Millport Camp: Developing an Open-World Zombie Shooter Game with Unity

Junyang Lyu
[redacted]
Supervisor: Raluca Daniela Gaina
[redacted]
MSc Computer Games

***Abstract*—Post-apocalyptic zombie games are popular but often merely use the setting as a backdrop. We believe this genre has the potential to raise the public's awareness on public health emergencies by letting the players experience the crises themselves in game. Also, there is a lack of demonstration projects that dissect the essential components of open-world zombie games and showcase the development process. In this project, we reviewed the relevant techniques to create an open-world zombie shooter game and developed *Millport Camp* using the Unity engine. We followed the classic steps to develop the game: requirements analysis, design, implementation, and testing. We applied the AGE (Actions, Gameplay, and Experience) framework during requirements analysis, identified key gameplay features, and utilized the Entity-Component (EC) architecture and various game patterns to build these features. We also developed a simple 'minimalist' hierarchical finite state machine (HFSM) for AI behaviors, and a level generator that uses Perlin noise maps and an offset grid to procedurally generate the game level. We conducted a playtest session on the final build of the game, which confirmed that the game includes all designed features and runs smoothly with satisfactory performance. Overall, player feedback was positive, while also showing opportunities for improvement in areas such as playstyle variety and story progression.**

***Index Terms*—game development, Unity, Entity-Component (EC) architecture, hierarchical finite state machine (HFSM), procedural level generation**

I. Introduction

Games bring joy to people, and also spread thoughts. According to a report from IGN, there are nearly 3.1 billion video game players worldwide as of mid-2020, which is about 40% of our planet's population (Bankhurst, 2020).

Zombie games are a very popular genre because they are exciting. Some of the successful games are *Resident Evil 2* (Capcom, 2019), *Days Gone* (Bend Studio, 2019), *The Last of Us* (Naughty Dog, 2013), *DayZ* (Bohemia Interactive, 2018), and *Left for Dead 2* (Valve, 2009). Many games in this genre are open-world action games as their gameplay is built around combating zombies. However, most of these games, whether single or multiplayer with their shift of focus on either the protagonist's adventurous journey or multiplayer collaboration and social interactions, use the post-apocalyptic zombie world setting as a story background but the apocalypses themselves are not part of the gameplay, such as *Days Gone*, *Resident Evil 2* and *Project Zomboid* (The Indie Stone, 2013). The reasons for this could be multiple: story-writing and art take time and effort, and building realistic and dynamic environments that change with the story's progression pose high standard for software and hardware, etc.

We think by covering the pre-apocalypse period by letting players play through and experience the apocalypses themselves, this type of game has the potential to raise awareness about public health emergencies. And for such a game, we identify that the key components include a realistic open-world, interesting player interactions with the world, and compelling storytelling. Another driving force for this project is there is currently a lack of 'best practice' demonstration projects that dissect this type of game's key components and illustrate how to develop them in Unity with the techniques like game patterns. We think this project can also serve as a case study for aspiring game developers and computer games students.

In this project, we developed a single-player, open-world, third-person zombie shooter game with a main storyline in the Unity engine, *Millport Camp*. The story begins with the protagonist, Harley, a survivor in a zombie apocalypse, departing from the city for a camp in the north to meet up with his best friend. Due to limited time budget, we built the fundamentals except for the plots: the 3Cs (character, camera, controls), basic gameplay (building, crafting, zombies and more), and a procedurally generated open-world level. We believe these fundamentals can be a good starting point for more content and storytelling. Here, we highlight some key aspects of this project: using an Entity-Component (EC) architecture to manage game objects, utilizing various game patterns for gameplay features, a custom-implemented, simple yet complete hierarchical finite state machine (HFSM) for AI behaviors, and a procedural level generator. See Fig. 1 for a screenshot of the game. The source code is available on GitHub[1].

Again, we believe our kind of game has the potential to tell serious stories. We have examples like the fun yet educational strategy game *Plague Inc.* (Ndemic Creations, 2016), which also has a DLC *Plague Inc: The Cure* released in 2021 in collaboration with the World Health Organization (WHO) during the COVID-19 pandemic to raise public awareness

[1]https://github.com/astro2049/Millport-Camp

Fig. 1. A screenshot of *Millport Camp*.

(WHO, 2021), and open-world games like *Cyberpunk 2077* (CD PROJEKT RED, 2020), which is set in a dystopian megalopolis in the future but successfully touches on modern-day topics and sparks discussions like depression (for example, in the "Happy Together" and "The Hunt" side quests ('Happy Together', 2024)) and late stage capitalism (Feral Historian, 2023).

II. Related Work

A. Game Architectures

Game architecture refers to how the entities in a game world get managed and updated. Two popular game architectures are the Entity-Component (EC) architecture and the Entity-Component-System (ECS) architecture. Both architectures work on top of a game pattern called the Game Loop.

Game Loop and Update Method. Almost every game has a game loop running in frames at a certain pace. The interval between each frame is known as delta time. If a game runs at 60 frames per second (FPS), the game world gets updated roughly every 16ms. Within each frame, the game processes inputs, updates the game world, and renders the game world to screen.

An adaptive game loop with a target framerate of 60 looks like this, see Code 1:

```
double MS_PER_FRAME = 0.016; // 60FPS
double lastTime = getCurrentTime();

while (true)
{
  double current = getCurrentTime();
  double elapsed = current - lastTime;

  processInput();
  update(elapsed);
  render();

  sleep(start + MS_PER_FRAME -
  getCurrentTime());

  lastTime = current;
}
```

Code 1: An adaptive game loop with a target framerate of 60FPS. This loop only guarantees the game does not run too fast.

Usually, the game world maintains a list of entities or game objects, such as rockets and cars, and the game's `update()` function calls each entity's `update()` every frame. This way, the game world moves forward in time.

1) Entity-Component (EC) Architecture: A common practice in game development and a "growing trend in software design, is to use composition when possible" (Nystrom, 2014).

Also known as the component pattern, this architecture separates an entity's features into different components, each containing the relevant data and logic for that feature. This way, the entity serves only as the container for these components.

Together with Update Method, each frame, the game calls its `update()`, which in turn iterates the entities in scene and call their `update()`, which in turn iterates their components and call their `update()`. In a word, it is the components that actually run the game world on the foundation level.

Unity by default uses this pattern. According to Unity docs (Unity, 2024a), "GameObjects are the fundamental objects in Unity that represent characters, props and scenery. They do not accomplish much in themselves but they act as containers for Components, which implement the functionality."

2) Entity-Component-System (ECS) Architecture: The ECS architecture takes the EC architecture further by completely separating out the components' logic to 'systems', e.g. movement system, and only letting them keep the data. At this point, someone has linked programming with an ECS system to with a relational database (Härkönen, 2019). That being said, following the Update Method, we will be calling every system's `update()` within each frame, which is quite frequent.

Unity provides ECS support, and Unity ECS has been utilized in commercial games like *V Rising* (Stunlock Studios, 2024), demonstrating potentials for better performance and advantages such as having good adaptability to gameplay changes because of its architectural superiority over object-oriented architectures (Unity, 2024b). However, we did not use it because of its learning curve and our tight development schedule.

B. Agent Behaviors

From regular pedestrians in *Watch_Dogs* (Ubisoft, 2014) to marauders with squad coordination in *Days Gone*, NPCs with intelligent behaviors are a common element in nowadays games.

For a formal definition, any in-game entity capable of making decisions is an agent, and behaviors are the decisions the agent makes. An agent can be an individual (one NPC), a group (a squad of NPCs), or a director (Gow, 2024).

Dawe et al. (2013) summarized popular techniques used for in-game agent behavior controls:

- Finite State Machines (FSMs). FSMs are simple, intuitive, easy to visualize, flexible, and currently the most common behavior control algorithm in game AIs. An FSM breaks down an NPC's behaviors into states and connects them with transitions. Each state represents a

specific behavior, and only one state is considered 'active' at a time. Transitions are directed links connecting states that allow them to switch in-between whenever certain conditions are met. Games: *DOOM (2016)* (AI and Games, 2018).

- Behavior Trees (BTs). Behavior tree organizes an agent's behaviors into a tree structure, where leaf nodes are conditions or actions, and composite nodes combine child behaviors. Games: *Bioshock Infinite* (Irrational Games, 2013), *Tom Clancy's The Division* (Massive Entertainment, 2016) (AI and Games, 2019).
- Utility System. Games: *The Sims 4* (Maxis, 2014), *F.E.A.R* (Monolith Productions, 2005), *Alien: Isolation* (Creative Assembly, 2014) (AI and Games, 2021)
- Goal-Oriented Action Planners (GOAPs). Games: *F.E.A.R*, *Assassin's Creed: Odyssey* (Ubisoft, 2018), *Tomb Raider* (Crystal Dynamics, 2013) (AI and Games, 2020; Girard, 2021)
- Hierarchical Task Networks (HTNs). Games: *KillZone 2* (Guerrilla Games, 2009), *Horizon: Zero Dawn* (Guerrilla Games, 2017) (Beij, 2017).

Agents that use the latter two techniques can also be refer to as planning agents. There are also several variant/upgrade versions of FSM, such as Concurrent State Machines, where multiple state machines that are responsible for different subsets of behaviors run at the same time (like moving and firing); and Hierarchical FSMs, in which states can contain their own state machines, are useful when there are 'super events' that could potentially interrupt a whole state machine. A good usage case for this is the service robot example in *AI for Games* (Millington, 2019).

There are existing implementations for HFSMs in Unity, such as UnityHFSM (Inspiaaa, 2024). However, we developed our own HFSM for full control while also aiming to strike a balance between achieving the essential components (states and transitions), ensuring structural modularity and state reusability, and code intuitiveness and simplicity, ultimately delivering a "minimalist" HFSM solution.

C. Procedural Level Generation

For our game, we generate the only level procedurally, which means our generator is in charge of spawning everything in the level: a big island in the middle of the map, foliage, the zombies etc. The terrain and biome generation and foliage placement are our primary focuses: we want to generate a world map with different biomes, each having different plant types with different densities, e.g., woodland with trees and bushes, and desert with stones.

An example we can look at for terrain and biome generation is *Minecraft* (Mojang Studios, 2011). *Minecraft* uses Perlin noise map/functions together with splines to generate game worlds. For terrain generation, it first uses three 2D Perlin noise maps with different amplitudes and own configured splines to generate the terrain height on the coordinates: continentalness, erosion, and peaks & valleys; and then, using a 3D Perlin density noise map, a sqaushing factor, and a height offset, to generate the actual terrain. The biomes are generated on top of the terrain with two more extra 2D Perlin noise maps: temperature and humidity (Kniberg, 2022). The reason why *Minecraft* mainly uses Perlin noise maps is to strike the balance between randomness and continuity, and the splines are used to create more variations and dramatic effects. This way, the game is able to generate consistent terrain and biomes with interesting variations. The biome generation references the Wittaker's biomes ('Biomes', 2024), a model for classifying biomes on earth, where the biome types are determined by two factors: temperature and precipitation.

No Man's Sky (Hello Games, 2016) uses domain warping which is a variation of Perlin noise to create coastlines and rivers (Murray, 2017). For foliage placement, the game uses noise maps to place smaller objects like grass and rocks, and an offset grid to place larger objects like trees when it needs to control the distances between them. The different plants are also placed on different LODs (levels of detail), e.g., trees on LOD 2, bushes on LOD 1 and grass on LOD 0, for rendering efficiency. The buildings are also placed using the offset grid technique (McKendrick, 2017).

Some games use other techniques instead of noise maps to generate game worlds. For example, *Unexplored 2* (Ludomotion, 2022) takes a more gameplay-focused approach: it uses two rounds of Voronoi tessellation to divide the map into Voronoi cells, create adventure sites, biomes, and rivers. Different locations in the biomes are then generated from a set of templates like 'a vault in the forest' encoded with a set of grammars (AI and Games, 2023).

When it comes to evaluations, currently there is a lack of consensus on how to evaluate procedurally generated levels due to their need to satisfy multiple aesthetic and functional requirements, the highly subjective nature of these requirements, and the diversity of game genres (Withington, Cook and Tokarchuk, 2024). In real-world events like the GDMC AI Settlement Generation Challenge in Minecraft, human experts are invited to assess contestants' work based on four criteria: adaptability, functionality, evocative narrative, and visual aesthetics (Salge et al., 2022; '2024 Settlement Generation Competition', 2024).

III. Background

A. Game Analysis Frameworks

Game analysis frameworks provide systematic views to comprehend games and can facilitate game design. MDA (Mechanics, Dynamics, and Aesthetics) proposed in 2004, is the first model to offer a pragmatic way to understand games. In the context of MDA, "the mechanics give rise to dynamic system behavior, which in turn leads to particular aesthetic experiences" (Hunicke, LeBlanc, and Zubek, 2004; Dillon, 2012a).

The AGE (Actions, Gameplay, and Experience) framework, proposed by Dillon in 2012b, is inspired by MDA and similarly breaks game into three layers of abstraction: Actions (e.g., moving, building a camp), Gameplay (e.g., try to be the last one alive, "territorial acquisition"), and Experience,

as in the emotions and instincts (e.g., joy, self-identification, revenge) in the context of the 6-11 framework (Dillon, 2010). However, it is a more intuitive and easier to understand game analysis framework regarding MDA, as it gives clearer definitions of the layers and presented a simple but pragmatic way to interpret games.

There are other frameworks for game analysis such as Tracy Fullerton's FDD (Formal, Dramatic, and Dynamic elements) framework, presented in the book *Game Design Workshop* (Fullerton, 2014; Bond 2022).

AGE was created for teaching, but can also be used to design real-world games. In this project, we referenced the AGE framework to facilitate our requirements analysis.

B. Software Development Lifecycle (SDLC) Models

In this project, we adopted the incremental model as our SDLC model over its flexibility. As we would be learning things and adding gameplay features along the way, the model allows us to build the game step by step and make small adjustments and upgrades on the fly.

The incremental model, as an intuitive variation to the waterfall model, first constructs a basic implementation of the system and then slowly adds functionalities throughout multiple iterations of smaller cycles (Ganney, Pisharody, and Claridge, 2020; Alshamrani and Bahattab, 2015).

C. SOLID principles in games

"S.O.L.I.D. is a collection of 5 programming principles that allow you to build larger applications, while keeping the code maintainable, flexible and, in general, with fewer bugs" (Sagmiller, 2017). The five letters correspond to: Single Responsibility, Open-Closed, Liskov Substitution, Interface Segregation, and Dependency Inversion.

The author of the SOLID principles, Martin (2017), argues that these principles can be applied to any software that has classes, i.e., groupings of data and functions. While this is generally true, some best practices in game development, such as dependency injection, do not adopt the same strategy when trying to solve a same design problem (Sagmiller, 2017).

While SOLID is a set of well-recognized software design principles, they are still abstract guidelines and low-level design considerations are still required when it comes to actual implementation.

D. Game Patterns

Over the years in the game industry, some useful game programming patterns have been summarized from the best practices. Nystrom (2014) in his book *Game Programming Patterns*, presented 20 commonly used programming patterns to achieve good game software architecture designs—that "accommodates changes", have good performance and speed, and simple. Here, we will review some of these patterns that we used in the project (despite what we've already mentioned in Section II.A).

Flyweight. The flyweight pattern saves memory by separating shared and independent data. Unity commonly uses this pattern. For example, if we have a player character with a mesh, its MeshFilter component will only hold a reference to the mesh file.

Observer. The Observer pattern is a popular and powerful decoupling pattern that allows a piece of code "announce something interesting (using messages) without actually caring about who is interested or listening to that message" (Perez-Liebana, 2024). It avoids the need for two game objects/entities wanting to communicate to call each other directly.

Prototype. The key idea of the prototype pattern is that "an object can spawn other objects similar to itself" (Nystrom, 2014). This pattern is commonly seen in Unity, where Unity Prefabs act as a kind of prototype, and game objects can be cloned using `Instantiate()`.

E. User Interface (UI) Architecture

MVC (Model, View, and Controller) is a family of design patterns commonly used when developing UIs in software applications. The general idea behind MVC is to separate the logical portion from the data and the presentation of a software (Unity Technologies, 2024).

As this separation of concerns is a good practice, traditional MVC implementation requires the View layer to actively listen to changes in the Model. However, in Unity, with UI frameworks like Unity UI and UI Toolkit in place and naturally function as the View, listening to changes can feel cumbersome. As a result, many Unity developers adopt a variation of MVC called *MVP* (Model, View, and Presenter), where the Controller is replaced by a Presenter. The Presenter acts as a go-between for the other layers, retrieving data from the Model and formatting it for display in the View, while the View handles player inputs. This is a very intuitive architecture to program UI for Unity games (Unity Technologies, 2024).

IV. Methodology

In this chapter, we introduce how the game is designed and implemented.

A. Requirements Analysis

As a start, we analyze the essential gameplay features from our game's premise. Referencing the AGE framework, we will also identify the emotions and instincts we want to evoke in the players.

1) Gameplay: Essential Elements. Our game is set in a zombie apocalypse and is a survival game, so the essential elements will be open-world, combat, crafting, and building. In the context of the 6-11 framework, these elements should relate to: *Emotions*: fear, pride, joy, excitement, sadness. *Instincts*: survival (combat, crafting and building), self-identification (role-playing), curiosity (exploring the world), and color appreciation (appreciating the environment).

Overall gameplay. We designed that the player will have to visit three quest locations located in different biomes in sequence to complete the storyline in an apocalyptic world filled with zombies. Centered around gunplay, the player can also craft items and build structures that provide survival

utilities. There will be vehicles scattered across the map for the player to drive around to gain some extra mobility.

Main actors and Gameplay Systems. The main actors in our game (i.e., game objects/entities that can perform actions) are: *Player* (the player's in-game character), *Gun*, *Vehicle*, *Zombie*, *Combat Robot*, and *Turret*. We also designed three main gameplay systems corresponding to the essential elements: Combat System, Crafting System, and Building System.

2) Performance Requirements: Our aim is to have the game run smoothly at minimum 60 FPS on middle-end PCs. A reference for this is a Dell G15 5520 gaming laptop with a configuration of Intel Core i7-12700H (CPU), NVIDIA GeForce RTX 3060 6GB (GPU), 16GB RAM, 512GB SSD, and Windows 11 Home 64-bit (OS).

B. Design

1) Game Objects: Firstly, we list out all actors' interactions using an object interactions table (see Table A 1). We also give brief descriptions of the actions, see Table A 2. An action performed by different actors might vary slightly.

2) Coding: High level Design Considerations and Choices: For game architecture, we adopt the Entity-Component (EC) architecture. Subsequently, we set some guidelines on the component design of the actors including but not limited to SOLID principles: 1) Intuitive and simple, for the sake of code readability and maintainability. 2) Single Responsibility Principle: One component, one functionality, e.g., an input component only processes input, and a health component only handles health. 3) Open-Closed Principle: A component should be modular (closed) and scalable (open). This combined with the last principle elicits a classic software engineering idea: high cohesion and low coupling. 4) Liskov Substitution Principle: Components may have simple inheritance hierarchies, but a child class should be able to present all behaviors of its parent.

Inter-Game Object Communications. We divided communications between different game objects into three scenarios and set up a guideline: 1) Inter-referencing and direct calls, for the manager scripts. For example, if the game manager needs to tell the UI manager to turn on/off the pause menu's Canvas at some points, we declare a `uiManager` field in the game manager class and plug it in in the editor. We do this because all managers are persistent and only have one instance in the scene so it's simple. 2) Through the event system, for objects created on the fly. An example is a *Turret* will need to listen to the *Dead* events to its newly added targets, but it doesn't know who it is going to add beforehand. 3) Singletons, in scenarios where some components need to access a globally provided service. An example for this the input components of different player-controlled actors have to enable/disable specific action maps of the input action asset at times. For this, we use an input manager singleton with a static input action asset reference to provide this capability. Another option for this scenario is using service locators as our game is rather small we didn't choose it.

As for event systems, there are two sets of event systems used in the game: A custom event system which is a implementation of the observer pattern in *Game Programming Patterns* (Nystrom, 2014), and the UnityEvents.

The custom event system comprises of three core parts: an `MCEvent` class, a `SubjectComponent`, and an `IObserver` interface. Using this event system, it is the game objects that populate events from their subject components and the components that implement the `IObserver` interface that listen to them. The subject component manages an `observers` dictionary so that no listeners are bothered with events they don't care about. All events are created and destroyed on the fly. Many inter-game object communications use this system. For example, the game manager listens to *Player*'s subject on `EnteredVehicle` event and switches inputs and cameras then it's invoked. As mentioned before, UnityEvents are also used, but mostly in lower frequency scenarios.

C. Implementation

1) The 3Cs: Character, Camera, Controls: Characters. Players can control two types of game objects in the game: the player character *Player* and *Vehicles*. Both *Player* and *Vehicles*' movement are physics based.

Cameras. As an isometric view game, we set the main camera's 'Projection' to 'Orthographic' and set up two Cinemachine virtual cameras for *Player* and vehicles. The virtual cameras use 'Framing Transposer' for 'Body', and set 'Follow' to the respective game object's Transform component when the player switches control.

Controls. For inputs, we use the Unity's new input system. We define two input action assets: one for gameplay and one for UI. The gameplay input action asset contains four action maps: Movement, Combat, Build, and Vehicle. The UI input asset contains the default UI input actions and several extra inputs for the in-game menus. For gameplay inputs, we use an input manager singleton to provide the only instance of the gameplay input action asset globally. The input components on different game objects are in charge of registering (and deregistering) their callbacks and toggle the corresponding action maps on (and off) upon activation (and deactivation) when the player switches control. For UI, inputs in the 'UI' action map work with the in-game UI event system and the menu inputs are processed directly by the game manager.

2) Gameplay: Entities and Components Overview. We use the EC architecture and the main actors of the game are *Player*, *Gun*, *Vehicle*, *Zombie*, *Combat Robot*, and *Turret*. See Table A 3 to Table A 8 for the component design of these actors. Here, we only show the MonoBehaviour components.

Health System. Some game objects have health and some can deal damage to others. Health is enabled by a health component, and damages are dealt through a damage dealer component. Specifically, *Player*, *Vehicles*, *Zombies*, and *Combat Robots* have health components, while *Zombies* and *Guns* have damage dealer components.

Guns. There are eight different guns in the game. We use a ScriptableObject class `GunData` to store the properties of each

Gun, utilizing the flyweight pattern. After defining `GunData`, we are able to configure different guns in the Unity editor. Upon firing, a *Gun* shoots a Raycast towards its target and sees if it hits a game object that is on a layer that it can apply damage to.

Interaction System. we designed the system to enable *Player* to interact with other objects like entering vehicles. Each interactable game object has two essential pieces: an interactable component derived from the `InteractableComponent`, and an interaction zone bounding volume collider with an interactable collider component. When *Player* enters an item's bounding volume collider, the item sets itself as the player's current interactable item. When the player presses the interaction key, it calls the item's interactable component's `Interact()` method. In the 'entering vehicles' case, when the player presses the interaction key by a *Vehicle*'s driver's seat door, the *Vehicle* will notify the game manager that the player has entered the *Vehicle*, and the game manager will proceed to switch inputs and cameras.

Building System. We have a MonoBehaviour script `BuildableComponent` and defined the relevant functions (mainly detecting overlapping objects) within this class for any buildable structures to be integrated into Build mode.

UI. The in-game UI divides into two categories: the HUD and the menus (pause menu, inventory&crafting menu, and the world map). The HUD, pause menu, and the world map are managed centrally by the the UI manager which has the references to all necessary elements (e.g., input prompts like 'Reload' and player's ammo count text), listens to events and receives calls from the game manager, and updates the elements accordingly. The inventory&crafting menu aligns with the definition of "frontend" as a "user Interface that exists outside of the core game loop" (Gonzales, 2023) and is managed using the MVP pattern: the menu's canvas and elements serve as the view, the inventory UI manager as the presenter, and the inventory manager as the model.

Quest System. As stated before, the player needs to get to three research bases to complete the game. In order to achieve this, there is a trigger collider attached to each of the bases. When the player approaches their current quest's destination, the collider activates and notifies the quest manager to proceed to the next quest. We also implemented a quest location indicator in game.

Activation Distance. To keep a good performance with our relatively big game world (512m x 512m), we only activate actors near the player during gameplay. We assign an activation distance component to *Player* (or the *Vehicle* the player is driving) with a sphere collider to activate other actors (*Zombies*, *Combat Robots*, *Vehicles*) within this distance.

3) AIs: There are three types of AIs in the game: *Zombie*, *Combat Robot*, and *Turret*. Here, we will take *Combat Robot* as an example to explain the two main components of the in-game AIs: perception and behaviors. *Combat Robots* are deployed at quest locations and actively engage approaching personnel and *Vehicles* with their rifles.

Perception. We designed a collider-based perception system. An AI that wishes to have perception derives its own perception component from the `PerceptionComponent`. This component should contain reactions for when another collider enters or exits its perception trigger collider with a sensor component attached. For example, when a *Combat Robot* senses *Player*, a *Vehicle* (only when the player is driving it), or a *Zombie*, it switches from patrol state to combat state.

Behaviors. We implemented a custom Hierarchical Finite State Machine (HFSM) for in-game AI behaviors. The implementation corresponds to its definition, where "each state can be a complete state machine in its own right", having a hierarchical structure in nature (Millington, 2009). Considering this, we order the states into a tree. This way speaking, an HFSM state is a state as well as a state machine, which is a 'state state machine'; although semantically and strictly speaking, only branch states are state state machines and leaf states are vanilla states.

The HFSM works like this: each frame, we start from the root state, executing its actions. We then recursively traverse down the tree, executing the actions of the currently active child state at each level until a leaf state is reached.

In code, we define `HFSMState<T>`, an abstract base state class for concrete child classes to implement (see Code A 1). `HFSMState<T>` comes with seven methods, with three virtual state methods: `Enter()`, `Execute(float deltaTime)`, and `Exit()`, left for child states to implement; and four fixed state machine (or branch state) methods: `EnterBranch()`, `ExecuteBranch(float deltaTime)`, `ExitBranch()`, and `ChangeState(string name)`. The three 'branch' methods are all called recursively. `ChangeState()` allows a state to switch its active child state to another one. This transition can occur in three scenarios: from one of its own state methods, from one of its active child state's state methods, or from an external component that has a reference to the state. It's worth noting that `ChangeState()` only registers the desired child state change within a state and the actual transition will take place in the next frame. This is to ensure ordered state transitions and that a branch state maintains full control over how and when a child state change should happen.

Some solutions such as UnityHFSM (Inspiaaa, 2024) defines the transitions explicitly, but we didn't in order to keep the state machines simple.

Combat Robot is controlled by two concurrent HFSMs: 'Gun' and 'Movement'. The 'Gun' HFSM decides whether the robot should reload or pull/release the trigger. The 'Movement' HFSM controls the robot's movement (as its name suggests) with two child FSMs: 'Patrol' and a 'Combat'. The 'Patrol' FSM lets the robot patrol to random locations, waiting for a few seconds in between, and the 'Combat' FSM lets the robot decide if it should move away from a target when it gets too close when in combat. See Fig. 2 for *Combat Robot* HFSM's visualization. We also attached the HFSMs of *Zombie* and *Turret* in Fig. A 1 and Fig. A 2.

4) Procedural Level Generation: The only level in this game is procedurally generated. The generation process is

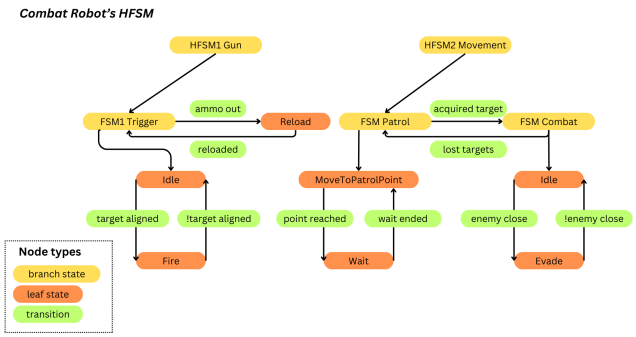

Fig. 2. Combat Robot's HFSM.

divided into two stages: first, terrain and biomes generation using Perlin noises, referencing *Minecraft*; followed by the placement of the environment objects (structures and foliage) and the actors (*Combat Robots*, *Vehicles*, *Player*, and *Zombies*), for which we utilized an offset grid technique inspired by *No Man's Sky* for some of them. The game world is a big island surround by an ocean with a total size of 512m x 512m, as shown in Fig. 3.

Terrain Generation. Just like *Minecraft*, our world is chunk/tile based. A chunk is 16m x 16m big, and the world size is 32 x 32 chunks. Although the final terrain is flat, we use a height map to determine if a chunk is *Ocean* or a land tile at the beginning. In this stage, we iterate through each chunk and use a Perlin noise map to determine its height. This height will have to then minus the corresponding value of this chunk on a falloff map. If a chunk's final height is below the designated sea level, it is an *Ocean* tile. This combination creates irregular shaped 'continents' between different generations. *Ocean* tiles will not be instantiated but instead an invisible collider cube will spawn at its location.

Biomes Generation. If a chunk is land, the generator then looks up the temperature map and the humidity map and refers to the Whittaker biomes lookup table to determine the chunk's biome type. For the biomes lookup table, we referenced Gallant's approach (2016), using a 2D array as a simplified representation of the Whittaker biomes. The temperature map linear interpolates temperature between the equator and the north pole, with a random fluctuation of ± 5 degrees applied to each chunk. The humidity map is a Perlin noise map. There are six land biome types: *Ice*, *Tundra*, *Taiga*, *Grassland*, *Woodland*, and *Desert*.

After biomes generation is complete, three research bases (which are quest locations) are placed respectively in the *Woodland*, *Taiga*, and *Ice* biomes' longitude center lines. The algorithm for this is rather straightforward: it selects the central chunk from a biome's `chunks` array and spawns the base at its center.

Foliage Placement. When it comes to placing plants, the method is to iterate through each chunk and generate its corresponding plants according to its biome's foliage configuration

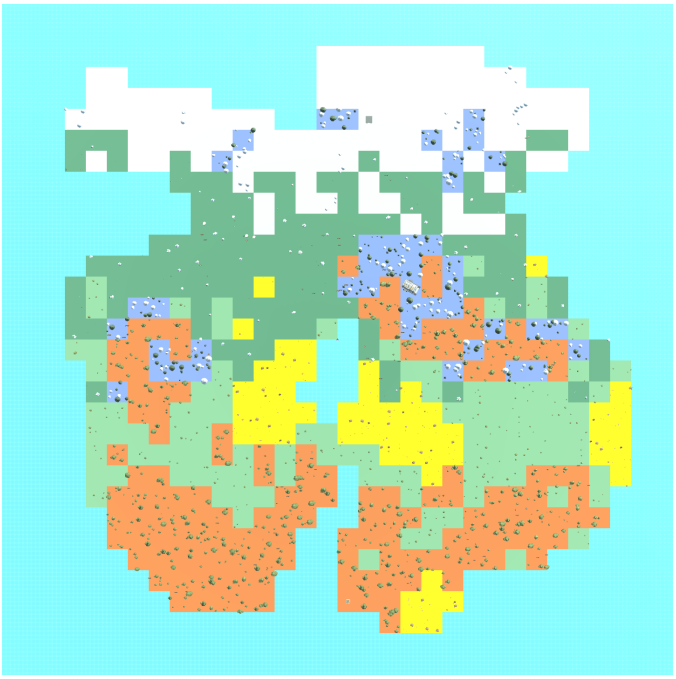

Fig. 3. A bird's eye view of the game world. The world is 512m x 512m big and have six biomes: *Ice*, *Tundra*, *Taiga*, *Grassland*, *Woodland*, and *Desert*.

(in ScriptableObject `BiomeData`). Each chunk is subdivided into a 16 x 16 offset grid, where each grid cell is 1m x 1m. The foliage configuration of a biome contains a list of configurations of different plant types (for example, trees and logs) that belong to this biome, with each of them having the Prefabs, occurrence, mesh scale range, and the number of plants of this plant type within a chunk. For each chunk, the generator reads these configurations in order and randomly selects the corresponding number of sub-grid cells to generate the plants. Using this offset grid method, we ensure that plants spawn randomly and efficiently without any overlapping.

Actors Placement. After placing the foliage, the generator constructs the NavMesh. Subsequently, the following actors are introduced into the world: *Combat Robots*, *Vehicles*, *Player*, and *Zombies*. All the actors except for *Player* are positioned on the NavMesh to avoid overlap with plants. *Combat Robots* are placed around the three research bases. Vehicles are spawned at the centers of selected chunks based on their occurrence setting. *Player* is placed at the center of one of the Woodland chunks on the right side of the map. *Zombies* are spawned in a similar manner as foliage, going chunk by chunk and picking cells in the sub-grid.

V. Evaluation

We conducted a playtest session on the final build of the game with three playtesters. Here, we present their feedback and our evaluation of the game's general appeal to players. Also, we collected some in-game data to assess how well our world generator performs.

A. AIs and Level Generator

For the AIs, in terms of behavior diversity, *Zombie*'s HFSM has 10 states and 8 transitions; *Combat Robot* has 12 states and 10 transitions; and *Turret* has 8 states and 6 transitions. We believe the AIs have diverse behaviors.

For the level generator, a typical world generation run outputs a level with an average of 600 floor tiles, 3 research bases, 1,800 plant instances (such as trees, trunks, and rocks), 36 *Combat Robots*, 25 *Vehicles*, 650 *Zombies*, 500 invisible cube colliders, and *Player*. We evaluate our world generator based on four criteria: functionality, diversity, aesthetics, and performance. We assign a score to each item on a scale of 0 (not working), 1 (working), 2 (working well), and 3 (exceptional). Functionality: The generator can stably produce a continent in the middle of the map with the six land biomes. The biomes somewhat simulate real-world systems in terms of latitude distribution and plant types. *Zombies* and *Vehicles* are spawned across the world, and *Combat Robots* appear around the research bases, meeting the gameplay requirements. There is a rare chance that a research base spawns on a separate island away from the continent due to our Perlin noise-based terrain generation approach, which could break the game. However, this issue can be mitigated with post-processing, such as connecting separated islands afterwards or improving the base placement algorithm. We give this item a score of 1; Diversity: The continent's shape, biome distribution, and foliage exhibit noticeable variations in each generation. We give this item a score of 1.5; Aesthetics: The game map and actors have a consistent low-poly art style, and the actors fit well into the environment. Visually, the game creates a borderline believable world. We give this item a score of 1; Performance: The generation process typically completes in less than two seconds, which is relatively fast. We assign this item a score of 3. Overall, we think the level generator is capable of producing levels that meet our design expectations while offering a certain degree of aesthetic appeal.

B. Playtest Results

We verified that all gameplay features functioned correctly during the playtests. Additionally, we tested the game on the PC described in Section IV.A.2 and observed a steady framerate of 165 FPS. Based on this, we conclude that the game performs well overall.

We also asked the participants in the playtests to complete a mini PXI (Player Experience Inventory) questionnaire (Haider et al., 2022) after playing, see the results in Table B 1. The feedback shows that players found the game easy to understand, with clear goals, satisfactory visuals, and functional gameplay, and most of them enjoyed it. However, the negative feedback highlighted a lack of varied playstyles, progression, and surprises. These responses suggest that expanding gameplay mechanics and adding more story content would be beneficial next steps.

VI. Conclusions and future work

In this project, we designed and developed *Millport Camp*, an open-world zombie shooter game using the Unity engine. We started by outlining the essential gameplay elements and systems, followed by identifying the main actors, their interactions, and the actions in more detail. We adopted the Entity-Component (EC) architecture and applied various game patterns to realize the gameplay features. For AIs, we developed a custom HFSM and used it to control their behaviors. The map is procedurally generated, using Perlin noises and an offset grid. Finally, we conducted a playtest on the game's final build. The results show that the game includes the planned features and performs well overall. Player feedback was positive, while also highlighting opportunities for improvement in terms of offering more playstyles and providing more story progression.

We think this project can be extended in the four following sectors:

More world-building and storytelling. Ultimately, the game needs to tell a compelling story that resonates with players, and the game world must (first) feel dynamic and real. More quest types, NPC dialogues, and collectible items can be added to increase player interaction with the game world and enhance storytelling. An AI companion might also be a good add-on to give players a sense of companionship and more motivation to explore the world.

More gameplay mechanics. The game could also benefit from introducing more mechanics and diverse playstyles to enhance its overall playability. A lot more real-world survival mechanics can be added to the game like managing food and water, injuries, crafting and using different tools, personal abilities etc., to provide players with a hardcore survival experience. A reference for these mechanics can be Project Zomboid. This way the game will also be able to teach players some really useful survival skills.

Improving the HFSM. We think the HFSM in our project can be improved by consolidating the 'transitions' concept in code, which would make state transitions more robust. Additionally, a standard of receiving and processing events can be established, either by allowing only the HFSM MonoBehaviour component to handle events and manage all state transitions within the HFSM, or by restricting only the branch states to process their own-relevant events and switch their active children states at appropriate times. There are also trade-offs between the HFSM's structural modularity and performance to consider here.

Research base generation. Instead of a single building, a research base might be expanded into a complex with multiple buildings and roads. This would introduce more gameplay possibilities and add greater depth to the world. The concept can be further extended to generate large blocks of urban areas including different facilities, roads, or even underground systems like subway stations. An interesting case to look at for this is the generative grammars used in *Unexplored 2*.

References

[1] AI and Games (2018) 'The AI of DOOM (2016) | AI and Games #30', *YouTube*, 5 August. Available at: https://www.youtube.com/watch?v=RcOdtwioEfI (Accessed: 21 June 2024).

[2] AI and Games (2019) 'Behaviour Trees: The Cornerstone of Modern Game AI | AI 101', *YouTube*, 2 January. Available at: https://www.youtube.com/watch?v=6VBCXvfNlCM (Accessed: 21 March 2024).

[3] AI and Games (2020) 'Building the AI of F.E.A.R. with Goal Oriented Action Planning | AI 101', *YouTube*, 6 May. Available at: https://www.youtube.com/watch?v=PaOLBOuyswI (Accessed: 21 March 2024).

[4] AI and Games (2021) 'How Utility AI Helps NPCs Decide What To Do Next | AI 101'. *YouTube*, 28 September. Available at: https://www.youtube.com/watch?v=p3Jbp2cZg3Q (Accessed: 21 March 2024).

[5] AI and Games (2023) 'How Unexplored 2 Generates Entire Fantasy Worlds from Scratch | Artifacts #1', *YouTube*, 26 May. Available at: https://www.youtube.com/watch?v=lL6A_MC1E2Y (Accessed: 4 August 2024).

[6] Alshamrani, A. and Bahattab, A. (2015) 'A Comparison Between Three SDLC Models: Waterfall Model, Spiral Model, and Incremental/Iterative Model', *IJCSI International Journal of Computer Science Issues*, 12(1), pp. 106-111.

[7] Bankhurst, A. (2020) 'Three Billion People Worldwide Now Play Video Games, New Report Shows', *IGN*, 15 August. Available at: https://www.ign.com/articles/three-billion-people-worldwide-now-play-video-games-new-report-shows (Accessed: 8 July 2024).

[8] Beij, A. (2017) 'The AI of Horizon Zero Dawn', *Game AI North 17*. Available at: https://www.guerrilla-games.com/media/News/Files/The-AI-of-Horizon-Zero-Dawn.pdf (Accessed: 12 July 2024).

[9] Bond, J.G. (2022) *Introduction to Game Design, Prototyping, and Development*. 3rd Edition. Addison-Wesley Professional.

[10] Dawe, M., Gargolinski, S., Dicken, L., Humphreys, T. and Mark, D. (2013) 'Behavior Selection Algorithms: An Overview'. Available at: https://www.gameaipro.com/GameAIPro/GameAIPro_Chapter04_Behavior_Selection_Algorithms.pdf (Accessed: 27 June 2024).

[11] Dillon, R. (2010) *On the Way to Fun: An Emotion-Based Approach to Successful Game Design*. Boca Raton: CRC Press.

[12] Dillon, R. (2012a) *A Modern Approach to Game Analysis and Design: The AGE Framework*. Available at: https://www.slideshare.net/slideshow/the-age-framework/14918917 (Accessed: 15 June 2024).

[13] Dillon, R. (2012b) 'Teaching Games through the A.G.E. Framework'. Available at: https://researchonline.jcu.edu.au/24660/1/24660_Dillon_2012.pdf (Accessed: 15 June 2024).

[14] Feral Historian (2023) 'Cyberpunk 2077 and "Late Stage Capitalism"', *YouTube*, 25 August. Available at: https://www.youtube.com/watch?v=c9_SNSuI5e0 (Accessed: 28 July 2024).

[15] Fullerton, T. (2014) *Game Design Workshop*. 3rd Edition. Boca Raton: CRC Press.

[16] Gallant, J. (2016) 'Procedurally Generating Wrapping World Maps in Unity C# – Part 4', *jgallant's Indie Game Developer Homepage*, 15 January. Available at: https://www.jgallant.com/procedurally-generating-wrapping-world-maps-in-unity-csharp-part-4/ (Accessed: 31 July 2024).

[17] Ganney, P.S., Pisharody, S. and Claridge, E. (2020) 'Software Engineering', in Taktak, A., Ganney, P.S., Long, D. and Axell, R.G. (eds) *Clinical Engineering: A Handbook for Clinical and Biomedical Engineers*. 2nd Edition. Academic Press, pp. 131-168.

[18] Girard, S. (2021) 'AI Action Planning on Assassin's Creed Odyssey and Immortals Fenyx Rising', *GDC Vault*. Available at: https://gdcvault.com/play/1027004/AI-Action-Planning-on-Assassin (Accessed: 12 July 2024).

[19] Gonzales, J. (2023) 'UI Engineering Patterns from "Marvel's Midnight Suns"', *GDC Vault*. Available at: https://gdcvault.com/play/1028880/UI-Engineering-Patterns-from-Marvel (Accessed: 19 May 2024).

[20] Haider, A., Harteveld, C., Johnson, D., Birk, M.V., Mandryk, R.L., Seif El-Nasr, M., Nacke, L.E., Gerling, K. and Vanden Abeele, V. (2022) 'miniPXI: Development and Validation of an Eleven-Item Measure of the Player Experience Inventory', *Proceedings of the ACM on Human-Computer Interaction*, 6, pp. 1-26. Available at: https://doi.org/10.1145/3549507 (Accessed: 20 August 2024).

[21] Härkönen, T. (2019) 'Advantages and Implementation of Entity-Component-Systems'. Available at: https://trepo.tuni.fi/bitstream/handle/123456789/27593/H%C3%A4rk%C3%B6nen.pdf (Accessed: 15 June 2024).

[22] Hunicke, R., LeBlanc, M. and Zubek, R. (2004) 'MDA: A Formal Approach to Game Design and Game Research'. Available at: https://users.cs.northwestern.edu/~hunicke/MDA.pdf (Accessed: 13 June 2024).

[23] Inspiaaa (2024) *UnityHFSM* (Version 2.1.0) [Computer program]. Available at: https://github.com/Inspiaaa/UnityHFSM (Accessed: 21 June 2024).

[24] Kniberg, H. (2022) 'Minecraft terrain generation in a nutshell', *YouTube*, 6 February. Available at: https://www.youtube.com/watch?v=CSa5O6knuwI (Accessed: 30 July 2024).

[25] Martin, R.C. (2017) *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Pearson.

[26] McKendrick, I. (2017) 'Continuous World Generation in No Man's Sky', *YouTube*, 9 August. Available at: https://www.youtube.com/watch?v=sCRzxEEcO2Y (Accessed: 4 August 2024).

[27] Millington, I. (2019) *AI for Games*. 3rd Edition. Boca Raton: CRC Press.

[28] Murray, S. (2017) 'Building Worlds in No Man's Sky Using Math(s)', *YouTube*, 17 April. Available at: https://www.youtube.com/watch?v=C9RyEiEzMiU (Accessed: 4 August 2024).

[29] Nystrom, R. (2014) *Game Programming Patterns*. Available at: https://gameprogrammingpatterns.com/contents.html (Accessed: 27 May 2024).

[30] Perez-Liebana, D. (2024) 'Messages and Event Queues'. *ECS7014P: Advanced Game Development*. Queen Mary University of London. Available at: https://qmplus.qmul.ac.uk/mod/resource/view.php?id=1907005 (Accessed: 10 April 2024).

[31] Sagmiller, D.V. (2017) 'Unite Austin 2017 - S.O.L.I.D. Unity', *YouTube*, 14 November. Available at: https://www.youtube.com/watch?v=eIf3-aDTOOA (Accessed: 5 July 2024).

[32] Salge, C., Aranha, C., Brightmoore, A., Butler, S., Canaan, R., Cook, M., Cerny Green, M., Fischer, H., Guckelsberger, C., Hadley, J., Hervé, J.-B., Johnson, M.R., Kybartas, Q., Mason, D., Preuss, M., Smith, T., Thawonmas, R. and Togelius, J. (2022) 'Impressions of the GDMC AI Settlement Generation Challenge in Minecraft', *FDG 2022: 17th International Conference on the Foundations of Digital Games*, Association for Computing Machinery, Athens, Greece, 5–8 September 2022. Available at: https://doi.org/10.1145/3555858.3555940 (Accessed: 20 August 2024).

[33] Unity (2024a) *ECS for Unity*. Available at: https://unity.com/ecs (Accessed: 6 July 2024).

[34] Unity (2024b) *Unity User Manual 2022.3 (LTS)*. Available at: https://docs.unity3d.com/Manual/ (Accessed: 15 June 2024).

[35] Unity Technologies (2024) *Build a Modular Codebase with MVC and MVP Programming Patterns*. Available at: https://learn.unity.com/tutorial/65e0cfacedbc2a2351773054 (Accessed: 1 July 2024).

[36] Withington, O., Cook, M. and Tokarchuk, L. (2024) 'On the evaluation of procedural level generation systems', *FDG 2024: Foundations of Digital Games*, Association for Computing Machinery, Worcester, MA, USA, 21–24 May 2024. Available at: https://doi.org/10.1145/3649921.3650016 (Accessed: 20 August 2024).

[37] World Health Organization (2021) *Experts and gamers join forces to fight COVID-19 and stop future disease outbreaks via Plague Inc: The Cure*. Available at: https://www.who.int/news-room/feature-stories/detail/experts-and-gamers-join-forces-to-fight-covid-19-and-stop-future-disease-outbreaks-via-plague-inc-the-cure (Accessed: 28 July 2024).

[38] '2024 Settlement Generation Competition' (2024) *Generative Design in Minecraft Competition Wiki*. Available at: https://gendesignmc.wikidot.com/wiki:2024-settlement-generation-competition (Accessed: 20 August 2024).

[39] 'Biomes' (2024) *Wikipedia*. Available at: https://en.wikipedia.org/wiki/Biome (Accessed: 30 July 2024).

[40] 'Happy Together' (2024) *Cyberpunk Wiki - Fandom*. Available at: https://cyberpunk.fandom.com/wiki/Happy_Together (Accessed: 4 July 2024).

Appendix A: *Millport Camp*

Table A 1
Game object interactions

Receiver **Initiator**	Player	Zombie	Combat Robot	Gun	Vehicle	Turret
Player	1. Move 2. Look at			3. Equip 4. Aim 5. Fire 6. Reload 7. Unequip	8. Enter 9. Drive 10. Exit	1. Move 11. Rotate 12. Place
Zombie	13. Chase 14. Melee Attack	15. Roam	13. Chase 14. Melee Attack		13. Chase 14. Melee Attack	
Combat Robot	16. Evade	2. Look at 16. Evade	15. Roam	4. Aim 5. Fire 6. Reload	2. Look at 16. Evade	
Gun	17. Damage	17. Damage	17. Damage	5. Fire 6. Reload	17. Damage	17. Damage
Vehicle		17. Damage	17. Damage		1. Move	
Turret		2. Look at	2. Look at	4. Aim 5. Fire 6. Reload		

Table A 2
Brief descriptions of actions

Action / Number, Name	**Brief Description**
1. Move	Move up(forward)/left/down(backward)/right. • Player-Player: In world coordinates, relative to camera. • Vehicle: In local coordinates, plus turning. Player-Turret: Move the Turret to mouse position.
2. Look at	Rotate horizontally to look at a position in the world. • Player: Look at mouse position in world. • Combat Robot, Turret: Look at a target.
3. Equip	Equip a weapon: gun.
4. Aim	Align the equipped gun to a point in the world.
5. Fire	• Player: Fire the equipped gun. • Gun: Shoot a Raycast bullet forward.
6. Reload	• Player: Reload the equipped gun. • Gun: Reload.
7. Unequip	Unequip the current weapon.
8. Enter	Enter a vehicle.
9. Drive	Make the vehicle 1. Move.
10. Exit	Exit a vehicle.
11. Rotate	Rotate a game object around the y axis +- 90 degrees (Build Mode).
12. Place	Place a structure (Build Mode).
13. Chase	Move towards a target. This is a continuous action.
14. Melee Attack	Strike a target once, do 17. Damage.
15. Roam	Walk around randomly.
16. Evade	Move in the opposite direction of an approaching enemy. This is a continuous action.
17. Damage	Deal some damage to a target.

Table A 3
Player's core components

Component / Number, Name	**Description**
1. Player Input	Handles the player's inputs in *Combat* and *Build* mode. Enables plyer to move, shoot etc.
2. Health	Keeps track of player's health. Player can also regenerate health.
3. Player State	Some player-specific gun and interaction related logic.
4. Player Pawn	Player-specific logic on death.
5. Player Observer	Listens to equipped gun's *Reloaded* event and sets status marker.
6. Subject	Sends messages on occasions: for example, to game manager upon entering vehicles.
7. Capsule Collider	Enables physical interactions with the world, let other AIs detect the player, and integrates with the combat system (gun raycasting).
8. Rigidbody	Works with the player input component to move the player based on physics.

Table A 4
Gun's core components

Component / Number, Name	**Description**
1. Gun State	Contains the firing and reloading logic.
2. Damage Dealer	Deals damage to another actor's health component.
3. Subject	Sends messages to holder about *MagEmpty*, *Reloaded* etc.
4. Audio Source	Makes sounds.

Table A 5
Vehicle's core components

Component / Number, Name	**Description**
1. Vehicle Input	Handles the player's input in *Combat* mode.
2. Health	Keep track of vehicle's health.
3. Damage Dealer	Deals damage to *Zombie*/*Combat Robot*'s health component.
4. Actor	Shared actor logic on death.
5. Vehicle State	Stores the driver game object's reference.
6. Vehicle	Reads the *moveInput* value in vehicle input component and moves the vehicle based on physics.
7. Vehicle Interactable	Holds the logic of what should happen when interacted with.
8. Subject	Sends messages on some occasions such as *Dead*.
9. Mesh Collider	Enables physical interactions with the world, let other AIs detect the vehicle, and integrates with the combat system (gun raycasting).
10. Rigidbody	Works with vehicle control component and the wheels.
11. Nav Mesh Obstacle	To push away *Zombies*/*Combat Robots*.
12. Activatee	Allows self to be activated/deactivated when player gets near/far.

Table A 6 (continued on next page)
Zombie's core components

Component / Number, Name	**Description**
1. Health	Keeps track of zombie's health.
2. Damage Dealer	Deals actual damages to another actor's health component.
3. NPC Pawn	NPC-specific logic on death.
4. Target Tracker	Works with AI perception component and maintains a perceived actors list.

5. Zombie State	Holds references to some other components. The HFSM component uses this component as an entrance to those components (for example, NavMesh agent).
6. Zombie HFSM	Controls zombie's behaviors.
7. Subject	Sends messages on some occasions: e.g., *Dead*.
8. Nav Mesh Agent	Enables zombie to move on the NavMesh.
9. Capsule Collider	Enables physical interactions with the world, let other AIs detect the zombie, and integrates with the combat system (gun raycasting).
10. Rigidbody	Works with the NPC perception component, enabling zombie to detect other actors.
11. AI Perception	Gives zombie a sphere-based perception and works with the target tracker component.
12. Activatee	Allows self to be activated/deactivated when player gets near/far.

Table A 7
Combat Robot's core components

Component / Number, Name	**Description**
1. Health	Keeps track of combat robot's health.
2. NPC Pawn	NPC-specific logic on death.
3. Combat Robot State	Holds references to some other components. The HFSM component uses this component as an entrance to those components (for example, gunner component).
4. Combat Robot HFSM	Controls combat robot's behaviors.
5. Subject	Sends messages on some occasions: e.g., *Dead*.
6. Nav Mesh Agent	Enables combat robot to move on the NavMesh.
7. Gunner	Encapsulates some logic gun holders have in common, e.g., pulling the trigger.
8. Target Tracker	Works with AI perception component and maintains a perceived actors list.
9. Capsule Collider	Enables physical interactions with the world, let other AIs detect the combat robot, and integrates with the combat system (gun raycasting).
10. Rigidbody	Works with the NPC perception component, enabling combat robot to detect other actors.
11. AI Perception	Gives combat robot a sphere-based perception and works with the target tracker component.
12. Activatee	Allows self to be activated/deactivated when player gets near/far.

Table A 8
Turret's core components

Component / Number, Name	**Description**
1. Turret State	Holds references to some other components. The HFSM component uses this component as an entrance to those components (for example, gunner component).
2. Turret HFSM	Controls turret's behaviors.
3. Buildable	Shared logic for buildable structures: changing material colors on spawn and detecting overlaps with other objects etc.
4. Subject	Sends messages to the UI manager when it's (no longer) overlapping with another game object in *Build* mode.
5. Gunner	Encapsulates some logic gun holders have in common, e.g., firing.
6. Target Tracker	Works with AI perception component and maintains a perceived actors list.
7. AI Perception	Gives turret a sphere-based perception and works with the target tracker component.
8. Activatee	Allows self to be activated/deactivated when player gets near/far.

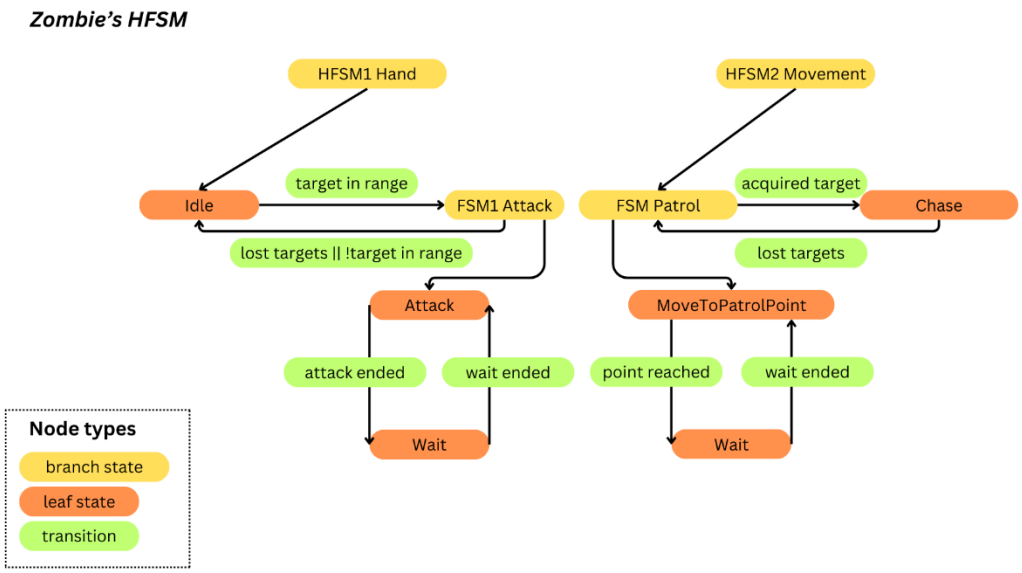

Fig. A 1: Zombie's HFSM.

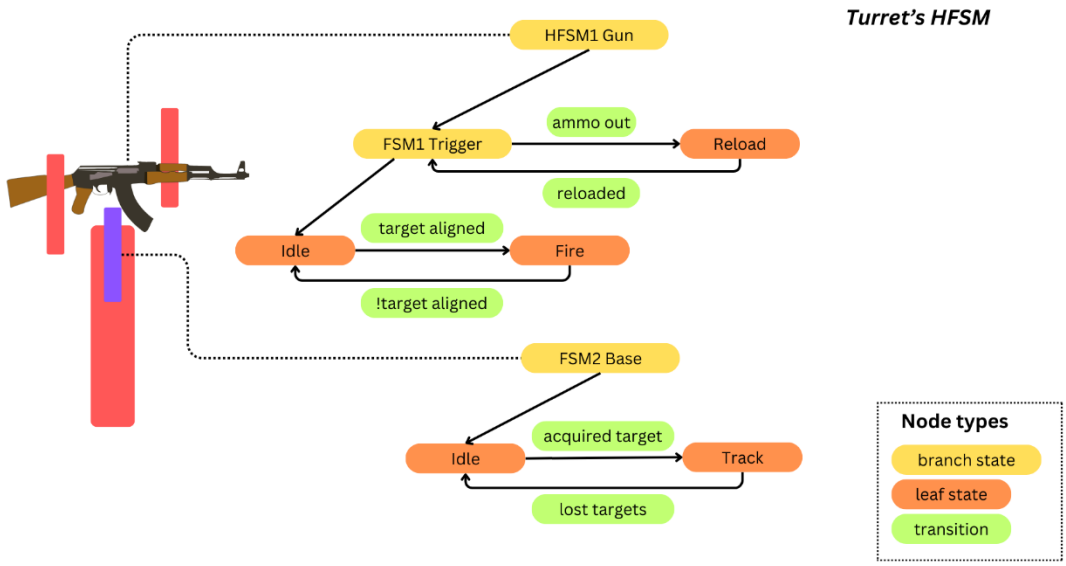

Fig. A 2: Turret's HFSM.

Code A 1
HFSM state class: HFSMState<T>

```
public abstract class HFSMState<T> where T : StateComponent
{
    /*
     * 0. Common fields
     */
    public string name; // Name of the state, e.g. "Idle", "Fire", "Gun"
    private bool isBranch = false;
    protected readonly T owner; // Owner game object's state component, manages
context
    protected readonly HFSMState<T> parentState; // Parent branch state i.e.
state machine

    /*
     * 1. State Machine fields
     */
    public HFSMState<T> current;
    private readonly Dictionary<string, HFSMState<T>> subStates = new
Dictionary<string, HFSMState<T>>();

    private bool requestedToChangeState = false;
    private HFSMState<T> next;

    protected void AddSubStates(params HFSMState<T>[] states)
    {
        foreach (HFSMState<T> state in states) {
            subStates.Add(state.name, state);
        }
        if (subStates.Count > 0) {
            isBranch = true;
        }
    }

    protected HFSMState(T owner, HFSMState<T> parentState)
    {
        this.owner = owner;
        this.parentState = parentState;
    }

    /*
     * 2. State methods
     * These 3 methods are left for derived classes to implement.
     */
    protected virtual void Enter() { }
    protected virtual void Execute(float deltaTime) { }
    protected virtual void Exit() { }

    /*
     * 3. State Machine methods
     * These 4 methods are fixed.
     */
    private void EnterBranch()
    {
        Enter();
        if (isBranch) {
            current.EnterBranch();
        }
    }

    public void ExecuteBranch(float deltaTime)
    {
```

Code A 1 (continued)

```
        Execute(deltaTime);

        // Potential change of active child state
        if (requestedToChangeState) {
            requestedToChangeState = false;
            current.ExitBranch();
            current = next;
            current.EnterBranch();
        }

        if (isBranch) {
            current.ExecuteBranch(deltaTime);
        }
    }

    private void ExitBranch()
    {
        if (isBranch) {
            current.ExitBranch();
        }
        Exit();
    }

    public void ChangeState(string name)
    {
        requestedToChangeState = true;
        next = subStates[name];
    }

    /*
     * 4. Debug methods
     */
    public string GetActiveBranch(List<string> stateNames)
    {
        stateNames.Add(name);

        if (!isBranch) {
            return string.Join(" > ", stateNames);
        }
        return current.GetActiveBranch(stateNames);
    }
}
```

Appendix B: Playtest Questionnaires

Table B 1
Mini PXI questionnaires (Haider et al., 2022) filled out by playtesters in the final playtest

Playtester 1:

	Constructs	**Items**	***Strongly disagree***	***Disagree***	***Slightly disagree***	***Neither disagree, neither agree***	***Slightly agree***	***Agree***	***Strongly agree***
			-3	***-2***	***-1***	***0***	***1***	***2***	***3***
Psychosocial Consequences	Meaning	Playing the game was meaningful to me.				✓			
	Mastery	I felt I was good at playing this game.							✓
	Immersion	I was fully focused on the game.							✓
	Autonomy	I felt free to play the game in my own way.			✓				
	Curiosity	I wanted to explore how the game evolved.				✓			
Functional Consequences	Ease of Control	It was easy to know how to perform actions in the game.							✓
	Challenge	The game was not too easy and not too hard to play.							✓
	Progress Feedback	The game gave clear feedback on my progress towards the goals.							✓
	Audiovisual Appeal	I liked the look and feel of the game.						✓	
	Clarity of Goals	The goals of the game were clear to me.							✓
Enjoyment		I had a good time playing this game.						✓	

Playtester 2:

	Constructs	**Items**	***Strongly disagree***	***Disagree***	***Slightly disagree***	***Neither disagree, neither agree***	***Slightly agree***	***Agree***	***Strongly agree***
			-3	***-2***	***-1***	***0***	***1***	***2***	***3***
Psychosocial Consequences	Meaning	Playing the game was meaningful to me.						✓	
	Mastery	I felt I was good at playing this game.			✓				
	Immersion	I was fully focused on the game.							✓
	Autonomy	I felt free to play the game in my own way.						✓	
	Curiosity	I wanted to explore how the game evolved.						✓	

Table B 1 (continued)

Functional Consequences	Ease of Control	It was easy to know how to perform actions in the game.						✓	
	Challenge	The game was not too easy and not too hard to play.			✓				
	Progress Feedback	The game gave clear feedback on my progress towards the goals.						✓	
	Audiovisual Appeal	I liked the look and feel of the game.						✓	
	Clarity of Goals	The goals of the game were clear to me.							✓
Enjoyment		I had a good time playing this game.							✓

Playtester 3:

	Constructs	**Items**	***Strongly disagree***	***Disagree***	***Slightly disagree***	***Neither disagree, neither agree***	***Slightly agree***	***Agree***	***Strongly agree***
			-3	***-2***	***-1***	***0***	***1***	***2***	***3***
Psychosocial Consequences	Meaning	Playing the game was meaningful to me.						✓	
	Mastery	I felt I was good at playing this game.				✓			
	Immersion	I was fully focused on the game.						✓	
	Autonomy	I felt free to play the game in my own way.			✓				
	Curiosity	I wanted to explore how the game evolved.				✓			
Functional Consequences	Ease of Control	It was easy to know how to perform actions in the game.							✓
	Challenge	The game was not too easy and not too hard to play.				✓			
	Progress Feedback	The game gave clear feedback on my progress towards the goals.							✓
	Audiovisual Appeal	I liked the look and feel of the game.							✓
	Clarity of Goals	The goals of the game were clear to me.							✓
Enjoyment		I had a good time playing this game.					✓		