|  |
| --- |
|  |

**Technical Design Document**

**Game Name**

**Shahil Saha**

**Tables of Contents**

Contents

[Introduction: 3](#_Toc152200245)

[Platforms: 4](#_Toc152200246)

[Engine Summary: 5](#_Toc152200247)

[Game Systems and Diagrams 5](#_Toc152200248)

[Movements 6](#_Toc152200249)

[Weapon Shooting: 10](#_Toc152200250)

[Properties 14](#_Toc152200251)

[Art Workflows 19](#_Toc152200252)

[Optimisation & Profiling 25](#_Toc152200253)

[Coding Standards and Summary 26](#_Toc152200254)

# Introduction:

* + **Project Name**

Strike

* + **Project Genre**

3rd Person Shooter

* + **Project Key Goal/s**

To create a multiplayer shooter game where the goal is to capture the opponent’s flag

* + **Project Key Challenges and Risks**

Managing the latency

Setting up an efficient network system.

Firewalls present a potential hurdle that can be difficult to navigate. The transmission of packets between systems necessitates validation by the firewall, and trialing the project within a university setting might pose restrictions.

* + **Software and Hardware Required**

Unreal Engine 5

Github

JetBrains Rider

Audacity

ScreenBits

Diagram.net

Processor: i7 and up

Graphics: RTX 2080 and up

Ram: 32gb and up

**MoSCow List**

|  |  |  |  |
| --- | --- | --- | --- |
| **Must** | **Should** | **Could** | **Wont have** |
| Server Menu | Scoreboard | A system to add an NPC bot if the team is uneven to balance it out | A Voice chat system to communicate with the teammates. |
| Server Host System | An extended matchmaking system to allow better customization for the host player | A HUD settings menu |  |
| A replicated Controller and a possessed Character | Have multiple maps to choose from | Weapon Swapping |  |
| Health System | Shop Menu | Inventory |  |
| At least one map to play on |  |  |  |
| Game Economy |  |  |  |
| Flag Capture mechanic |  |  |  |

# Platforms:

* + **PC / Console**

This game will be played on PC.

* + **Minimum Platform Specification / Recommended Specification**

***Minimum Spec***

OS: Windows 10 64-bit

Processor: Intel Core i5-2400 or AMD Ryzen 3 1200

Memory: 8 GB RAM

Graphics: GeForce GTX 750 Ti or Radeon HD 7850

DirectX: Version 11

Storage: 40 GB Available Space

***Recommended Spec***

OS: Windows 10 64-bit

Processor: Intel Core i5-6600 or AMD Ryzen 3 3100

Memory: 8 GB RAM

Graphics: GeForce GTX 780 or Radeon R9 290

DirectX: Version 11

Storage: 40 GB Available Space

* + **Engine Specifics Specifications and Limits**

**• Storage/Disk Budget**

4 GB

**• Texture Compression**

Default Compression

**• Shader Model/Instruction Limits**

No Specific Limits

**• Optimisation for different consoles**

This game will only be played on Windows, so no optimization for other platforms is necessary

# Engine Summary:

* + **Engine Version**

Standard version of Unreal Engine 5.2.1 (Stable Release)

* + **Plugins / Middleware**

Niagara plugins: Will be used for adding particle assets to the project

Default Online Subsystem: To Support Session handling features like creating, starting and joining sessions.

# Game Systems and Diagrams

**Core game pillars:**

For the core game pillars, I decided to go for the Call of Duty game features for Capturing the Flag. The first pillar is the Client Server network model. This means that there would be an authoritative host, and the clients would connect to the host to play it together.

Since it is a team-based game, tactical strategy is an essential pillar of my project. This is because this gives the players an opportunity to develop and execute tactical plans, including offensive and defensive strategies.

Finally, another essential pillar is “Objective-based Competition.” Since the focus of this game is flag-centric, and the main objective is to capture the opponent's flag and bring it back to the team base, this makes up one of the most essential pillars of this project.

A diagram of a structure

Description automatically generated

**Core Mechanics**

**Player Mechanics**

### Movements

#### Jog:

The player will be able to move in all directions using the WASD keys. This would be useful when escaping the enemies or dodging them.

The default movement of the player will be jogging.

#### Jump:

The player will jump using the SPACEBAR key. This would be needed to climb blocks or other objects within a certain height from the ground.

#### Buy:

The player can access the shop menu using the B key in order to buy weapons, ammo, and other gear.

#### Aim/Shoot:

The player can use the RMB (Right Mouse button) to aim and LMB (Left Mouse button) to fire weapons. The Aim would be helpful if they were using a sniper weapon.

#### Flag Interaction:

The player can interact with the flag by capturing, dropping, and retrieving it. They can interact by pressing E within the range and drop it by pressing G.

**Capture the Flag**

The primary goal is to attain the predetermined target before the rival team does. There are two methods to accomplish this.

Firstly, a player can eliminate members of the opposing team, thereby accruing points for their own team.

Alternatively, players can infiltrate the enemy's base, seize their flag, and return it to their own base to earn points.

#### Goal

The two-fold scoring system enriches the gameplay experience by introducing varied tactics and playstyles, thereby adding more depth and excitement to the game. This development elevates the competitive nature of the game and accentuates its strategic elements, motivating players to adopt more deliberate and tactically sound approaches to their gameplay.

A diagram of a diagram

Description automatically generated

**Network and Multiplayer Mechanics**

The game uses a peer-to-peer connection model where one player hosts the session and others connect to it via LAN.

#### Hosting a Session

Players looking to host a game can easily do so through a user-friendly interface. From the main menu, selecting the "Host Game" button brings up the Host widget. Here, the host player can tailor the session settings to their preference, adjusting variables such as Target Score and starting money. Once the settings are fine-tuned, clicking the "Host" button initiates the session, transitioning the host directly into the map.

#### Finding and Joining a Session

For players seeking to join an existing game, the process is streamlined for quick and effortless access. Clicking on "Find Server" in the main menu opens an overlay widget, which displays a list of all active sessions on the LAN. This list is dynamically updated, ensuring players see real-time session availability. Players can then select and join any listed session, allowing them to jump into the game and connect with others.

#### Leaving a Session

Players have the option to exit an ongoing session whenever they choose. By opening the in-game pause menu, players can select the "Leave Session" button. This action disconnects them from the current session and returns them to the main menu, from where they can host a new game, join another session, or exit the game.

**Game Mechanics**

**Health System**

During gameplay, players can engage in combat and shoot at each other, which can result in health damage. However, if players are on the same team, friendly fire does not apply, and team members do not lose health from accidental attacks by their allies.

During gameplay, players can engage in combat and shoot at each other, which can result in health damage. However, if players are on the same team, friendly fire does not apply, and team members do not lose health from accidental attacks by their allies.

Players may want to regenerate health during the gameplay, they can do this by buying a Medi kit from the shop.

When the player fires a weapon, a hit trace is shot from the camera straight toward the centre of the screen for a specified distance from the camera. If the hit trace returns a positive hit result, it checks whether the actor implements the Player Interface. And if that’s true, damage will be applied to the opponent player. But

A white rectangle with black text

Description automatically generated

A white rectangle with black text

Description automatically generated

**Weapon and Item Purchases:**

The player gets the opportunity to buy their weapons and equipment. As long as the players are frozen at their spawn locations, they will be able to access the buy menu. This allows players to buy their weapon equipment for a specific time at the start of the match.

The system will check whether the menu is opened using a bool “mIsOpened” and will spawn the menu if the value is false.

A diagram with text and arrows

Description automatically generated

**World Object Interaction:**

Players can interact with various objects in the game world, including grabbing weapons off the map and interacting with the flag.

**Flag Retrieval and Reset:**

The flag's retrieval and reset mechanics are designed to enhance strategic gameplay. Only the team that owns the flag can reset it back to their base. Additionally, if a player manages to deliver the flag to their base, it automatically resets to its starting position. The system also actively monitors interaction with the flag; if a player from the opposing team touches the flag, it becomes attached to them. Should there be no interaction from the opposing team, the flag will revert to its original base. Furthermore, any collision involving the flag triggers a check to determine if it has entered the opponent's base. If so, the flag is again reset to its original location.

**Scoring and Objectives**

The team gets a score when either of the conditions is met:

1. The player killed an opponent.
2. The Player managed to get the opponent’s flag to the base.

A diagram of a graph

Description automatically generated

### Weapon Shooting:

* + The shooting mechanic is enhanced with visual effects like muzzle flashes, bullet trails, and impact decals, coupled with camera shake. This combination creates a realistic and engaging shooting experience, making players feel like they're genuinely in the middle of the action.

Health And Damage

When a player takes health damage, the system checks if the player health is greater than zero. If the condition is true, then the player dies, waits for a few seconds, and again respawns at the team's base. Otherwise, if the condition is false, it will continue with the gameplay.

A diagram of health and damage

Description automatically generated

Economy

For this project, the money is reset every time the player is respawned.

A diagram of a game economy

Description automatically generated

* + **Game Loops**

Core Gameplay Loop

A diagram of a flag

Description automatically generated

Scoring Loop

A diagram of a diagram

Description automatically generated with medium confidence

Elimination Loop

A white ovals with black text

Description automatically generated

Capturing Flag loop

A black background with white ovals and words

Description automatically generated

**Data Structures**

|  |  |
| --- | --- |
| **Main/Base Classes** | **Uses** |
| Game Mode | This will be responsible for managing the entire game. |
| Game State | This will be responsible for managing the state of the game, like managing the team scores and updating them. |
| Input Controller | The controller will hold all the key bindings and will also be responsible for handling mapping contexts. |
| Player Character | This is the script for the player inside the game and will hold all the control implementations like move, look etc. |
| Base Widget | This class will be used as the parent to create all the widgets inside the game. |
| Base Weapon | This class would have all the data related to a weapon, which will include things like rate of fire, damage, ammo, etc. |
| Game Instance | This is a class that spawns at the very start of the game and persists till the player. exits out of the game. This is responsible for managing project sessions such as creating, joining, and ending them. Additionally, it will also store the data which will persist across levels. |

|  |  |
| --- | --- |
| **Child Classes** | **Uses** |
| Base Widget | |
| Main Menu | This would contain the Title of the game and the option buttons for Hosting and Finding a session and Quit |
| Find Server | When the player clicks the "Find Server" button, an overlay widget will appear on top of the main menu. This widget will display a list of all available servers on the network, allowing players to choose and join them. |
| Host Server | Just like the “Find Server”, this is also an overlay widget and contains a list of options like Starting Money and Target Scores to set and a “Host” button which creates a session and opens the map. |
| Overhead UI | This widget displays the username and health bar on top of all players when they are created. The username appears in either blue or red, indicating their team (Terrorist or Counter-Terrorist), and the health bar updates when a player is shot. |
| Pause Menu | This widget is spawned by pressing the ‘P’ key which comes with the options to Resume the game, leave the session, or quit the game. |
| Player HUD | This Player HUD is created when the player is spawned and contains details like the health, ammo, and team scores. |
| Team Choosing UI | At the start of each session, a widget is created before the player character is created. The player is given the option to select their preferred team and character. After the player has made their selections, the pawn is then created. |
| Base Button | |
| Shop Button Widget | This class contains data for the main tab in the shop widget, such as Primary, Secondary, and Ammo. |
| Shop Item Button | This class contains data for weapon item buttons, such as buttons for different guns. |
| Base Weapon | |
| BP Base Weapon | This is the Blueprint version of the Base Weapon class, and the weapon actors derive from this base weapon class. |

## Properties

|  |  |
| --- | --- |
| **Game Instance** | |
| mPlayerDetails | Is used by the main menu to store Player details that will be carried over to the other levels. |
| mMatchDetails | Is used by the main menu to store the details about the session and needs to be replicated across all the clients |
| mMultiplayerSessionsSubsystem | This is to store the reference to the Subsystem |
| mTravelMap | This is to store the reference to the map where the players compete against each other |
| mMainMenuLevel | This is to store the reference to the main menu map. |
| Username | To store the edited username in the main menu |

|  |  |
| --- | --- |
| **Player Controller** | |
| InputData | Stores the reference to the Data asset which stores all the input actions related to the player |
| UIInput | Stores the reference to the Data asset which stores all the input actions related to the UI |
| mPlayerRef | Stores the reference to the player |
| mPlayerState | Stores the reference to the player State |
| bHasRestarted | To check whether the pawn has restarted |
| mHudRef | Stores the reference to the HUD class |
| SpawnWeaponSignature | Called by the Shop Widget when the player clicks on a weapon button |
| OnPawnDeadSignature | Called by the Player state when the health is less than or equal to 0 |
| CharacterMeshDetails | Stores the reference to the player’s chosen mesh |
| InputMappingContext | Stores reference to the mapping context related to the player input |
| UI\_MappingContext | Stores reference to the mapping context related to the UI |
| mMinCamPitch, mMaxCamPitch | To store the camera’s clamped pitch values |

|  |  |
| --- | --- |
| **Game Mode** | |
| Controllers | Stores the reference to all the controllers that are connected to the session |
| TeamBases | Stores the reference to all the bases that are there in the map (mainly 2). This will be used by the FindStart() method to find a suitable place to spawn the player start |
| PointsToAddOnKill | Stores the total points to add when an opponent is killed. This is used by the Game State to calculate the results |
| PointsToAddOnCapture | Stores the total points to add when the opponent’s flag is captured and being returned to the team’ base. This is used by the Game State to calculate the results |

|  |  |
| --- | --- |
| **Base Weapon** | |
| mWeapon | This is a skeletal component which stores the actual gun mesh |
| mWidgetClass | This is the reference to the pop up widget class |
| mMuzzleSocket | This stores the reference to the Muzzle Socket in the gun skeleton mesh |
| mOwnerRef | This stores the reference to the owner of the weapon |
| mInteractableDetails | This is to store the interaction details about the weapon |
| TraceChannel | This stores the reference to the trace channel that will be used during the hit trace |
| TimeHandler | This is a reference to the time manager which is created when the player starts firing and destroyed when the player stops firing |
| mWeaponDetails | This contains all the details related to the weapon like the fire rate, image, class reference etc. |
| WeaponAsset | This is the reference to the Data Asset used for spawning the weapon |
| mInFirstDelay, | This is the delay that the weapon takes before it fires the first bullet. |
| mSpread | This is the value which indicates how much a weapon bullet should spread when fired. |
| mFireRate | This is the rate at which the bullets will be fired from the weapon |
| TraceRange | This is the trace range of the weapon and indicates the distance from the source till the end point |
| MuzzleDuration | This indicates the time the muzzle will last before getting destroyed |
| mWeaponSound | This is the sound the weapon will play when its fired |
| mDamageRate | The damage it will do to the player on being fired |
| mAmmo | The ammo value of the weapon |

|  |  |
| --- | --- |
| **PlayerCharacter** | |
| mPrimaryWeapon | Stores reference to the weapon after being equipped by the player. |
| InitialCamTransform | This stores the initial transform of the camera so that when the player aims at something, it moves forward and once the player stops aiming, the camera can return to the default position. |
| FlagSocket | Stores the reference to the flag socket in the character mesh. This is used by the flag when it spawns and attaches to that socket |
| isAiming | Is used by the animation blueprint to trigger the aim animation |
| bIsDead | Is used by the animation blueprint to trigger the dead animation |
| CollidedActor | Stores the reference to the overlapped interactable actor so that if the player decides to interact with it, this reference can be used to trigger the Interact interface method in the actor |
| mFlagRef | Stores the reference to the flag actor after it spawns as a child on the player. |

|  |  |
| --- | --- |
| **Player State** |  |
| Health | This is used by the player state OnDamageTaken delegate to check if the player is dead and calls the controller to request for updating the player hud or the player if its dead. |
| mPlayerDetails | This struct type variable is used by a lot of classes who wants to access the data related to the player |

|  |  |
| --- | --- |
| **HUD class** | |
| OnTeamChosen | This is called by the Team Choosing UI widget when the player chooses their desired team. Which than calls an interface method in the controller that sets up the player on the map. |

* + **Data Flow**

For this project, the initiation chain is set up as follows:

* The Game Mode, being the master class, is spawning the controller with a spectator pawn. Additionally, it is also spawning the Game State
* The Controller then spawns the HUD, Player State.
* The Controller then calls the HUD class to spawn the Team Choosing UI for the user.
* Once the User chooses the team, the controller then spawns the pawn on the map with the relevant details.
* Finally, the HUD class is triggered by the controller to spawn the player HUD.

Various kinds of data need to be passed into the Widget components when they are spawned to get them to work as expected. For this project, a lot of widget classes, especially the buttons that are spawned on runtime, are taking References as input when they are created. This data is then required by these widget components to bind members inside the class or update data on the parent widget.

A black background with white rectangles

Description automatically generated

|  |  |
| --- | --- |
| **Interface** | **Uses** |
| PlayerInterface | This interface is designed to manage key player-related functionalities in the game. It includes methods for spawning weapons, adding ammunition, and setting up weapons. |
| PlayerInputInterface | This interface is integral to how players interact with the game world. This includes all methods related to player actions, such as moving around, interacting with objects, jumping, and shooting. |
| ControllerInterface | The ControllerInterface plays a crucial role in updating in-game HUD and widgets, as it has direct access to the HUD class. It also includes methods for setting up and respawning pawns. This interface is essential for maintaining the user interface and managing player characters, ensuring smooth gameplay and effective player feedback through the HUD. |
| HUDInterface | HUDInterface contains all the methods related to widgets since this class is responsible for managing everything related to widgets, like spawning/ destroying widgets |
| GameInstanceInterface | This interface is used by the different classes to store data that needs to be persistent across all levels, like Setting and getting match details that were set in the main menu. |
| InteractableInterface | This interface is used on the objects that can be interacted with by the player on the map. |
| PlayerHUDInterface | The PlayerHUD needs to be updated very often throughout the gameplay. Therefore, interfaces have been given more preference over Casting since it can be computationally expensive. |
| PlayerStateInterface | The PlayerStateInterface contains methods that are used for updating player-related data like health, Team, etc |
| WeaponInterface | This contains methods for Firing weapons, which are implemented by the Weapon class and called by the player class when the player fires/stops fires. |

|  |  |
| --- | --- |
| **Data Assets** | **Uses** |
| DA\_CharacterMeshDetails | This Data Asset stores all the data about all the different types of character mesh. |
| DA\_Input Data | This stores all input actions for character movement and actions and is implemented by the Player Class. |
| DA\_UIInputs | This contains all the inputs related to the Widgets which is implemented by the controller |
| DA\_WeaponBuy | This contains all the information about the Shop widget in the main menu. |
| DA\_WeaponDetails | This data is specific to weapon assets and is used by the Shop widget to create buttons. The controller then sends this data to spawn the attached actor for the player. |

# Art Workflows

* + **Materials & Textures**

Most of the Materials and Textures will be taken from asset packs and the Unreal marketplace. However, some of them need to be tweaked to make it look better. The main program that will be used to create them would be photoshop which is already licensed by the university.

All the textures will be in PNG format.

* + **Animation**

As it is a Shooter game, there is very minimal requirement of complex animations. Therefore, this project will be making use of the animations that comes with the starter kit from Unreal Engine and they would be retargeted to be able to be used by the custom characters.

* + **Particle System & Effects**

The use of particle systems in this project is very limited as it’s a shooter game, and the intention is to make the game look very classic. However, the weapons do make use of the Niagara particle system for representing the muzzle flash. Additionally, the map itself comes with some decals, which makes it look more realistic.

* + **Lighting**

For this project, we will be using the Static SkyLight. This is a more preferable option over dynamic lighting as it will help keep the project simple and avoid over-scoping. Dynamic lighting should only be used when necessary, and since this is a multiplayer game, it would be best to avoid using it to prevent any latency issues.

* + **Camera**

Developing a third-person shooter game requires careful consideration of the location. The player must be able to see ahead without straining their eyes and aim with precision. To achieve this, the crosshair should not overlap with the player's mesh, and the camera should be offset to the right or left of the player. This is a commonly used technique in the industry.

When the player aims at something, the camera should move towards the player. Instead of using multiple cameras, the same camera can be used with a bit of calculation to move back and forth depending on the player's input. This will allow for accurate and precise aiming.

* + **Collision**

The project will utilize the default collision settings provided by the engine. However, since it is a shooter game and the weapon will utilize hit tracing for player detection, a custom collision channel will be used. This will prove advantageous in identifying which actors need to be ignored and which ones need to be traced.

* + **Menu Wireframe**

Main Menu

A screenshot of a computer

Description automatically generated

Host Server

A screenshot of a computer

Description automatically generated

Find Server

A screenshot of a computer

Description automatically generated

Player HUD

A screenshot of a game

Description automatically generated

Buy Menu

A screenshot of a computer

Description automatically generated

Team Choosing UI

A screenshot of a computer

Description automatically generated

Game Info UI

A screenshot of a computer

Description automatically generated

A diagram of a menu

Description automatically generated

* + **Fonts**

For this project I will be using the RobotoMono font style from:

[Roboto Mono - Google Fonts](https://fonts.google.com/specimen/Roboto+Mono)

Additionally, for the world UIs, I will be making use of the default font that comes with the engine.

* + **Audio / Video**

Audio:

File format: .wav

Bits: 16/24 bit

* + **Assets**

This project will use assets from the unreal asset store. Multiple different asset packs would be used to make different assets and use them in the project.

The first asset pack is [Elite Soldier](https://www.unrealengine.com/marketplace/en-US/product/elite-soldiers), which will be used for the player characters in the game. This asset includes four different characters with different sets of meshes, which can be combined to create some new ones if required.

The second one is the Lyra Animation Starter pack which comes with all the different types of weapon combat animations that are a requirement for this project.

The map will be taken from the Lyra starter pack, which comes with a demo map that is good enough for a game like this. Additionally, the meshes can used to create our own meshes.

As the game’s primary objective is Capture the Flag, [this](https://www.unrealengine.com/marketplace/en-US/product/flags-package-250-flags) asset pack would be used to get that moving effect of the flags as well as, and the primary material would be used to create custom materials, which will be used to indicate the team color.

# Optimisation & Profiling

Throughout the development process, planning application will be utilized to meticulously monitor the progress. This tool will also serve as a repository for tracking and documenting any encountered bugs.

To ensure precision and quality, various mechanics and prototypes will be created in isolation. Once these components meet the expectations, this will be integrated into the main game system. Following integration, rigorous testing will be done, involving not only the newly added mechanics but also all previously implemented systems to identify any potential issues arising from the integration process like replication issues and broken systems. Should any problems surface, I will promptly record them in the planner for subsequent resolution.

Given that the game operates as a multiplayer experience over a LAN connection, the game will be tested on different systems to assess potential latency issues during gameplay. Additionally, we will verify the ability of other computers to successfully locate and connect to gaming sessions.

There are several methods for testing and debugging that can be beneficial for this project.

The first one is Log Prints. Utilizing log printing proves highly valuable for verifying if the system successfully passes all checks and doesn't overlook any critical elements. This simplifies the process of identifying bugs within the game.

However, log printing may not always suffice. In such instances, the use of breakpoints becomes necessary. Breakpoints enable us to step into each state, assess the values, and ensure they yield valid results. Furthermore, breakpoints assist in confirming whether the code is being executed by the system.

Nevertheless, there are cases where breakpoints may also fail to work. In these situations, we resort to "backtracking" to precisely determine the point at which errors occurred.

To track down issues related to networking “Network profiler” is an external tool that comes in handy which gives a graph-based data about the packets exchange. This includes things like bandwidth, latency, etc.

# Coding Standards and Summary

* + **Programming Standards**

In projects like this, particularly those involving numerous game mechanics, it's vital to have a clear sense of organization and adhere to coding standards diligently. Failing to do so can lead to code becoming a tangled mess sooner than expected, often without immediate awareness. Unreal Engine follows specific guidelines for replication, and it's essential to utilize the engine's provided classes as intended to maintain proper structure.

Another method of organising the code base is to put comments which should not exceed more than one line and explains what the specific piece of code is doing. However, at the same time, it is also important to make sure, we don’t overdo it. For instance, writing comments after every two lines or writing paragraphs.

A screenshot of a computer

Description automatically generated

Writing Optimized and avoiding hard coding is very important to maintain. This can be achieved by using variables to store data instead of passing values directly to a method. Additionally, repetitive logic should be wrapped up in methods or macros which keeps the scripts clean and makes it easier to manage as well as make it readable.

Using naming conventions are very useful. It is very helpful when identifying the type of property or file it is and can save a lot of time when trying to identify where a specific member is a global member and what type is it.

|  |  |
| --- | --- |
| **Naming Conventions** | |
| **Blueprints** | |
| Actors and Normal Blueprints | BP\_ |
| Data Assets | DA\_ |
| Materials | M\_ |
| Interfaces | BI\_ |
| Material Instances | MI\_ |
| Input Actions | IA\_ |
| Texture | T\_ |
| Widgets | W\_ |
|  |  |
| **C++** | |
| Properties | m |
| Server methods | Server\_ |
| Multicast methods | Multicast\_ |
| Boolean | b |
| Client methods | Client\_ |
| Blueprint Server Methods | BlueprintServer\_ |
| Blueprint Client Methods | BlueprintClient\_ |
| Blueprint Multicast Methods | BlueprintMulticast\_ |
| Base Classes | Base |
| Data Asset Classes | DA\_ |
| Widgets | U |
| Enums | E |
| Structs | F |
| Delegates | (Suffix) Signature |
| Interfaces | I |
| Actor | A |
|  |  |
|  |  |
|  |  |
|  |  |

A screen shot of a computer program

Description automatically generated