Neighborhood Defense

Programming for 3D

2025–2026

University of Sussex

Íñigo Martínez Jiménez

Introduction

The game Neighborhood Defense is a survival first-person shooter (FPS). The narrative premise of the game is that the character faces beings known as ‘The Glitches’.

The player faces waves of enemies that arrive with each phase of the attack, carrying a weapon with which to fight the enemies. To overcome these waves, it will be necessary to use shooting skills, be agile, and have a good strategy to survive in a residential environment that, far from being peaceful, becomes a battlefield.

The narrative unfolds within an ordinary neighborhood, to which the character belongs, with the intention of defending it at all costs. Key mechanics are implemented within the game, such as physical projectiles using Rigidbody and collision, advanced artificial intelligence that uses NavMesh for enemy movements, and a hybrid level design that combines modelled and imported elements.

Planning

To ensure the smooth development of the video game, I used the Agile methodology with a Kanban approach. This allowed me to divide the work into small, specific tasks and then distribute them throughout the project.

This way, I could ensure that each part of the game was ready at the right time. It also allowed for quick adjustments when necessary, depending on how the project progressed. This helps to keep everything under control and ensures that the project progresses properly.

The three main sprints used in development were:

1- Basic movement and shooting

The main objective was to implement the character's movement mechanics and the basic shooting system. During this phase, the fundamental scripts were created so that the player could move around the map and shoot enemies. This stage was crucial in establishing a solid foundation for gameplay.

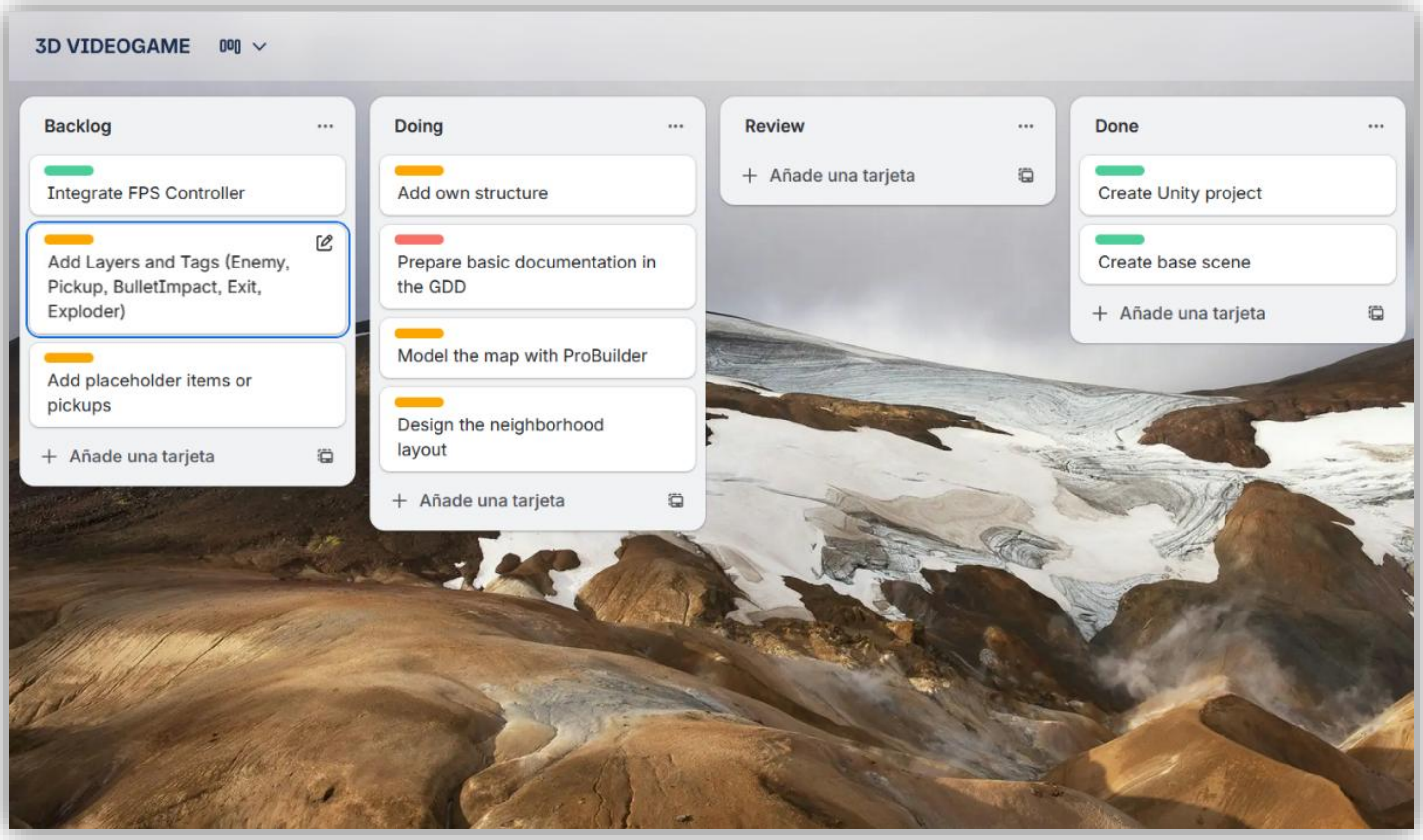
2- Modelling the safe house and park

This focused on creating 3D models of important locations in the game. These locations are the main house in the neighborhood and the park, which is an extra area where you can fight... Texture work was also carried out in this sprint, and the models were put into the game engine.

3- Enemy AI and round system

The third sprint focused on artificial intelligence for enemies. This allowed enemies to move around the map on their own, thanks to NavMesh. I also added the round system, where the player must survive waves of increasingly difficult enemies. This phase was very important for the game, as it made the difficulty increase gradually.

In the Trello screenshots, you can see how the tasks were distributed across the Backlog, Doing, Review, and Done columns, reflecting weekly progress. The first screenshot (*Figure 1*) shows the board at the start of the project, where the tasks were clearly defined and in the process of being started. The second screenshot (*Figure 2*) shows the progress of the project a few weeks later, with many tasks completed and the focus on the final stages of development.



3D VIDEOGAME

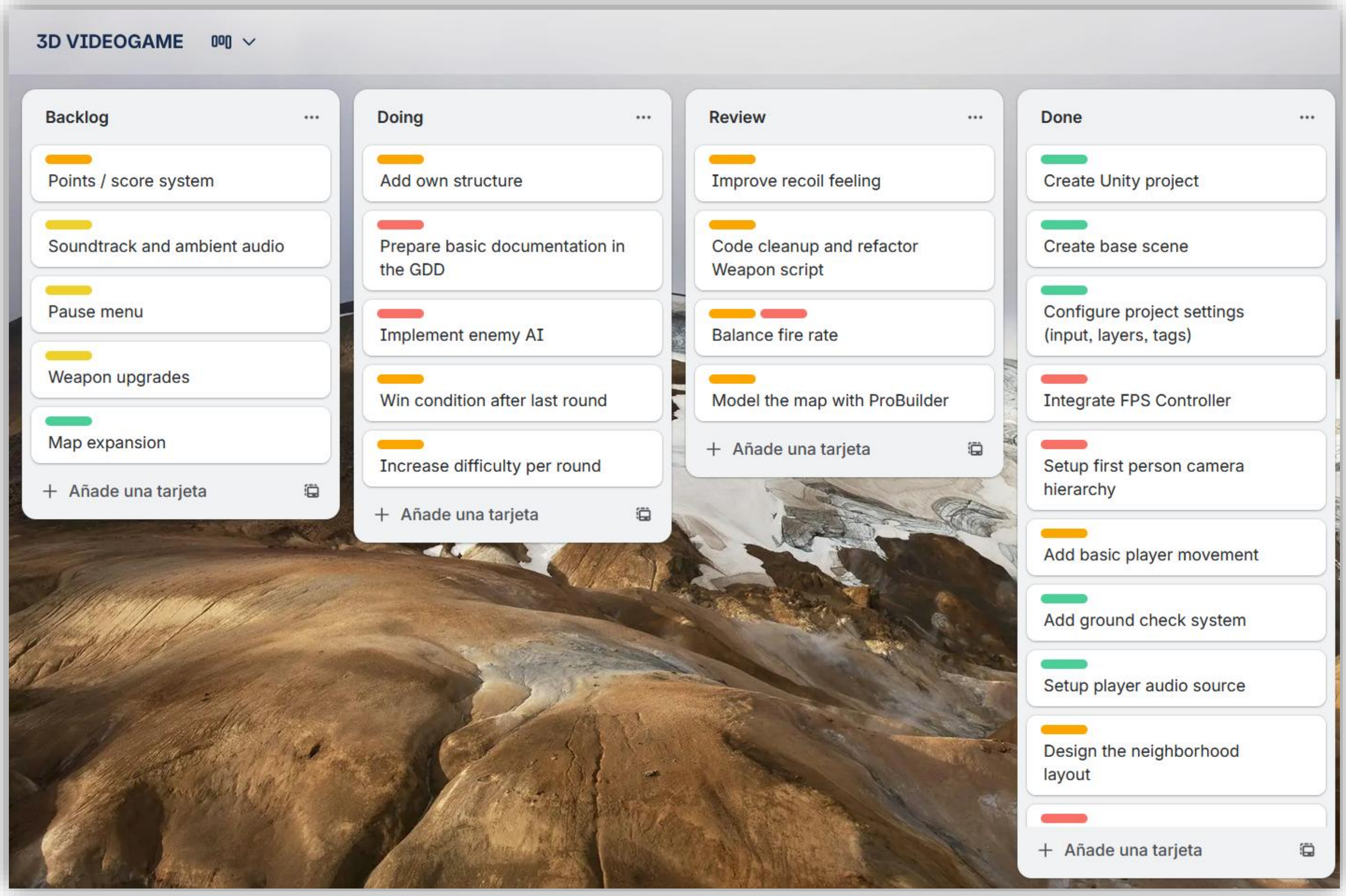
Backlog
- Integrate FPS Controller
- Add Layers and Tags (Enemy, Pickup, BulletImpact, Exit, Exploder)
- Add placeholder items or pickups
- + Añade una tarjeta

Doing
- Add own structure
- Prepare basic documentation in the GDD
- Model the map with ProBuilder
- Design the neighborhood layout
- + Añade una tarjeta

Review
- + Añade una tarjeta

Done
- Create Unity project
- Create base scene
- + Añade una tarjeta

Figure 1



3D VIDEOGAME

Backlog
- Points / score system
- Soundtrack and ambient audio
- Pause menu
- Weapon upgrades
- Map expansion
- + Añade una tarjeta

Doing
- Add own structure
- Prepare basic documentation in the GDD
- Implement enemy AI
- Win condition after last round
- Increase difficulty per round
- + Añade una tarjeta

Review
- Improve recoil feeling
- Code cleanup and refactor Weapon script
- Balance fire rate
- Model the map with ProBuilder
- + Añade una tarjeta

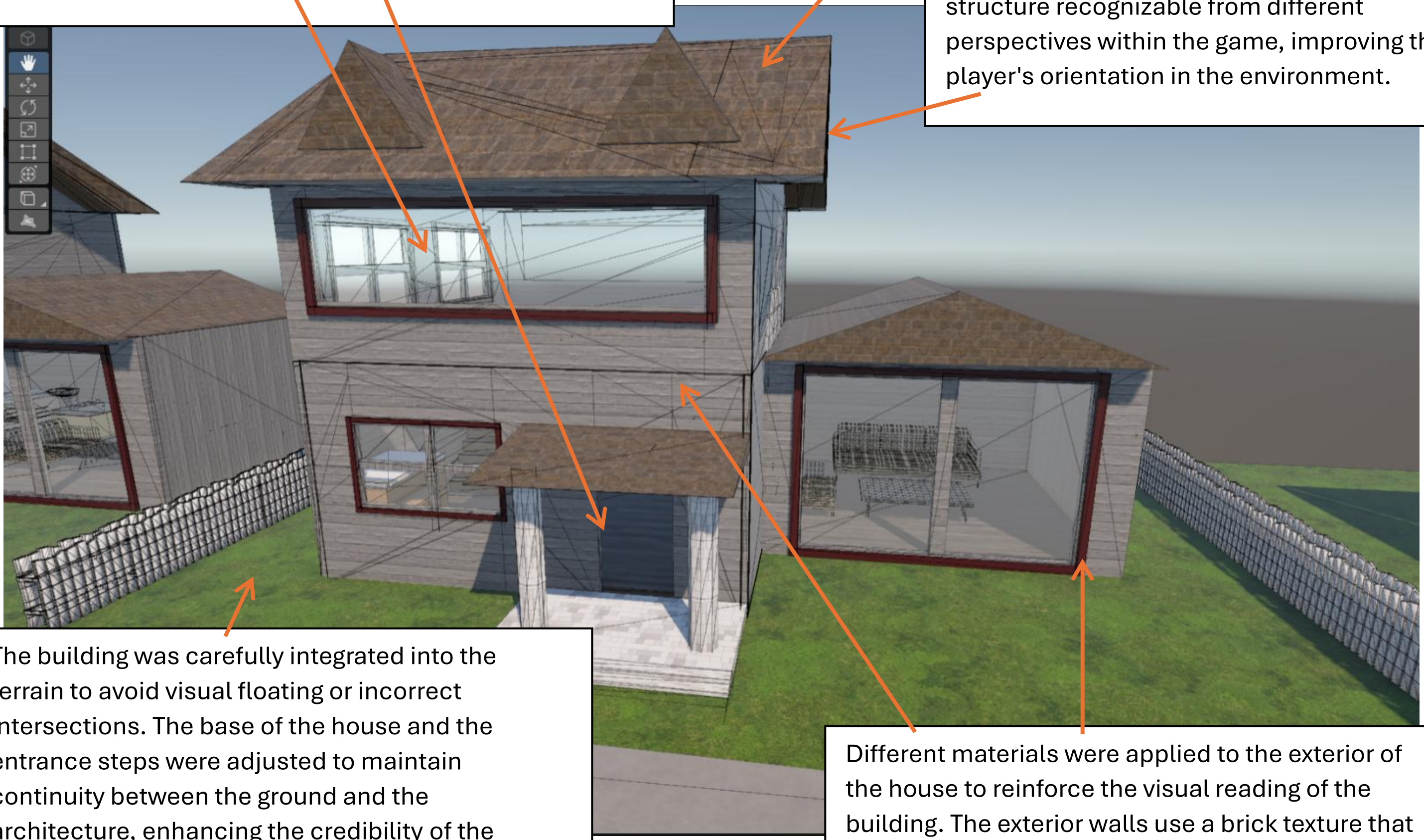
Done
- Create Unity project
- Create base scene
- Configure project settings (input, layers, tags)
- Integrate FPS Controller
- Setup first person camera hierarchy
- Add basic player movement
- Add ground check system
- Setup player audio source
- Design the neighborhood layout
- + Añade una tarjeta

Figure 2

Modelling

The dimensions of doors and windows were adjusted using the player's Character Controller as a reference. This ensures that the player can pass through entrances without unexpected collisions and that the height of windows and frames maintains a realistic scale. Scale checking was performed from a first-person view during the modelling process.

The composition of the roof and the lateral extension were designed to break up the cubic silhouette of the building and add visual variety. These volumes help make the structure recognizable from different perspectives within the game, improving the player's orientation in the environment.

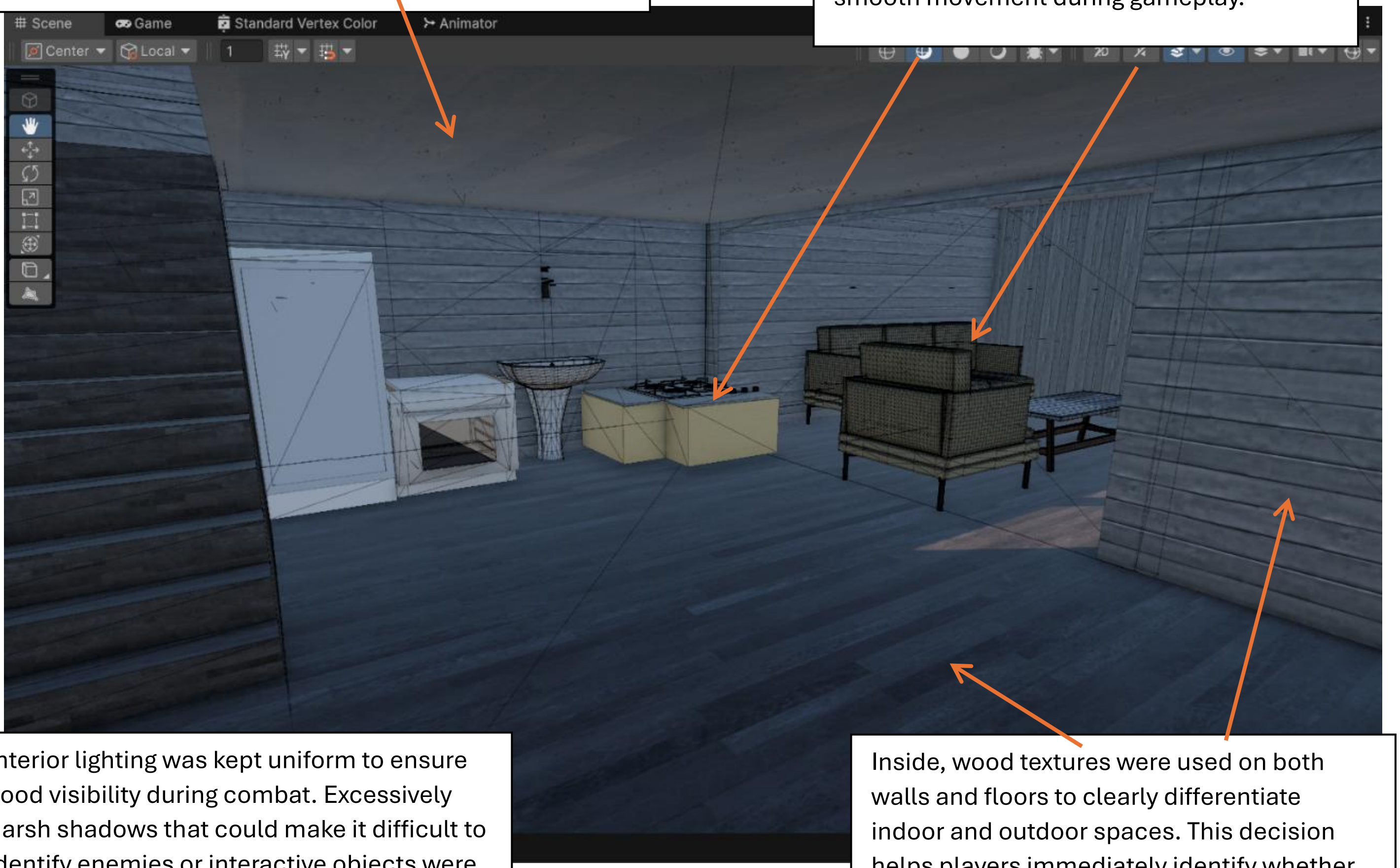


The building was carefully integrated into the terrain to avoid visual floating or incorrect intersections. The base of the house and the entrance steps were adjusted to maintain continuity between the ground and the architecture, enhancing the credibility of the environment.

Different materials were applied to the exterior of the house to reinforce the visual reading of the building. The exterior walls use a brick texture that conveys structural solidity, while the window and door frames use darker materials to create contrast and clearly define the openings.

The ceiling height was adjusted to avoid a feeling of claustrophobia in first person, while maintaining realistic proportions. This aspect was checked directly from the player's camera during development.

The furniture was placed with player navigation and combat in mind. Corridors were kept clear and sufficient space was left between objects to prevent the Character Controller from getting stuck and to allow for smooth movement during gameplay.



Interior lighting was kept uniform to ensure good visibility during combat. Excessively harsh shadows that could make it difficult to identify enemies or interactive objects were avoided.

Inside, wood textures were used on both walls and floors to clearly differentiate indoor and outdoor spaces. This decision helps players immediately identify whether they are inside or outside the building, reinforcing spatial navigation.

The game environment combines imported assets for the background with key structures that I modelled myself. The centerpiece of this work is the ‘Standard House’, which I modelled using ProBuilder in Unity. At this stage, I made sure that the dimensions of the doors were accurate, as they need to match the player's Character Controller perfectly, allowing for smooth movement within the space.

Once the basic structure of the house was ready, I added different textures to distinguish the interior and exterior areas. On the exterior, I used a wood-like texture to give a sense of strength and realism. In contrast, the roofs have a brick-like texture. This creates a contrast that helps the player visually navigate the different spaces in the house. I also added furniture throughout the house to give the environment a touch of realism.

One of the biggest problems I had was with the Mesh Colliders. These are necessary for bullets to pass through windows. At first, the colliders prevented invisible bullets from passing through, which meant that shooting did not work properly. So, I adjusted the colliders on the windows so that bullets could pass through them without any problems and without affecting the player's experience.

In the screenshot provided, you can see the main parts of the modelled house, along with the key areas I discussed, such as the windows and colliders. The arrows indicate the most important parts that were adjusted during the modelling process.

3D Physics

The game's 3D physics system manages the interaction between the player, enemies, and the environment, enabling impact detection, damage application, AI navigation, and visual feedback generation. Physics is not used to simulate real ballistics, but rather as a tool for detecting and controlling gameplay.

Camera-aligned projectile firing system

The firing system is implemented with physical projectiles: when firing, a bullet prefab is instantiated at the muzzlePoint, velocity is applied using Rigidbody.velocity, and the impact is detected by collision. To ensure that the shot matches the crosshairs in both the Editor and the executable, the direction is calculated with a ray generated from the camera using the actual position of the crosshairs on the screen.

In terms of gameplay, this system allows for immediate impact detection. If the object hit has the 'EnemyHealth' component, damage is applied directly by calling the 'TakeDamage(damage)' method. In addition, the use of a LayerMask allows differentiation between enemies and environmental elements, making it possible to react differently depending on the type of surface hit, such as displaying different visual effects or playing specific sounds.

Colliders and Layers System

The colliders and layers system provides a solid foundation for collision detection and the correct identification of objects within the scene. All interactive elements in the game have colliders, allowing them to be consistently detected by the interaction and combat systems.

Enemies use simple colliders, usually Capsule Colliders, as they offer a good balance between precision and stability during movement. Environment elements, such as walls or structures, use colliders adapted to their geometry and level of detail, use simple colliders whenever possible and reserving Mesh Colliders for cases where greater precision is required.

In addition, enemies are assigned to a specific layer, which facilitates their identification within the game logic and allows for differentiated behaviors to be applied in relation to other elements of the environment. This layer-based organization contributes to a clear separation between collision detection and game logic, improving code readability and allowing for greater system scalability as new types of objects or interactions are added.

Rigidbodies and enemy control

The control of enemy movement and behavior is based on an artificial intelligence system supported by navigation, rather than relying on actual physical forces. This allows enemies to move in a controlled and consistent manner within the environment.

NavMeshAgent is used for movement, allowing enemies to move around the environment following predefined navigable areas. The agent's target position is continuously updated so that the enemy pursues the player, and attack behavior is activated based on distance, allowing for clear transitions between states such as pursuit and combat.

From a physical standpoint, enemies have a Rigidbody configured in Kinematic mode, which prevents collisions or pushes derived from physics from interfering with the navigation system. In this way, movement is completely under the control of the AI and NavMesh, eliminating erratic or unpredictable behavior.

Particle system linked to physics

Impact effects are generated when the bullet collides: a VFX is instantiated at the point of contact and oriented with the normal surface, differentiating between impact on enemies and on the environment. VFX are automatically destroyed after their duration to avoid accumulation on the scene.

Visual physics of the weapon

The visual physics of the weapon relies on animations that add realism and a sense of weight to the shot without interfering with the logic of the combat system. These effects are applied only to the visual model of the weapon, leaving the functional behavior of the shot intact.

After each shot, a recoil effect is applied, consisting of a slight displacement and rotation of the weapon. This recoil is calculated cumulatively and smoothly interpolated back to its original position, avoiding sudden movements. In addition, small random variations are introduced to break the mechanical repetition and add naturalness to the weapon's behavior.

Complementarily, the weapon incorporates an idle sway effect, which simulates the player's natural breathing and swaying when not firing. This movement is generated using sine and cosine functions, creating a smooth, cyclical displacement. The system is fully configurable and can be enabled or disabled according to design requirements.

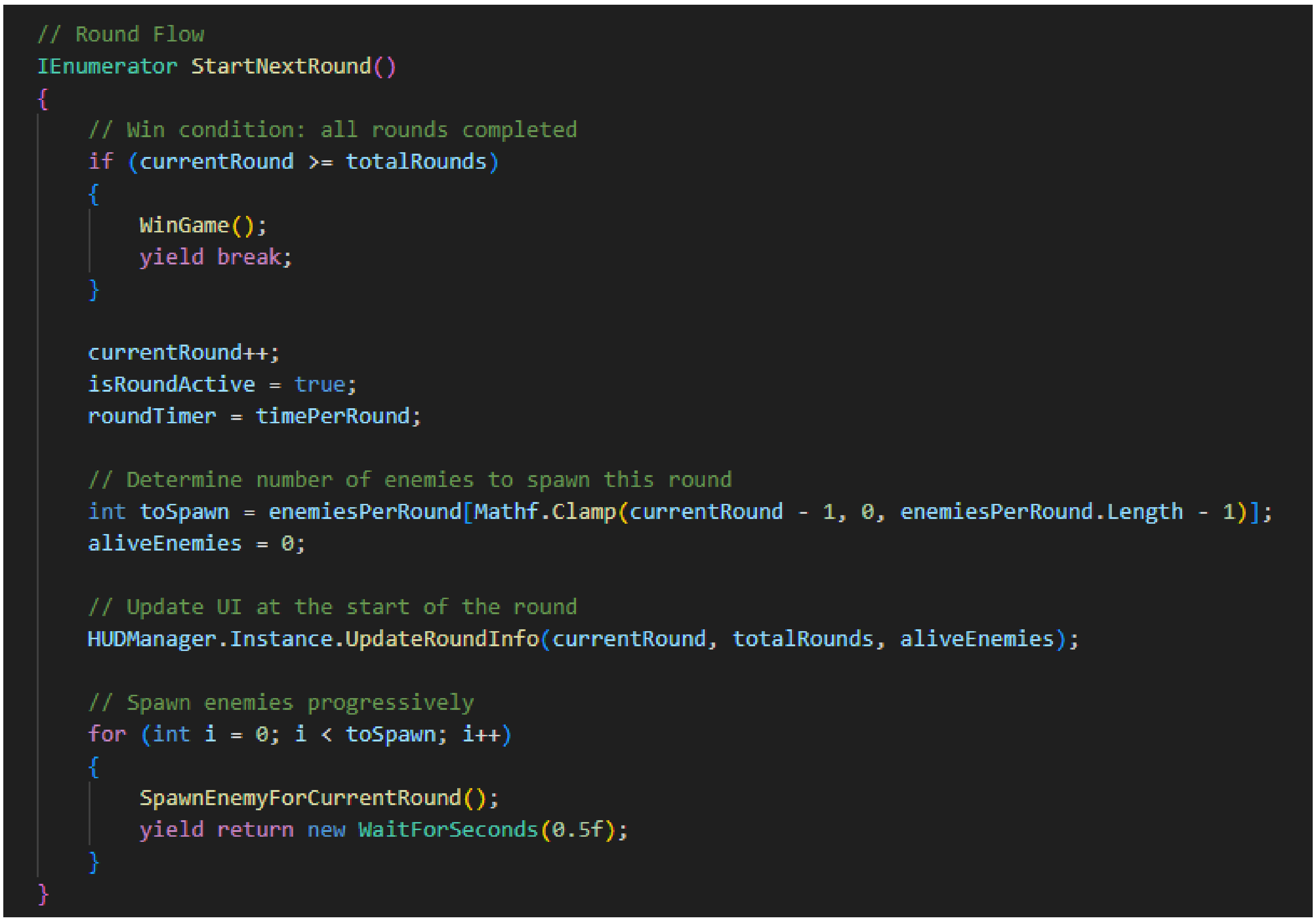
A key aspect of this implementation is that the shot calculation is always performed from the player's camera, so the recoil and idle sway effects do not affect the accuracy or consistency of the gameplay. From a technical standpoint, this approach significantly improves visual immersion, maintains precise control over weapon behavior, and avoids the need for predefined animations or an ‘Animator’ system.

Scripting

Among all the scripts in the project, ‘RoundManager.cs’ has been selected because it is responsible for coordinating the overall flow of the game. Unlike other more specific scripts, this one does not control a single object, but rather manages the general state of the game, including the round system, the appearance of enemies, the progression of difficulty, and the synchronization between the different systems.

This script clearly reflects the level of programming achieved in the project, as it combines the use of data structures, state control, dynamic object instantiation, and communication between scripts to maintain consistent game behavior.

The main function of ‘RoundManager’ is to control the shooter's round system. It is responsible for generating enemies in random positions, keeping track of active enemies in each round, and detecting when the necessary conditions have been met to end the round. Once a round is complete, the script starts with the next one, progressively increasing the difficulty and ensuring a constant evolution of the challenge. In this way, the progress of the game depends exclusively on the rules and conditions defined by this central system.



```
// Round Flow
IEnumerator StartNextRound()
{
    // Win condition: all rounds completed
    if (currentRound >= totalRounds)
    {
        WinGame();
        yield break;
    }

    currentRound++;
    isRoundActive = true;
    roundTimer = timePerRound;

    // Determine number of enemies to spawn this round
    int toSpawn = enemiesPerRound[Mathf.Clamp(currentRound - 1, 0, enemiesPerRound.Length - 1)];
    aliveEnemies = 0;

    // Update UI at the start of the round
    HUDManager.Instance.UpdateRoundInfo(currentRound, totalRounds, aliveEnemies);

    // Spawn enemies progressively
    for (int i = 0; i < toSpawn; i++)
    {
        SpawnEnemyForCurrentRound();
        yield return new WaitForSeconds(0.5f);
    }
}
```

This system uses an array (enemiesPerRound) to define the number of enemies in each round, allowing the difficulty to be scaled in a controlled manner. The use of Mathf.Clamp ensures that the system is robust even if the number of rounds exceeds the length of the array.

Enemies are generated progressively using a delayed loop (WaitForSeconds), avoiding load spikes and improving the pace of the game. In addition, the round status is synchronized with the interface using the HUDManager, keeping the player always informed.

Overall, this snippet demonstrates the use of coroutines, state management, arrays, flow control, and communication between systems, reflecting a level of programming that goes beyond the basic exercises performed in the labs.

Scene design and visual enhancement

The visual design of the scene focused on enhancing player immersion without compromising gameplay clarity.

Textures were used to clearly differentiate between interior and exterior spaces. Inside the house, wood textures were applied to floors and walls, creating a warmer and more recognizable feel for the player, while outside, brick materials and rougher surfaces were used to convey structural solidity. This visual differentiation helps the player to quickly orientate themselves within the environment.

The materials were configured in a simple but effective way, prioritizing readability over extreme realism. In some cases, normal maps were used to add visual depth without increasing the number of polygons, thus maintaining good performance during combat.

Lighting was configured using the Universal Render Pipeline (URP). An HDRI was used as a skybox to provide consistent and realistic ambient lighting, combined with a directional light that simulates sunlight. In addition, a Global Volume with post-processing was added, including ACES tone mapping, color adjustments, subtle bloom and ambient occlusion, allowing for a more cohesive and professional visual appearance without compromising player visibility.

Sound was conceived as an element to reinforce player feedback, including weapon firing sounds, impact effects, and ambient audio. Although the focus of the project is visual and mechanical, the structure of the system allows for the easy integration of sound effects associated with shots, impacts, and player actions, improving the sense of response and action.

Particles play a key role in the visual communication of gameplay. Particle systems were implemented for shooting impacts, differentiating between impacts on enemies and on the environment. These effects are instantiated at the exact point of impact detected by raycasting and are automatically destroyed after their duration, providing immediate and accurate visual feedback.

The project includes a Main Menu scene that acts as the entry point to the game (Play, Instructions and Quit), and a Pause Menu accessible during gameplay (ESC) that freezes game time and unlocks the cursor to ensure consistent usability in both the Editor and the standalone build.

Finally, the overall image and atmosphere of the scene were designed to be functional and consistent with the genre. Priority was given to a clear and legible atmosphere, avoiding visual excesses that could interfere with gameplay. The result is a balanced scene, where visual elements reinforce immersion without distracting the player from the main action.

Highlights

One of the main achievements is the scalable patrol system, managed by a centralized RoundManager. This system dynamically controls enemy generation, the number of active enemies and difficulty progression, introducing game state management that was not directly addressed in the practical sessions.

Another notable element is the enemies' artificial intelligence, based on NavMeshAgent. Enemies use automatic navigation to pursue the player, combined with distance checks and attack timing. This approach provides stable and credible behavior without relying on physical forces, surpassing the basic examples seen in class.

The project also includes a recoil system and procedural weapon animation implemented entirely by code, without using Animator or predefined animations. Through interpolations and mathematical functions, the weapon reacts to firing and simulates breathing (idle sway), maintaining shooting accuracy by separating visual logic from gameplay logic.

In addition to round progression, the project implements basic game state control through UI and scene flow: a Main Menu scene for navigation and a Pause Menu overlay that sets Time.timeScale to 0, disables player input scripts and manages cursor lock/visibility, ensuring the gameplay systems remain stable when paused.

Finally, a projectile firing system (bullets with Rigidbody) is used, with collision impacts and direction calculated from the actual position of the crosshairs on the screen to ensure consistency between the Editor and the build. This decision demonstrates an efficient use of physics in first-person shooters, ensuring accuracy, better performance, and total control over hits, damage, and visual effects.

Summary

Overall, one of the aspects that worked best was the clear separation between gameplay logic and visual elements, which allowed for stable and predictable behavior during gameplay. The use of collision-based projectiles for firing and NavMesh navigation for enemies has been efficient in terms of both accuracy and performance.

However, several limitations were identified during development. The collider configuration presented some initial problems, especially with doors and complex architectural elements, requiring manual adjustments to prevent player blockages. Likewise, although artificial intelligence fulfils its basic function of pursuit and attack, its behavior is relatively simple and could benefit from additional states or greater variation in decisions.

Another limitation of the project is the lack of enemy variety. All enemies share similar behavior, which can reduce the depth of gameplay in extended sessions. In addition, some visual decisions prioritized performance over graphical detail, meaning that certain materials and environmental elements could be further refined without excessively compromising stability.

In terms of technical decisions, simple and controllable systems were chosen over more complex solutions. This choice allowed the project to be completed within the available time frame, ensuring a solid gaming experience while leaving room for future improvements in both visual and mechanical aspects.

The final build of the game was tested as a standalone executable to ensure consistent behavior outside the Unity Editor. Attention was given to shooting accuracy, enemy navigation, and collision behavior. Minor issues related to colliders and player navigation were identified and resolved during this testing phase.

Future Work

The project establishes a solid foundation that could be expanded in multiple directions. A logical evolution of the game would be the incorporation of new weapons, each with different damage values, rate of fire, and recoil behavior, reusing the recoil and firing system already implemented.

As for enemies, new types could be introduced with movement animations, as well as faster enemies, ranged units, or enemies with greater resistance. This would allow for a wider range of difficulty in the round system and increase the depth of gameplay through more varied progression.

Visually, the environment could benefit from greater variety, including more decorative elements, dynamic indoor lighting, and improvements in materials through roughness and detail maps. Likewise, the full integration of ambient sound and sound effects would help to reinforce the atmosphere and player immersion.

Finally, additional systems such as a more advanced interface, visual damage feedback, and difficulty settings could be added, expanding the experience without altering the main structure of the project. Some things were left half-finished, but with more time for development, they could be implemented.

Appendices

APPENDIX A – Game Design Document (GDD)

GDD

APPENDIX B – Source code

Neighborhood_Defense

APPENDIX C – Executable

Neighborhood_Defense_build

APPENDIX D – Video

VIDEO

APPENDIX E – Scripts

SCRIPTS

APPENDIX F – Assets references

The following third-party assets were used during the development of Neighborhood Defense. All assets were obtained from the Unity Asset Store and are used for educational purposes only.

Environmental Assets

Chainlink Fences

Author: Kobra Game Studios

Source: Unity Asset Store

Usage: Environmental decoration (fences surrounding exterior areas)

Low Polygon House

Source: Unity Asset Store

Usage: Background residential buildings

Village Buildings

Source: Unity Asset Store

Usage: Environment and background structures

Desert Buildings

Source: Unity Asset Store

Usage: Environmental props

Old Building NYC #3

Source: Unity Asset Store

Usage: Urban background building

Urban Building

Source: Unity Asset Store

Usage: Exterior environment

Props & Objects

Furniture Mega Pack – Free

Source: Unity Asset Store

Usage: Interior furniture for the house

Food Props

Source: Unity Asset Store

Usage: Decorative interior objects

Water Fountain Round Four-Tier

Source: Unity Asset Store

Usage: Decorative outdoor prop

Weapons

FPS AKM – Model & Textures

Source: Unity Asset Store

Usage: Player weapon model

Low Poly Pistol Weapon Pack 3

Source: Unity Asset Store

Usage: Additional weapon models

Low Poly Weapons Vol.1

Source: Unity Asset Store

Usage: Weapon variations

FREE Sci-Fi Shotgun

Source: Unity Asset Store

Usage: Weapon model

Textures & Terrain

Plank Textures PBR

Source: Unity Asset Store

Usage: Interior floor and wall materials

P3D: Outdoor Wall Tile Texture Pack

Source: Unity Asset Store

Usage: Exterior wall materials

Realistic Terrain Textures FREE

Source: Unity Asset Store

Usage: Terrain texturing

Terrain Sample Asset Pack

Source: Unity Asset Store

Usage: Terrain and ground details

Grass Flowers Pack Free

Source: Unity Asset Store

Usage: Vegetation and environmental detail

Mountain Terrain, Rocks and Trees

Source: Unity Asset Store

Usage: Background environment

Controllers & Systems

Mini First Person Controller

Source: Unity Asset Store

Usage: Base first-person movement system (modified)

APPENDIX G – Report references

Unity Technologies (2025)

Unity Documentation.

Available at:

https://docs.unity3d.com

(Accessed: November 2025)

Unity Technologies (2025)

NavMeshAgent – Unity Manual.

Available at:

https://docs.unity3d.com/Manual/nav-AgentNavigation.html

(Accessed: November 2025)

Camera.ScreenPointToRay – Scripting API.

Available at: https://docs.unity3d.com/ScriptReference/Camera.ScreenPointToRay.html

(Accessed: January 2026)

RectTransformUtility.WorldToScreenPoint – Scripting API.

Available at:
https://docs.unity3d.com/ScriptReference/RectTransformUtility.WorldToScreenPoint.html

(Accessed: January 2026)

Rigidbody.velocity – Scripting API.

Available at:

https://docs.unity3d.com/ScriptReference/Rigidbody-velocity.html

(Accessed: January 2026)

MonoBehaviour.OnCollisionEnter – Scripting API.

Available at:

https://docs.unity3d.com/ScriptReference/MonoBehaviour.OnCollisionEnter.html

(Accessed: January 2026)

Pixabay (2025)

Pixabay Free Sound Effects and Audio.

Available at:

https://pixabay.com/sound-effects/

(Accessed: January 2026)

Generative AI tools were used selectively during the development of this project as auxiliary support resources. Their use was limited to assisting with conceptual understanding, problem-solving, and visual ideation, rather than producing final implementation content.

OpenAI – ChatGPT

ChatGPT was used primarily as a learning and clarification tool to:

- Support the understanding of C# scripting concepts relevant to Unity.
- Explore alternative approaches to gameplay logic, such as round progression and enemy management.
- Improve the structure and clarity of technical explanations within the report.

All scripts included in the project were written, tested, and integrated manually by the author, with AI-generated suggestions adapted or rewritten to meet the specific requirements of the game.

Examples:

“How can a round-based enemy system be structured in Unity using C#?”

“What is a clean way to track enemies alive in a wave-based shooter?”

Google – Gemini

Google Gemini was used for visual ideation purposes only, specifically to:

- Generate the game icon.
- Generate the cover image for the project report.
- These images were used as visual identifiers and did not affect gameplay, mechanics, or technical systems.

Examples:

“Generate a game icon for a first-person shooter set in a suburban environment.”

“Create a minimal cover image for a university game development report.”