simulate sound changes in confined vs. open environments.

Overall, this phase highlighted how multimedia elements—particularly sound—can deeply influence gameplay experience, sometimes more than visuals or mechanics alone.

Chapter 6

Weeks 9 and 10 - Visuals (Eliott and Nicolas)

6.1 Introduction

During the first two weeks of the project, the focus was primarily on building the visual and gameplay foundations of *Kangaroo Planet*. A significant part of this phase involved creating 3D models for the main character and enemies while ensuring that the game's aesthetic remained consistent with its relaxing yet action-packed theme. Alongside this, visual effects were carefully designed to enhance the player experience, adding an extra layer of polish to the game.

This chapter outlines the work done on 3D modeling and visual enhancements, detailing the thought process behind the design choices and their impact on the overall gaming experience.

6.2 3D modeling: Bringing *Kangaroo Planet* to life

One of the key tasks during this early development stage was to create 3D models that would define the visual identity of the game. Given that *Kangaroo Planet* revolves around a kangaroo protagonist defending its home, it was crucial to craft a character that was both visually appealing and stylistically in line with the game's unique world.

Creating 3D models requires an understanding of shape composition, proportions, and animation readiness. Since *Kangaroo Planet* is set in an imaginative world, the art style did not need to be hyper-realistic but rather consistent with the playful and engaging atmosphere of the game. To achieve this, the following approach was taken:

- **Basic shape manipulation**: The process began with experimenting in Blender, one of the most widely used 3D modeling software tools. By learning how to manipulate simple geometric shapes such as spheres, cylinders, and cubes, it became possible to construct more complex structures by combining and modifying these basic forms. The kangaroo model, for instance, started as a rough composition of multiple spheres and ovals, which were gradually reshaped to resemble the distinct anatomy of the animal.

- **Creating the Kangaroo model**: A kangaroo has a very recognizable silhouette, with its strong hind legs, small front paws, long tail, and upright stance. To maintain a balance between realism and stylization, the design exaggerated some of these features to make the character more expressive. The legs were made slightly larger to emphasize movement, while the tail was extended to serve as a counterbalance, enhancing the animation's fluidity.
- **Designing the enemies**: The enemy models followed a different design philosophy. Since the game features humans as invaders, their models had to contrast with the kangaroo's natural, organic look. The human enemies were designed using more rigid and blocky shapes to subtly highlight their artificial, uninviting presence. These design choices reinforced the game's underlying theme of nature versus intrusion.
- **Texturing and refinements**: Once the base models were completed, textures and colors were applied to bring them to life. Instead of using highly detailed textures, a more simplified and stylized approach was chosen to maintain the game's unique visual identity. Soft shading and vibrant colors helped distinguish different elements of the characters, making them instantly recognizable in the fast-paced environment of the game.

Beyond simply constructing models, these early design decisions played a crucial role in shaping the game's overall feel. A well-designed main character can strengthen the player's emotional connection to the game, making them more engaged in the story and gameplay. Similarly, well-designed enemies contribute to the clarity of combat interactions, ensuring that the player can quickly identify threats and react accordingly.

By focusing on intuitive shape language, the team ensured that *Kangaroo Planet* maintained a coherent aesthetic while allowing room for creative and dynamic character animations in later stages of development.

6.3 Enhancing gameplay through visual effects

While modeling defined the game's physical appearance, visual effects played an equally vital role in making the experience more engaging and immersive. These effects were designed to align with the game's relaxing yet action-packed nature, providing visual feedback that enhanced player satisfaction.

The overarching goal of *Kangaroo Planet* was not just to create an action game but to offer an experience that felt smooth and enjoyable. The visual effects played a key role in reinforcing this balance:

- **Soft and smooth effects**: Instead of relying on harsh visual elements commonly found in high-intensity shooters, *Kangaroo Planet* adopted a more fluid and subtle approach. Particle effects were designed to be visually appealing without being overwhelming, ensuring that combat remained engaging without disrupting the relaxed atmosphere.
- **Enhancing combat responsiveness**: One of the core aspects of a good shooter is how satisfying the feedback is when hitting a target. To improve this, various visual cues were implemented:
  - Enemies briefly flashed red upon being hit, providing an immediate visual confirmation of a successful attack.
  - Small impact effects were added to simulate the feeling of a well-placed shot without cluttering the screen.
  - Smooth animations accompanied enemy reactions to hits, reinforcing a sense of weight and impact in combat interactions.
- **Stylized explosions and environmental effects**: When an enemy was defeated, instead of using hyper-realistic explosions or excessive gore, a more stylized approach was taken. Enemies disappeared in a puff of colorful particles, keeping the visuals dynamic while maintaining the game's approachable tone.
- **Adapting effects to gameplay mechanics**: Every visual effect had to be carefully calibrated to match the pace of the game. Effects that were too slow would make combat feel sluggish, while effects that were too fast could become distracting. Through iterations and fine-tuning, the effects were refined to strike the perfect balance between responsiveness and aesthetic appeal.

Beyond their functional role, visual effects contribute significantly to the overall enjoyment of a game. They provide an intuitive way for players to understand what is happening on screen, guide their attention toward important gameplay elements, and reinforce the emotional impact of each action.

In *Kangaroo Planet*, every effect was designed with these principles in mind. Whether it was a simple muzzle flash when shooting, a burst of color when an enemy was hit, or a dynamic lighting change during specific moments, each visual enhancement worked together to create a cohesive and enjoyable experience.

6.4 Conclusion

The first two weeks of development were crucial in establishing the artistic and visual identity of *Kangaroo Planet*. Through the creation of 3D models and the implementation of carefully crafted visual effects, the foundation was laid for an engaging and immersive gameplay experience.

By learning Blender and experimenting with shape composition, the team successfully designed a unique and expressive protagonist along with distinct enemies that fit the game's theme. Meanwhile, the focus on visual feedback and environmental effects ensured that every interaction felt rewarding and fluid.

Chapter 7

Weeks 11 and 12 - Start of the P2P multiplayer (Tanguy)

7.1 Introduction

As *Kangaroo Planet* evolved, it became clear that adding a multiplayer mode would significantly enhance the game's experience. Multiplayer gaming introduces new dimensions of fun, competitiveness, and cooperation, making it a highly desirable feature. However, implementing multiplayer functionality comes with a range of technical challenges, from networking architecture decisions to latency management and synchronization.

The first step in this process was choosing a suitable networking architecture. After careful consideration, we decided to implement a **peer-to-peer (P2P)** multiplayer system using the **Mirror** networking library for Unity. This chapter explores the fundamentals of P2P networking, its advantages and disadvantages, and why Mirror was ultimately chosen as our solution.

7.2 Understanding Peer-to-Peer (P2P) networking

Peer-to-peer (P2P) networking is a decentralized communication model where multiple devices (peers) communicate directly with each other without relying on a central server. In gaming, this means that players' devices connect to each other directly to exchange data, such as player movements, actions, and game state updates.

In a P2P multiplayer game, one player's device often acts as the host (sometimes called the "authoritative peer"), while other players connect as clients. The host is responsible for processing game logic, synchronizing data, and resolving conflicts between players. However, in true P2P models, all peers share responsibility for maintaining game state, which can sometimes lead to inconsistencies if not properly managed.

P2P networking is just one of several architectures used in online multiplayer games. The main alternatives include:

- **Client-Server model**: A dedicated server manages all game logic and communications, ensuring a synchronized and controlled multiplayer experience.
- **Hybrid model**: A combination of P2P and client-server, where a central authority (such as a server) exists but offloads some tasks to peers.

Each model has trade-offs in terms of performance, cost, and complexity, which we explored before deciding on P2P for *Kangaroo Planet*.

7.3 Advantages of P2P multiplayer

One of the biggest advantages of P2P networking is that it eliminates the need for dedicated game servers. Since players directly communicate with each other, there is no need to maintain expensive infrastructure, making P2P an appealing choice for indie games and projects with limited budgets.

Since players connect directly without routing data through a central server, latency can sometimes be lower than in traditional client-server models. This is particularly useful in smaller-scale games with players located in close proximity.

For games that support a limited number of players per session (such as 2-4 player co-op games), P2P can be a lightweight and efficient solution. It does not require the complex matchmaking and server distribution systems needed for large-scale multiplayer experiences.

7.4 Disadvantages of P2P multiplayer

In most P2P implementations, one player takes on the role of the host, meaning their game runs the authoritative version of the game state. This can lead to unfair advantages for the host, such as lower latency and instant processing of inputs. Additionally, if the host disconnects, the entire session can be disrupted unless a proper host migration system is implemented.

P2P games are generally more vulnerable to cheating and hacking than client-server games. Since game logic is processed on individual players' machines rather than a secured server, malicious users can modify game data more easily. Techniques like encrypted packets and validation checks must be implemented to minimize these risks.

Although P2P can reduce latency in ideal conditions, it can also lead to unpredictable performance. If one peer has a poor internet connection, all players may experience lag, as data must travel between multiple endpoints instead of a stable central server.

7.5 Choosing a networking solution for Unity

Unity provides several networking solutions for implementing multiplayer functionality, each with its strengths and weaknesses. The most commonly used options include:

- **Unity Netcode for GameObjects**: Unity's official networking framework, which is still evolving but is tightly integrated with the Unity engine.
- **Photon PUN (Photon Unity Networking)**: A widely used third-party networking solution that offers a cloud-hosted architecture for multiplayer games.
- **Mirror**: An open-source alternative to Unity's deprecated UNET, designed to simplify the development of multiplayer games while providing robust features.
- **Fish-Net**: Another community-driven networking framework that aims to provide high-performance multiplayer solutions.

Each option was evaluated based on its ease of use, performance, and compatibility with our game design.

7.6 Why we chose Mirror for *Kangaroo Planet*

After researching different networking solutions, we selected **Mirror** as the best option for implementing P2P multiplayer in *Kangaroo Planet*. Several key factors influenced this decision:

Mirror is designed as a drop-in replacement for Unity's old networking system (UNET). This makes it relatively easy to integrate into a Unity project, especially compared to more complex solutions like Photon or Fish-Net. Its API is straightforward and well-documented, allowing for quick implementation of core multiplayer features.

Unlike some proprietary networking solutions that require licensing fees or impose usage restrictions, Mirror is fully open-source. This allows for greater flexibility in modifying the framework to suit the game's specific needs. Additionally, Mirror has an active development community, ensuring ongoing support and updates.

Mirror is particularly well-suited for games with a small number of players per session, which aligns with *Kangaroo Planet*'s design. Since our multiplayer mode is not designed for massive online matches, we do not require the scalability of a client-server model.

Mirror includes a variety of transport options, allowing developers to fine-tune networking performance. It also provides built-in solutions for handling data synchronization, message reliability, and network security, reducing the need for custom implementations.

7.7 Multiplayer code implementation in *Kangaroo Planet*

Developing multiplayer functionality in *Kangaroo Planet* required modifying core mechanics to ensure smooth synchronization between clients while maintaining responsiveness. This section analyzes key parts of the multiplayer code, covering player movement, shooting mechanics, enemy AI, host migration, and latency handling.

Synchronizing player movement in a multiplayer environment

Player movement must be consistent across all clients while remaining responsive. This is handled in `PlayerController.cs` using Mirror's networking system:

```
[SyncVar] private Vector3 syncVelocity;
[SyncVar] private Quaternion syncRotation;

void FixedUpdate() {
    if (isLocalPlayer) {
        Vector3 moveInput = new Vector3(Input.GetAxis("Horizontal"), 0
        Vector3 moveVelocity = moveInput.normalized * moveSpeed;
        CmdMove(moveVelocity);

        myRigidbody.MovePosition(myRigidbody.position + moveVelocity *
    } else {
        myRigidbody.velocity = syncVelocity;
        transform.rotation = syncRotation;
    }
}

[Command]
void CmdMove(Vector3 newVelocity) {
    syncVelocity = newVelocity;
}
```

Explanation:

- The local player processes movement input and updates the Rigidbody accordingly.
- `CmdMove()` sends movement updates to the server.
- The `SyncVar` ensures updates are propagated to all clients efficiently.

By using `SyncVar`, we reduce network traffic compared to continuous RPC updates while ensuring all players see accurate movement.

Implementing gun mechanics in multiplayer

Weapons must be synchronized so that shooting and reloading actions are visible to all players. The `GunController.cs` script ensures this:

```
public void OnTriggerHold() {
    if (isLocalPlayer && equippedGun != null) {
        CmdShoot();
    }
}

[Command]
void CmdShoot() {
    if (equippedGun != null) {
        equippedGun.GetComponent<Gun>().OnTriggerHold();
        RpcShoot();
    }
}

[ClientRpc]
void RpcShoot() {
    if (!isLocalPlayer && equippedGun != null) {
        equippedGun.GetComponent<Gun>().OnTriggerHold();
    }
}
```

Explanation:

- `OnTriggerHold()` is called when a player shoots.
- `CmdShoot()` ensures shooting is processed on the server.
- `RpcShoot()` replicates the action for all clients.

This structure prevents cheating by ensuring that shooting actions are validated on the server.

Implementing enemy AI with multiplayer support

Enemy AI must be controlled centrally (on the server) to prevent desynchronization and exploits. The `Enemy.cs` script ensures this:

```
IEnumerator UpdatePath() {
    float refreshRate = 0.25f;

    while (hasTarget) {
        if (currentState == State.Chasing) {
            Vector3 targetPosition = target.position;
            pathfinder.SetDestination(targetPosition);
            CmdSyncEnemyPosition(targetPosition);
        }
        yield return new WaitForSeconds(refreshRate);
    }
}

[Command]
void CmdSyncEnemyPosition(Vector3 position) {
    RpcSyncEnemyPosition(position);
}

[ClientRpc]
void RpcSyncEnemyPosition(Vector3 position) {
    if (!isServer) {
        pathfinder.SetDestination(position);
    }
}
```

Explanation:

```
}
```

- The server updates the enemy's target position every 0.25 seconds.
- The enemy's position is synchronized across clients.
- Since the logic runs on the server, cheating attempts are ineffective.

This guarantees that all players see the same enemy behavior.

Handling host migration

One major issue with Peer-to-Peer networking is host disconnection. If the host leaves the game, all players would typically be disconnected. To address this, Mirror supports host migration, allowing another client to take over the host role.

```
void OnServerDisconnect(NetworkConnection conn) {
    if (NetworkServer.connections.Count > 1) {
        NetworkConnection newHost = NetworkServer.connections[0];
        NetworkServer.SetClientReady(newHost);
    }
}
```

Explanation:

- When the host disconnects, a new host is assigned.
- `NetworkServer.SetClientReady()` ensures the new host has proper authority.

Managing latency and lag compensation

P2P multiplayer is vulnerable to latency issues. To mitigate this, we use lag compensation techniques.

```
[Command]
void CmdMove(Vector3 newVelocity, double timestamp) {
    double latency = NetworkTime.time - timestamp;
    Vector3 predictedPosition = newVelocity * (float)latency;
    syncVelocity = newVelocity;
    RpcUpdatePosition(predictedPosition);
}

[ClientRpc]
void RpcUpdatePosition(Vector3 predictedPosition) {
    if (!isLocalPlayer) {
        transform.position += predictedPosition;
    }
}
```

Explanation:

- When a player moves, their timestamp is sent to the server.
- The server calculates the latency and predicts movement.
- The updated position is sent back to clients to maintain consistency.

This reduces the impact of network lag, ensuring a smoother experience.

With these key networking components in place, we have established a solid foundation for multiplayer gameplay in *Kangaroo Planet*. The next section explores the conclusions drawn from our implementation and potential future improvements.

7.8 Conclusion

The decision to implement P2P multiplayer in *Kangaroo Planet* was made after careful evaluation of different networking models. While P2P presents challenges such as host advantage and security concerns, it also provides benefits such as reduced server costs and lower latency in ideal conditions.

After analyzing multiple networking solutions for Unity, we determined that Mirror was the best fit for our project due to its ease of use, open-source nature, and strong community support. With this foundation in place, the next phase of development focused on implementing core multiplayer functionality, including player synchronization, movement replication, and handling game events across the network.

Chapter 8

Weeks 13 and 14 - Finalization of P2P Multiplayer (Hélios)

In the final two weeks, we completed the multiplayer implementation using the `Mirror` networking library, combined with the lightweight and efficient `KCP Transport`. The game now supports a competitive 1v1 mode, with each player starting with 50 health points. The experience is now fully synchronized, with one peer hosting the match and authoritative server logic handling all critical gameplay events.

To support this new multiplayer mode, we created a revised version of the original `Player.cs` script, now renamed and restructured as `MPlayer.cs`. This script governs all aspects of a networked player: movement, aiming, shooting, and health.

8.1 Structure of `MPlayer.cs`

The `MPlayer` class inherits from `NetworkBehaviour`, which is essential when using Mirror for handling network identity and authority. At the top, we ensure that our player GameObject includes the necessary movement and shooting components:

```
[RequireComponent(typeof(MPlayerController))]
[RequireComponent(typeof(MGunController))]
public class MPlayer : NetworkBehaviour {
```

These attributes automatically add `MPlayerController` and `MGunController` to any GameObject that uses this script, ensuring the player has both movement and gun functionality.

Next, we declare the player's movement speed and max health, as well as a synchronized variable for health across the network:

```
    public float moveSpeed = 5;
    public int maxHealth = 50;

    [SyncVar]
```

```
public int currentHealth;
```

The [SyncVar] attribute tells Mirror to automatically keep this variable in sync across all clients. Any change to currentHealth on the server is reflected on all connected clients in real time.

We also define local references for components that the player will need:

```
Camera viewCamera;
MPlayerController controller;
MGunController guncontroller;
```

The Start() method initializes these references. It sets up the camera only for the local player and initializes the health to its maximum value:

```
void Start() {
    controller = GetComponent<MPlayerController>();
    guncontroller = GetComponent<MGunController>();

    if (isLocalPlayer) {
        viewCamera = Camera.main;
    }

    currentHealth = maxHealth;
}
```

Only the local player needs a camera to render their own view. The check for isLocalPlayer ensures that other players don't attempt to control or assign the main camera.

The Update() method handles input processing and is executed only on the local player instance to avoid conflicts or multiple players acting on one object:

```
void Update() {
    if (!isLocalPlayer) return;
```

Player movement is captured using Unity's input system and normalized to maintain consistent movement speed regardless of direction:

```
    Vector3 moveInput = new Vector3(Input.GetAxisRaw("Horizontal")
    Vector3 moveVelocity = moveInput.normalized * moveSpeed;
    controller.Move(moveVelocity);
```

The player also rotates to face the mouse cursor, using a raycast from the camera to the ground plane:

```
        Ray ray = viewCamera.ScreenPointToRay(Input.mousePosition);
        Plane groundPlane = new Plane(Vector3.up, Vector3.zero);
        float rayDistance;
        if (groundPlane.Raycast(ray, out rayDistance)) {
            Vector3 point = ray.GetPoint(rayDistance);
            controller.LookAt(point);
        }
```

Shooting is triggered by the left mouse button. Rather than shooting directly from the client (which would be unsafe), the player sends a command to the server:

```
        if (Input.GetMouseButton(0)) {
            CmdShoot();
        }
    }
```

The CmdShoot() method is a Mirror [Command], meaning it is called by the client but executed on the server. This keeps gameplay logic secure and centralized:

```
    [Command]
    void CmdShoot() {
        guncontroller.Shoot();
    }
```

Damage and death are also server-authoritative. The TakeDamage() method is called on the server and reduces health, only triggering Die() if the player's health reaches zero:

```
    [Server]
    public void TakeDamage(int damage) {
        if (currentHealth <= 0) return;

        currentHealth -= damage;
        if (currentHealth <= 0) {
            Die();
        }
    }
```

Once the player dies, their object is destroyed on the network, removing them from the match:

```
    [Server]
    void Die() {
        NetworkServer.Destroy(gameObject);
    }
}
```

8.2 Weapon System for Multiplayer

To support shooting in the networked game, we introduced two new scripts: `MGun.cs` and `MGunController.cs`. These are modified versions of our original single-player weapon scripts, updated to handle server-authoritative shooting and object spawning using Mirror.

The `MGun` class handles the core shooting mechanism. It is responsible for instantiating and launching projectiles, ensuring that only the server handles the spawning to maintain proper network synchronization. We begin by declaring the essential references and variables:

```
public Transform muzzle;
public MProjectile projectile;
public float msBetweenShots = 100;
public float muzzleVelocity = 35;
```

- `muzzle` is the point from which the projectile is instantiated.
- `projectile` is a reference to the projectile prefab that will be instantiated when firing.
- `msBetweenShots` defines the delay (in milliseconds) between two shots to avoid spamming.
- `muzzleVelocity` controls how fast the projectile is launched.

We use a private variable to track the cooldown between shots:

```
float nextShotTime;
```

The actual shooting method checks if the cooldown has passed, instantiates the projectile, sets its velocity, and uses Mirror's `NetworkServer.Spawn()` to synchronize it across the network:

```
public void Shoot() {
    if (Time.time > nextShotTime) {
        nextShotTime = Time.time + msBetweenShots / 1000f;
        MProjectile newProjectile = Instantiate(projectile, muzzle.pos
        newProjectile.SetSpeed(muzzleVelocity);
        NetworkServer.Spawn(newProjectile.gameObject);
    }
}
```

- The cooldown is managed using `Time.time` and `msBetweenShots`.
- The projectile is spawned at the muzzle's position and orientation.
- The call to `NetworkServer.Spawn()` ensures the projectile exists for all players and behaves identically.

The `MGunController` script is attached to the player object and manages equipping and using weapons. First, we declare the weapon slot and the starting weapon:

```
public Transform weaponHold;
public MGun startingGun;
MGun equippedGun;
```

- `weaponHold` defines where the gun should be attached on the player.
- `startingGun` allows assigning a default weapon in the Inspector.
- `equippedGun` stores the current active weapon instance.

When the player object is initialized, we equip the starting weapon if one is assigned:

```
void Start() {
    if (startingGun != null) {
        EquipGun(startingGun);
    }
}
```

The `EquipGun()` method handles replacing the current weapon with a new one. If a weapon is already equipped, it is first destroyed to avoid duplicates:

```
public void EquipGun(MGun gunToEquip) {
    if (equippedGun != null) {
        Destroy(equippedGun.gameObject);
    }
    equippedGun = Instantiate(gunToEquip, weaponHold.position, weaponH
    equippedGun.transform.parent = weaponHold;
}
```

Finally, the `Shoot()` method simply delegates the action to the currently equipped gun:

```
public void Shoot() {
    if (equippedGun != null) {
        equippedGun.Shoot();
    }
}
```

This modular approach allows us to easily add support for multiple weapon types in the future. By isolating the firing logic in `MGun`, and the management logic in `MGunController`, we maintain clear separation of responsibilities.

8.3 Conclusion

This final implementation of `MPlayer.cs` concludes the multiplayer integration. By leveraging Mirror's networking model, we were able to enforce authoritative logic on the server, synchronize player states, and handle inputs and gameplay interactions in a secure and deterministic way. The result is a responsive 1v1 mode where each action—from movement to death—is managed cleanly and consistently across both peers. This foundation leaves the door open for expanding to more players or more complex interactions in the future.

Chapter 9

Weeks 15 and 16 - Training Mode : Infinite Level and Infinite Wave System (Nicolas)

To allow players to practice their skills in a controlled but endlessly challenging environment, we implemented a specialized **Training Mode**. This mode features an infinite level with a single wave that never ends, constantly spawning enemies at a fixed rate. The implementation centers around the `Spawner.cs` script, which handles both finite wave gameplay and this infinite loop.

9.1 Wave System Overview

In our game, enemy spawning is handled by the `Spawner` component. The main structure of the spawning logic revolves around a list of `Wave` objects. Each wave defines parameters such as:

- `enemyCount` – total number of enemies to spawn in the wave.
- `timeBetweenSpawns` – delay between each enemy spawn.
- `moveSpeed`, `enemyHealth`, and `hitsToKillPlayer` – define enemy difficulty.
- `skinColour` – provides a visual identity for the wave.
- `infinite` – if `true`, spawns enemies forever.

Training Mode works by defining a wave with the `infinite` flag set to `true`. This tells the `Spawner` to disregard the enemy count and continue spawning enemies indefinitely.

9.2 Training Mode Initialization

The system starts in the `Start()` method:

```
void Start() {
    playerEntity = FindObjectOfType<Player>();
    playerT = playerEntity.transform;
    ...
    NextWave();
}
```

This method sets up references to the player and the map, hooks into the player's death event, and initiates the first wave (which, in Training Mode, is an infinite wave).

9.3 Infinite Spawning Logic

The infinite behavior is implemented within the `Update()` loop:

```
if ((enemiesRemainingToSpawn > 0 || currentWave.infinite)
    && Time.time > nextSpawnTime) {
    enemiesRemainingToSpawn--;
    nextSpawnTime = Time.time + currentWave.timeBetweenSpawns;
    StartCoroutine("SpawnEnemy");
}
```

In standard waves, spawning only continues while `enemiesRemainingToSpawn > 0`. But in an infinite wave, the `currentWave.infinite` condition is always true, allowing the system to continuously spawn new enemies at the configured interval.

Even though `enemiesRemainingToSpawn-` still executes, it does not affect the flow because the infinite flag overrides the termination condition.

9.4 Camping Detection

To discourage players from camping in a single spot, the `Spawner` includes logic to monitor player movement:

```
if (Time.time > nextCampCheckTime) {
    nextCampCheckTime = Time.time + timeBetweenCampingChecks;
    isCamping = (Vector3.Distance(playerT.position, campPositionOld)
                 < campThresholdDistance);
    campPositionOld = playerT.position;
}
```

If the player hasn't moved more than `campThresholdDistance` between checks, the system flags them as camping. When camping, the next enemy will spawn directly at the player's location instead of a random tile:

```
Transform spawnTile = isCamping
    ? map.GetTileFromPosition(playerT.position)
    : map.GetRandomOpenTile();
```

9.5 Enemy Spawn Routine

Spawning is handled as a coroutine, which adds a delay and visual effect before the enemy appears:

```
IEnumerator SpawnEnemy() {
    ...
    while (spawnTimer < spawnDelay) {
        tileMat.color = Color.Lerp(initialColour, flashColour,
            Mathf.PingPong(spawnTimer * tileFlashSpeed, 1));
        ...
    }

    Enemy spawnedEnemy = Instantiate(enemy, spawnTile.position + Vecto
    ...
}
```

- The spawn tile flashes red before the enemy appears.
- After spawning, the enemy is configured with the current wave's difficulty settings using `SetCharacteristics()`.

9.6 Wave Progression and Locking

In normal gameplay, once all enemies in a wave are killed, `NextWave()` is called:

```
void OnEnemyDeath() {
    enemiesRemainingAlive--;
    if (enemiesRemainingAlive == 0) {
        NextWave();
    }
}
```

But in Training Mode, because the wave is infinite and enemies are spawned endlessly, `enemiesRemainingAlive` never reaches zero, so the wave never ends and `NextWave()` is never triggered again. This creates the intended endless loop.

9.7 Developer Tools

A development feature allows manual wave progression and reset using the Return key:

```
if (devMode) {
    if (Input.GetKeyDown(KeyCode.Return)) {
        StopCoroutine("SpawnEnemy");
        foreach (Enemy enemy in FindObjectsOfType<Enemy>()) {
            GameObject.Destroy(enemy.gameObject);
        }
        NextWave();
    }
}
```

This tool is useful during testing but is disabled in production builds.

9.8 Conclusion

Training Mode is a minimalist but highly effective implementation that takes advantage of existing wave mechanics. By defining a wave with the `infinite` flag set, we create a non-stop combat experience. The player's skill is tested not by increasingly complex levels, but by the sheer volume and relentlessness of the enemies. Through smart reuse of the wave logic and minor condition checks, we deliver an infinitely replayable mode with minimal additional code complexity.

Chapter 10

Work organization and team dynamics

The development of this project took place over several weeks, during which we worked closely together as a team. Here is an overview of our organization and the dynamic that allowed us to make progress, despite some initial challenges.

10.1 Weekly meetings

To maintain optimal coordination and ensure regular tracking of the project's progress, we decided to hold a weekly meeting. These meetings were held every week at the EPITA cafeteria, a place we chose for its friendly atmosphere and accessibility.

During these meetings, we discussed ongoing tasks, any issues encountered, and goals for the following week. Each team member presented their progress, and we collectively debated the priorities to be set. This working method allowed us to stay aligned and progress in a structured manner.

To structure the work, we divided the project into several stages corresponding to the different systems to be developed (enemy management, visual effects, audio, etc.). Each member was assigned specific responsibilities while remaining flexible to help each other when needed.

- **Each meeting started with a recap of the previous week's objectives,** to verify if the assigned tasks had been completed.
- **A part of the meeting was dedicated to identifying obstacles,** whether technical or organizational.
- Finally, we defined **the priorities for the upcoming week** and reassigned tasks accordingly.

10.2 Team challenges

Although our organization was effective overall, we encountered some difficulties, particularly at the beginning of the project. An unexpected disagreement disrupted our initial dynamic: one of the team members, over a topic as unlikely as kangaroos, decided to leave the group. This departure, following a difference of opinion, reduced our team from five to four members.

This moment was a challenge for the team, as it forced us to redistribute the missing member's tasks and revise our organization. However, this episode also strengthened our bond as a team, reinforcing our ability to collaborate and resolve conflicts constructively.

10.3 A collective effort and an enriching experience

Despite the obstacles encountered, working on this project has been an overall positive experience. Each team member contributed their skills and creativity, helping to advance the project at every stage. The weekly meetings at the EPITA cafeteria quickly became more than just work sessions: they allowed us to share ideas, debate technical choices, and even laugh about the most improbable enemies we could add to the game.

In the end, this project was not just a technical exercise but also a true human adventure. Working together, overcoming challenges, and seeing our vision come to life has been a source of satisfaction for each of us. Even though everything was not always easy, the joy of collaborating and creating together will remain the most valuable memory of this experience.

10.4 The absence of a version manager

One of the most significant technical mistakes we made was not implementing a version control system from the beginning of the project. At several points during development, we found ourselves manually copying files between computers or using cloud storage platforms like Google Drive or OneDrive.

This lack of version control led to several problems:

- Conflicts when multiple people modified the same script without realizing it.

- Lost work due to overwritten files or untracked changes.
- Difficulty in rolling back changes or understanding the history of modifications.

Looking back, integrating a Git-based workflow with platforms like GitHub or GitLab would have improved our collaboration, made our work more reliable, and avoided unnecessary confusion. A branching strategy such as Git Flow would have allowed us to work independently on features while maintaining a stable main branch for integration.

Version control is not just a technical tool — it's a communication layer. It provides a shared timeline of the project's evolution, something we deeply missed at times.

10.5 The need for more communication

Another recurring problem was the lack of regular communication outside of weekly meetings. Important decisions were sometimes taken individually and not discussed with the group. For example, some teammates implemented features without clearly specifying how others were supposed to use or integrate them, which led to misunderstandings and additional rework.

Improving communication could have solved several issues:

- Avoiding duplicated work.
- Clarifying expectations and responsibilities.
- Identifying integration problems earlier in the development cycle.

In hindsight, using a shared chat tool (e.g., Discord, Slack) combined with short daily check-ins (even just messages) would have made a significant difference in team cohesion and technical alignment.

10.6 Insufficient planning and task scoping

We initially jumped into coding with enthusiasm but without a clear, structured roadmap. Tasks were defined on the fly, often based on what seemed interesting rather than what was essential. This led to uneven progress and moments of confusion where we weren't sure what to prioritize next.

The absence of a precise timeline with defined milestones made it difficult to track our progress effectively. Some systems were started early but never finished, while others were rushed near the deadline.

A better approach would have been:

- Defining core features and stretch goals from the start.
- Creating a visual roadmap or Gantt chart to assign milestones.
- Reviewing and adjusting the plan weekly based on our actual progress.

Proper project management, even in a light form, would have improved our efficiency and allowed us to detect issues earlier.

10.7 Multiplayer added at the last minute

One of the most technically ambitious aspects of the game — multiplayer support — was only integrated at the very end of the project. It was something we always intended to do, but we kept postponing it due to its perceived complexity.

In practice, adding multiplayer at the end introduced a lot of integration problems:

- We had to refactor several existing systems to support networked behaviors.
- Many bugs emerged due to the switch from local to synchronized state.
- We were forced to cut features we had already implemented but couldn't easily adapt to multiplayer.

If we had started with multiplayer as a constraint from the beginning, our architecture would have been better prepared, and integration would have been smoother. Instead, it felt like we were retrofitting an entire second game onto the one we had already built.

10.8 Overly ambitious goals and unrealistic complexity

Another major lesson from this project is the importance of setting realistic goals. We aimed to create a procedurally generated universe, complete with custom planets, randomized biomes, and unique map generation at runtime.

While these ideas were exciting on paper, they turned out to be overly ambitious for the time and resources we had. In particular:

- Generating playable maps on the fly introduced huge technical challenges.
- Designing a unique universe with consistent lore and assets became overwhelming.
- Many of these features were either dropped or implemented in a simplified form.

A simpler and more focused project would have allowed us to polish fewer systems to a higher level of quality. We now understand the value of building a solid core experience before expanding it with complex mechanics.

10.9 Conclusion and takeaways

This project was a success in many ways, but it also offered valuable lessons about team organization, scope management, and planning. If we were to restart from scratch, we would do several things differently:

- Set up a version control system immediately.
- Communicate daily, not weekly.
- Plan the project more thoroughly before coding.
- Start with multiplayer if it's a key feature.
- Choose a smaller, more achievable scope.

Yet, despite our mistakes, the experience was deeply rewarding. We not only improved our technical skills but also grew as collaborators and project managers. These insights will carry over into our future projects and help us build even better games — both in code and in teamwork.

Chapter 11

Scope for improvement

Although the project has reached an advanced and playable state, several enhancements are still planned to further refine the experience and align the game with its full creative and technical potential. These upcoming tasks aim to polish the gameplay, deepen immersion, and introduce new features that respond to player expectations and feedback.

11.1 Game environment

One of the most significant upcoming visual upgrades is the transformation of the terrain to resemble the surface of the moon. This change will strengthen the game's atmosphere and visual identity, grounding the experience in a science-fiction-inspired aesthetic.

The moon-like ground will be created using layered textures and advanced shaders to accurately portray its rugged, cratered surface. Craters, dust effects, and occasional dynamic elements such as falling rocks or ambient particles will be introduced to breathe life into the environment. Special emphasis will be placed on realistic lighting to simulate moonlight, shadows, and subtle reflections.

This visual foundation will serve not just as a background, but as an integral component of the game's tone and identity.

11.2 Multiplayer improvements

The core peer-to-peer multiplayer system has been implemented and is functional. Players can now connect and play together in real time, sharing a synchronized game state. However, there is still room for significant improvement.

One of the planned additions is a **life bar display in multiplayer**, allowing players to see each other's health and coordinate support more effectively during intense fights. Visual cues, such as flashing outlines or color transitions, will indicate critical health states to improve team dynamics.

Additional improvements will focus on smoother matchmaking, more intuitive player feedback during connection, and better handling of rare edge cases such as mid-game joins, disconnections, and reconnections. Minor latency issues will continue to be addressed, and playtesting will help identify pain points in peer synchronization and input lag.

11.3 Gameplay mechanics and balancing

While core mechanics are in place, we identified several areas where gameplay can be improved and made more dynamic.

A major addition under development is a **manual weapon switching system**, allowing players to carry and swap between different weapons. This will introduce new strategic elements, letting players adapt their approach in real time depending on the enemy type or environment. The interface will be designed to make weapon management fluid and unobtrusive.

We also plan to introduce a variety of **power-ups** that spawn throughout levels. These could include temporary damage boosts, speed enhancements, shields, or special projectiles. Power-ups will encourage exploration and introduce variability into combat, creating more moments of tension and excitement.

Another critical focus is **difficulty balancing**. Currently, difficulty spikes too rapidly in some levels, while others feel underwhelming. To fix this, we will:

- Introduce scaling difficulty based on the number of players in multiplayer.
- Adjust spawn rates, enemy behaviors, and pacing according to player progression.
- Refine the parameters of the infinite training mode to keep it challenging without being overwhelming.

Fine-tuning the challenge curve is essential to keeping players engaged while avoiding frustration or boredom.

11.4 Main mode completion

The predefined levels of the main game mode are nearing completion, but additional work remains to ensure cohesion, polish, and pacing.

These levels will undergo iterative refinement to integrate the new features mentioned above—such as power-ups and manual weapon switching—while maintaining a coherent challenge ramp.

Each level will be reviewed with a critical eye toward gameplay variety, aesthetic consistency, and narrative flow. We aim to embed more visual storytelling into the environments and enemy placement, turning each level into a memorable segment of the player's journey.

Additionally, the inclusion of moon terrain, lighting effects, and more detailed map geometry will enhance the uniqueness of each level.

11.5 Summary of remaining work

The main upcoming tasks include:

- **Implementing a moon-like ground and enhancing environmental detail** to improve immersion and world-building.
- **Refining multiplayer features**, particularly the addition of visible life bars and improved connectivity handling.
- **Balancing difficulty** across all game modes to ensure a fair and engaging challenge curve.
- **Adding manual weapon switching and power-ups** to deepen gameplay and promote tactical decisions.
- **Polishing the predefined levels**, both in terms of design and presentation, to ensure a consistent and rewarding experience.

With these additions, the project is set to transition from a solid prototype into a refined and cohesive game. The team remains committed to delivering a finished product that combines technical quality with creative ambition. As we approach the final development phase, our focus will be on attention to detail, polish, and user experience.

Chapter 12

Game preview

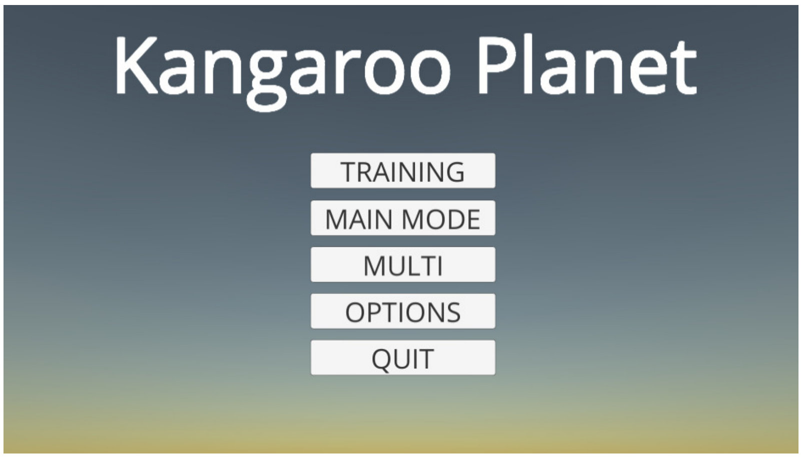

Figure 12.1: Screenshot of the main menu.

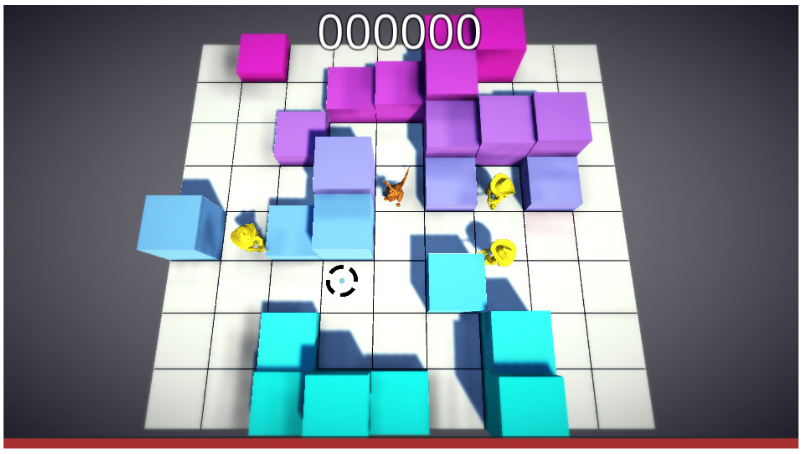

Figure 12.2: Screenshot of the gameplay.

Chapter 13

Conclusion

Throughout the development of *Kangaroo Planet*, our team has navigated a complex and ambitious journey—transforming a creative concept into a fully playable and engaging experience. This report has detailed the technical systems, design decisions, and collaborative efforts that shaped the project, from procedural map generation and multiplayer architecture to visual effects, sound design, and user interface development.

Over the course of several weeks, we overcame a wide range of technical and creative challenges. Each phase of the project contributed to our collective growth as developers and collaborators. Whether it was optimizing performance, debugging networking issues, or fine-tuning visual polish, every task brought us closer to the vision we originally set out to achieve.

The current state of the game reflects both our accomplishments and our potential. Core systems such as enemy AI, real-time multiplayer, dynamic audio, and responsive controls are fully functional and integrated. At the same time, we recognize that game development is an iterative process—further improvements, refinements, and polish will continue beyond this milestone.

Ultimately, this project has been a demonstration of both technical skill and creative expression. It required careful coordination, continuous iteration, and a strong sense of teamwork. We are proud of what we have built and confident in the foundation we've laid. *Kangaroo Planet* is not only a game but a testament to our ability to transform an unconventional idea into a compelling and coherent product.

As we conclude this phase, we remain committed to refining the experience further. With the major systems in place, our focus will now shift toward optimization, final balance tweaks, and polish—ensuring that every detail supports an enjoyable, immersive, and unforgettable gameplay experience.