Kingston University London

CI7825 Programming Report

Prajwal Shetty Vijaykumar K2371155
Asheen Mathasing K2358719

Introduction to Game Implementation

This implementation focuses on optimizing network performance, enhancing player interactions, and ensuring seamless gameplay continuity, which are crucial for the fast-paced nature of our game. Our ambition was to sculpt a gameplay experience that challenges players' strategic thinking in a friendly, fast paced environment.

The game itself is a team-based (2v2), top-down multiplayer game in which players choose from one of three refrigerator types - Mini, Single Door, and Double Door - each equipped with distinct capabilities, movement speeds, and health parameters. In order to have this experience to meet our expectations as developers we needed to have the player be able to:

1. Log in using an existing username / account
2. Create a lobby with a custom name
3. Have their lobby be joinable
4. Choose from a selection of characters
5. Choose their team
6. Start the game
7. View their score and names in game (and have this replicate to their team)

To implement our multiplayer game we used Unreal Engine 5.3.2 and alongside this, the Epic Online Services SDK. In order to meet our goals, our aim was to implement the following systems: Resource management, combat mechanics, in-game replicated UI, login, create / join lobby.

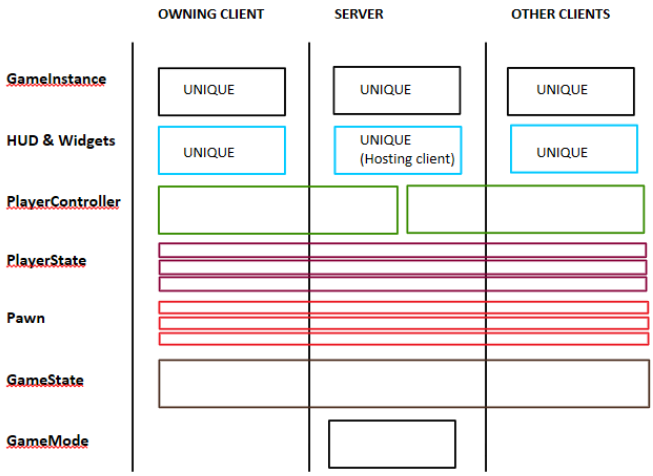
Unfortunately, in terms of polish, we simply did not have the “art assets” to implement and so we worked with what we had and aimed to have the code set up so that if there was more time, the project could continue with a strong foundation. We will discuss more in depth these systems and how we have met some expectations and the challenges involved when we were unable to meet others.

Game Implementation Features and Practices

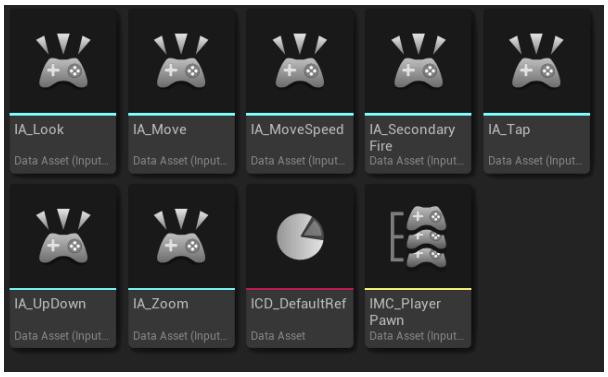
The development of Refriger-hater involved several advanced Unreal Engine features and best practices:

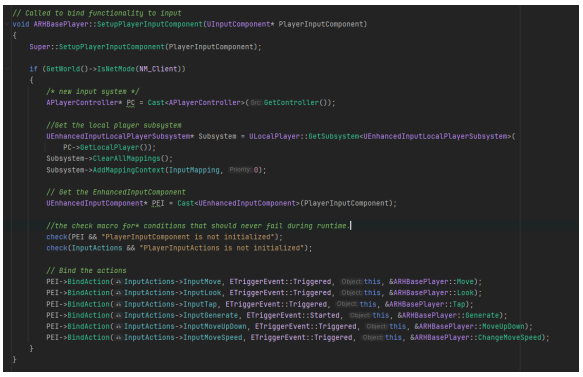
Multiplayer Game Architecture: Utilizing Unreal’s multiplayer framework, the game implements custom player states, game states, and network replication to ensure smooth and

synchronized gameplay across different client sessions. In Unreal Engine, multiplayer architecture is managed through distinct classes: GameInstance provides persistent data storage across sessions uniquely for each player; GameMode defines game rules and exists only server-side to centralize authoritative control; GameState synchronizes essential game data like scores across all clients; PlayerState replicates individual player information to all clients; and PlayerController manages input and actions locally for each player while also existing on the server for authority verification. Together, these elements ensure efficient data flow and secure gameplay management in multiplayer environments.



Unreal's New Input System: We explored and implemented Unreal's new system for the project, this is particularly important for making our game to adopt any kind of future input devices like gamepad, touch etc, by simply remapping the current input context. The C++ implementation or the same was also straight forward.





```
// Called to bind functionality to input
void ARHBasePlayer::SetupPlayerInputComponent(UInputComponent* PlayerInputComponent)
{
    Super::SetupPlayerInputComponent(PlayerInputComponent);

    if (GetWorld()->IsNetMode(NM_Client))
    {
        /* new input system */
        APlayerController* PC = Cast<APlayerController>(GetController());

        //Get the local player subsystem
        UEnhancedInputLocalPlayerSubsystem* Subsystem = ULocalPlayer::GetSubsystem<UEnhancedInputLocalPlayerSubsystem>(
            PC->GetLocalPlayer());
        Subsystem->ClearAllMappings();
        Subsystem->AddMappingContext(InputMapping, Priority: 0);

        // Get the EnhancedInputComponent
        UEnhancedInputComponent* PEI = Cast<UEnhancedInputComponent>(PlayerInputComponent);

        //the check macro for* conditions that should never fail during runtime
        check(PEI && "PlayerInputComponent is not initialized");
        check(InputActions && "PlayerInputActions is not initialized");

        // Bind the actions
        PEI->BindAction(InputActions->InputMove, ETriggerEvent::Triggered, Object: this, &ARHBasePlayer::Move);
        PEI->BindAction(InputActions->InputLook, ETriggerEvent::Triggered, Object: this, &ARHBasePlayer::Look);
        PEI->BindAction(InputActions->InputTap, ETriggerEvent::Triggered, Object: this, &ARHBasePlayer::Tap);
        PEI->BindAction(InputActions->InputGenerate, ETriggerEvent::Started, Object: this, &ARHBasePlayer::Generate);
        PEI->BindAction(InputActions->InputMoveUpDown, ETriggerEvent::Triggered, Object: this, &ARHBasePlayer::MoveUpDown);
        PEI->BindAction(InputActions->InputMoveSpeed, ETriggerEvent::Triggered, Object: this, &ARHBasePlayer::ChangeMoveSpeed);
    }
}
```

RPCs and Network Ownership: Remote Procedure Calls (RPCs) and ownership models are essential for managing network communications effectively. Unreal RPCs are specialized functions that, although called locally, are executed on another machine—either a server or client. In Unreal, Ownership determines the execution context for RPCs, meaning that only the owning client or server can execute certain RPCs. In practical terms, ownership is defined by the PlayerController associated with a network connection. For example, a Pawn possessed by a PlayerController is considered to be owned by that PlayerController's connection. This

ownership is crucial during gameplay; it ensures that only the owner can execute certain actions on the Pawn, enhancing security and gameplay integrity.

The table below from Epic shows how RPCs behave with respect to the device and ownership of the actor's instance from which it is invoked from.

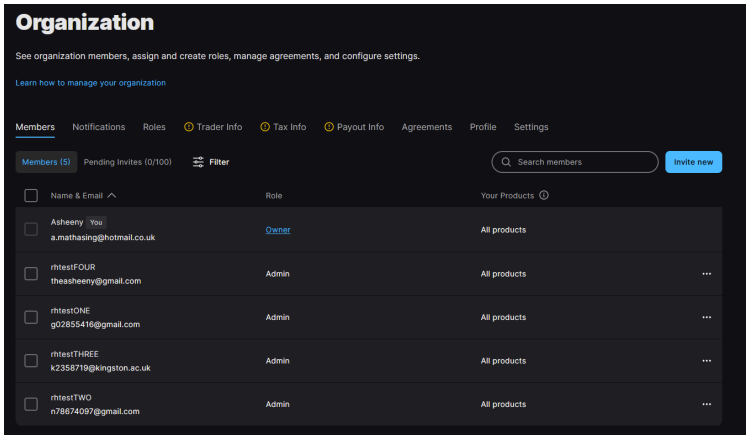
RPC invoked from the server

Actor ownership	Not replicated	NetMulticast	Server	Client
Client-owned actor	Runs on server	Runs on server and all clients	Runs on server	Runs on actor's owning client
Server-owned actor	Runs on server	Runs on server and all clients	Runs on server	Runs on server
Unowned actor	Runs on server	Runs on server and all clients	Runs on server	Runs on server

RPC invoked from a client

Actor ownership	Not replicated	NetMulticast	Server	Client
Owned by invoking client	Runs on invoking client	Runs on invoking client	Runs on server	Runs on invoking client
Owned by a different client	Runs on invoking client	Runs on invoking client	Dropped	Runs on invoking client
Server-owned actor	Runs on invoking client	Runs on invoking client	Dropped	Runs on invoking client
Unowned actor	Runs on invoking client	Runs on invoking client	Dropped	Runs on invoking client

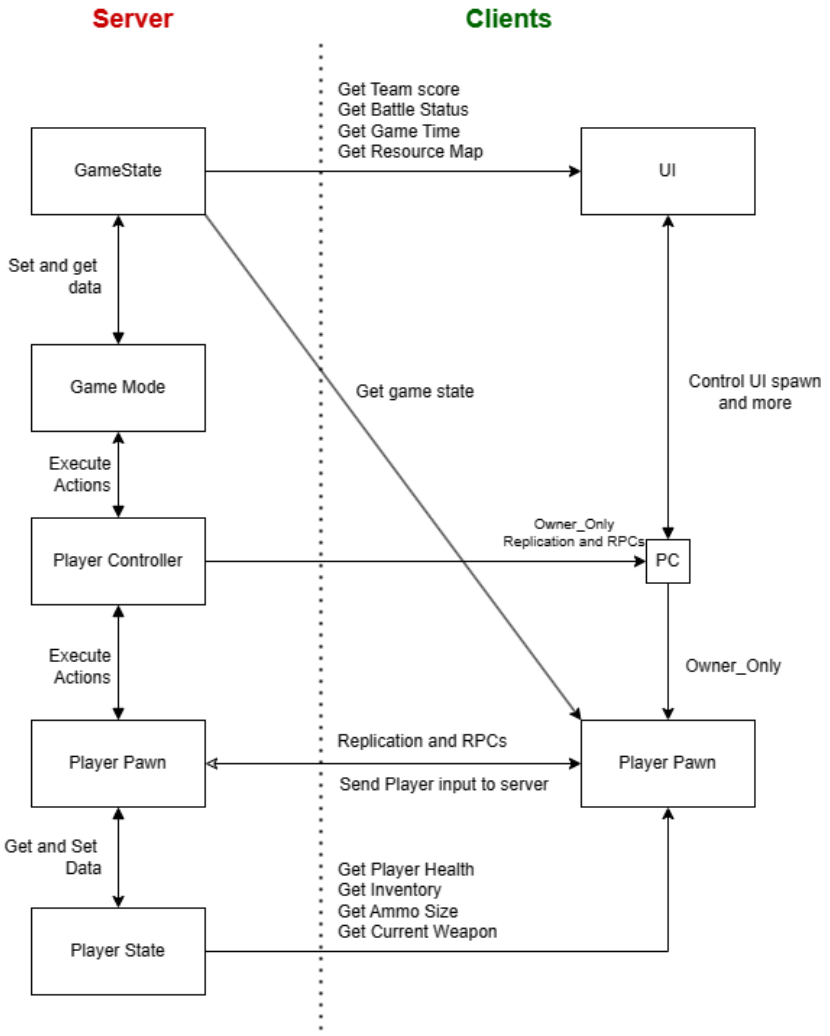
Epic's Online Services: We utilized Epic's Online Services (EOS), which supports a range of cross-platform functionalities including authentication, matchmaking, and player progression. Although integrating EOS midway through the project led to some initial technical challenges, its comprehensive features—such as account, game, and store services—greatly enhanced the scalability and maintenance of multiplayer games. If we had integrated EOS from the onset, we could have avoided these issues. Accessing EOS through its SDK, as seen in uses by Fortnite and the Epic Games Store, proved beneficial for realistic multiplayer game expectations.



Analysis of Our Implementation:

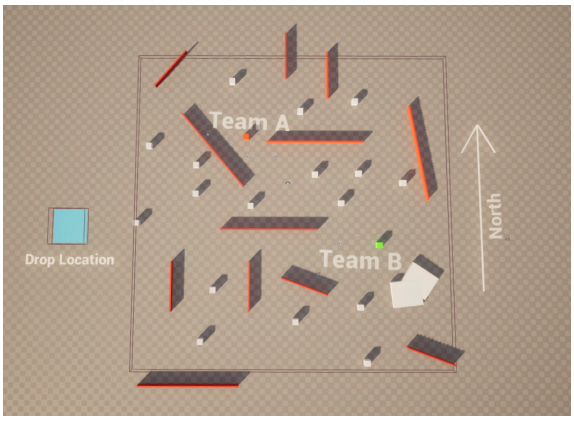
Multiplayer Architecture and Security:

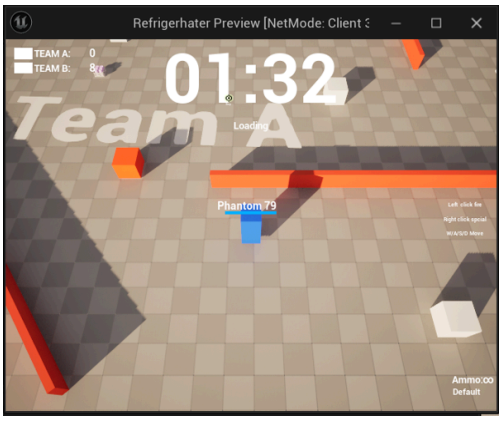
In the architectural design of Refriger-hater, we meticulously structured our gameplay logic and data management to support a robust multiplayer environment. With the Multiplayer context in mind we considered writing almost all of the game's core logic and physics calculations on the server side. Only exception here was the player's movement logic as that involves larger effort into making it server authoritative, especially lag free and smooth. The below image shows how control is distributed among various unreal actors and how only the actor instances in the server have control over the logic and game execution.



The main game mode of the battle level called `RHGameModeBase` is responsible for handling critical game functions such as managing player spawns and deaths, allocating team points, and other game-critical events. This particular class only exists on the server side, where it can

maintain authoritative control over game rules and player interactions, ensuring that gameplay decisions are securely managed and consistent across all clients.





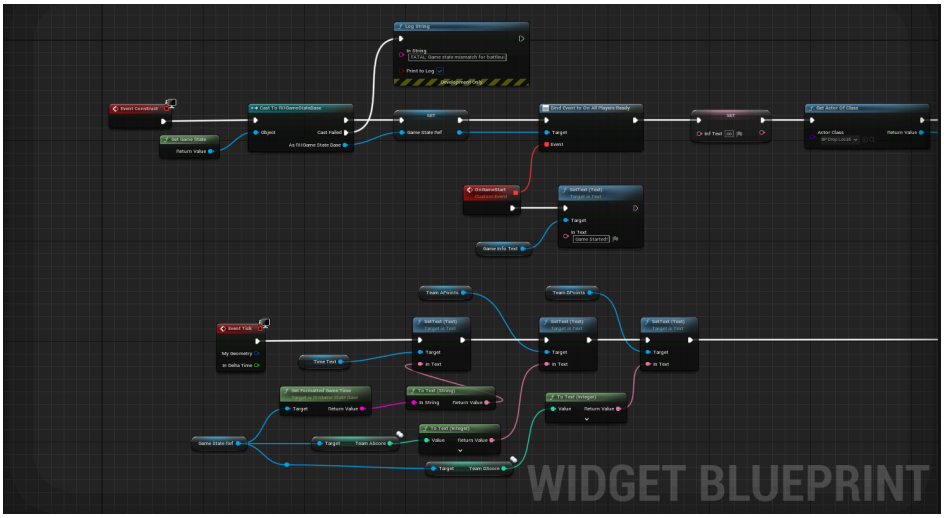
Parallel to this, RHGameStateBase acts as the repository for all dynamic game data, such as team scores, battle durations, and spawn points. Unlike the game mode, the game state actually exists across all the client and server. Thus when its data is exclusively set and updated on the server, it always replicates the information to all clients. This ensures that all players receive timely and synchronized updates about the game state. Clients are equipped with a series of dynamic delegates that respond to these updates by refreshing UI elements and visual cues, thus keeping the player's view consistent with the game's current state.

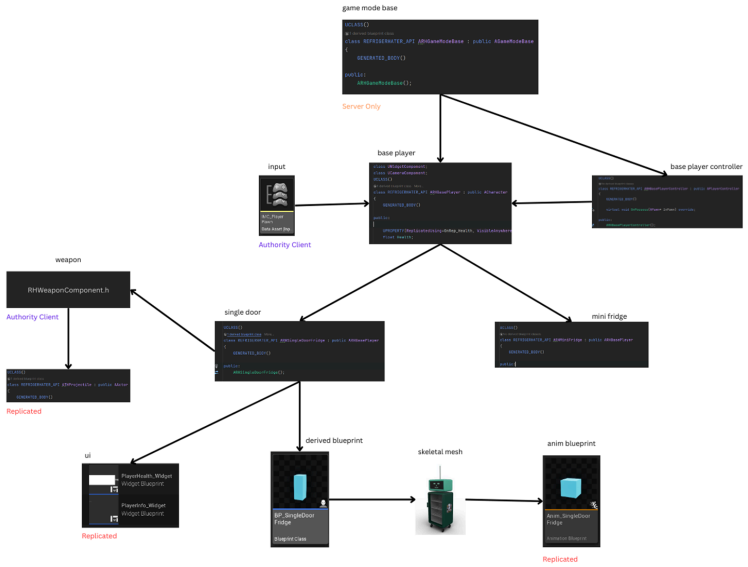
On the player side, the architecture separates concerns effectively between the server and client. Critical logic such as damage reception, resource pickup, and weapon firing are computed on the server in the RHBasePlayerController.cpp and RHBasePlayerPawn.cpp scripts to prevent cheating and ensure fairness. Conversely, player inputs and movements are handled locally on the client side to minimize latency and enhance responsiveness. This distinction ensures that while the server maintains strict control over game logic, the player experience remains fluid and engaging.

Relation between C++ Classes and Unreal Blueprints:

The project being 90% C++, it still utilizes Unreal's powerful blueprints system at places, especially for the UI logic and maintaining scene/asset references between object instances. This has given us a great flexibility to focus on the C++ code preliminary for the game's foundational logic and then expose high level functions for blueprint and Ui graphs. This also makes debugging easy and separates project structure really well. The second image below demonstrates how the structure starts with RHPlayerBase.cpp, it then is inherited by the RHSingleDoorFridge.cpp, when again is inherited by a blueprint called bp_singledoorfridge.

Unreal blueprints are also exceptional for handling character animation lifecycle and also makes integrating UI to the game logic extremely easy. Whereas its C++ api is perfect for any low level game logic that needs to be built. Also whenever we got stuck with the C++ implementation we could simply get deep into the game engine code and that gave us a lot of insights on how the overall engine behaves!





EOS implementation:

For the EOS implementation there are 7 main classes which handle the flow for the players, the other classes generally act as UI elements

URHEosGameInstance: This is really the core class, this class handles the connection to the EOS. There are methods for logging in, creating the session (or lobby - though there is a distinction), joining sessions and loading the levels (i.e. lobby level and gameplay level). It inherits from Unreal's UGameInstance.

URHMainMenu: This is the UI for the Main menu from which the user can click on a UButton to log in, create a lobby, find a lobby, select and join a lobby.

URHLobbyMenu: This is the UI for the lobby menu from which the user can click on a RHCharacterEntry (another widget used to display character choices) and select their

character. The other lobby members are displayed in a list view which is populated via script.

ARHLobbyGameMode: Ensures that the sessionName (lobby name) is replicated across clients, etc.

ARHEosPlayerController: Mainly responsible for overriding the OnRep_PlayerState() in the parent which is called when the player state is replicated. From here a custom blueprint event is called which should allow us to replicate things like character selection. Unfortunately this doesn't always seem to work - perhaps more work on the blueprint itself needs to be done.

ARHEosPlayerState: When the player's character selection has been replicated across clients, the value of the character will be stored here.

URHCharacterSelectionType: This stores information like the skeletal mesh to display, character name and icon and of course the blueprint class to be spawned. It works similarly to a Scriptable Object in Unity.

Gameplay Mechanics and Balancing:

You Win, Team B!

Diverse Player Classes: The current design of player classes implementation makes it extremely easy to add more player types to the game, as they already share the same root/parent class and most of the player logic is written generically irrespective of the player type, the changes in player behaviors can be easily adjusted by altering the player config data like walking speed, health etc.

Resource Dynamics: The game's resource system introduces an interesting layer of strategy. Resources not only differ in their point values but also come with special abilities or drawbacks. Although we could not implement the special abilities part on time the architecture still considers it and the player weapon has the provision to fire different kinds of ammos obtained based on the carrying resources like with beer crates players can through beers etc, the game also implements legendary and general resource types.

Spawn Mechanics: Resource spawns and deposit location movement are timed and vary in rarity throughout the match. This mechanic encourages players to keep moving and strategizing according to the changing game environment. Balancing the timing and location of these

spawns was critical to ensure that the game pace felt right and that players had equal opportunities regardless of their starting positions.

Results and Feedback on the Implementation:

We did multiple levels of internal testing among ourselves in the local network with the listening server, and although the game lacked art assets and some key features like resource special abilities, it was already extremely fun and we could see the potential that a full release build would have after probably months more of polish with a bigger team. Refriger-hater is the kind of game that would be a lot more fun when played with friends rather than strangers from the internet, especially friends being in the same team to strategize the battle!

But that being said we do see how the game is not fully polished to be published into the store, there are some branches in code where more checks and balances is needed as well, and we would personally wish we could work on it a little more to polish already existing features and fix bugs during the process.

Conclusions and Future Work

The bones of this project are in a good place and we've set it up in a way which will allow us to build a realistic multiplayer game. Unfortunately the design work is lacking and we were not able to integrate assets or polish the game as much as we would have liked to. The programming workload for this was quite substantial and, though we knew this from the start, Unreal 5's complexity increased the workload further. There were some issues with merging branches in the end as well.

Even though we began from a multiplayer perspective we should have integrated EOS earlier or, ideally, started with a project with this already setup - though this would have still taken a good amount of time. Overall the challenge was worth it though as what we've learned is greater than what we would have learned by using Unity, though it certainly came at the cost of more than a few headaches.

Future Directions:

We honestly believe Refriger-hater is a unique approach towards fast paced resource gathering games and a fully finished game with all the art assets, sound and effects can be extremely engaging and fun to play with! So we have thought about potential additions to the technical side of the game.

Resource Unique Attack: This was actually part of our plan already but we had to cut down the scope to finish the game on time, the idea here is when a player picks up a resource they should be able to throw it until all the resources they had in the fridge are used! Our current implementation is already built with this setup in mind.
Analytics: It is important to understand how our level and resources are functioning with gamers, which hero they often use, which resource is overpowered etc.

A Mini Map: In a fast paced resource gathering game a mini map can go a long way! It can make the battle more engaging and will help teams to strategize better.

Audio connection and Game Audio: A game like ours needs a lot of communication between teammates so audio in game will massively boost overall engagement and the game lacks audio and music.

References

- Unreal Engine Official C++ Documentation: https://dev.epicgames.com/documentation/en-us/unreal-engine/networking-and-multiplayer-in-unreal-engine?application_version=5.3
- Tom Looman's Unreal C++ Guide: (General C++ guide) https://www.tomlooman.com/unreal-engine-cpp-guide/
- Unreal Multiplayer Network Compendium: https://cedric-neukirchen.net/docs/category/multiplayer-network-compendium/
- Multiplayer Tips and Tricks: https://wizardcell.com/unreal/multiplayer-tips-and-tricks/
- Unreal Collision Filtering: (Info for setting for collision in the game) https://www.unrealengine.com/en-US/blog/collision-filtering
- Delegates In Unreal Engine: (Info for game's Dynamic Multicast delegates) https://benui.ca/unreal/delegates-advanced
- Network Clock: medium.com/@invicticide/accurately-syncing-unreals-network-clock-87a3f9262594
- Hits and overlaps in Unreal: (C++ Code snippet for collision overlap and hit) https://dev.epicgames.com/community/learning/tutorials/zw7m/hits-and-overlaps-bp-c-multiplayer
- Rajen Kishna, 2021. Introduction to Epic Online Services (EOS). Available at: https://dev.epicgames.com/en-US/news/introduction-to-epic-online-services-eos
- Unreal 5.3 EOS Issue. Available at: https://eoshelp.epicgames.com/s/article/Why-am-I-getting-the-error-LogEOSLobby-Lobby-membership-count-inconsistency-using-the-EOS-OSS-plugin?language=en_US
- SebBergy, 2023. The EOS Online Subsystem (OSS) Plugin. Available At: https://dev.epicgames.com/community/learning/courses/1px/unreal-engine-the-eos-online-subsystem-oss-plugin/JGKP/unreal-engine-eos-p2p-lobbies-and-voice
- Unreal's Enhanced input System Plugin: https://dev.epicgames.com/documentation/en-us/unreal-engine/enhanced-input-in-unreal-engine?application_version=5.2
- External Images used in the report:

    - Unreal base actor ownership behavior image from: https://www.linkedin.com/pulse/unreal-engine-networking-multiplayer-games-part-1-mika-laaksonen/
    - RPCs ownership behavior context comparison image from: https://docs.unrealengine.com/4.26/en-US/InteractiveExperiences/Networking/Actors/RPCs/
- External Resources/Assets used in the game:
    - "SWorldUserWidget.cpp" script for world UI spawn logic: https://github.com/tomlooman/ActionRoguelike/blob/master/Source/ActionRoguelike/Private/SWorldUserWidget.cpp