Mini Football Prototype Document

Hyunsoo Henry Park

Overview

This mini football game prototype is a third person perspective football game like Rematch, focusing on a 1vs1 setup with the basic features running in the multiplayer setup including the basic rules, interactions based on the collision and replication logic.

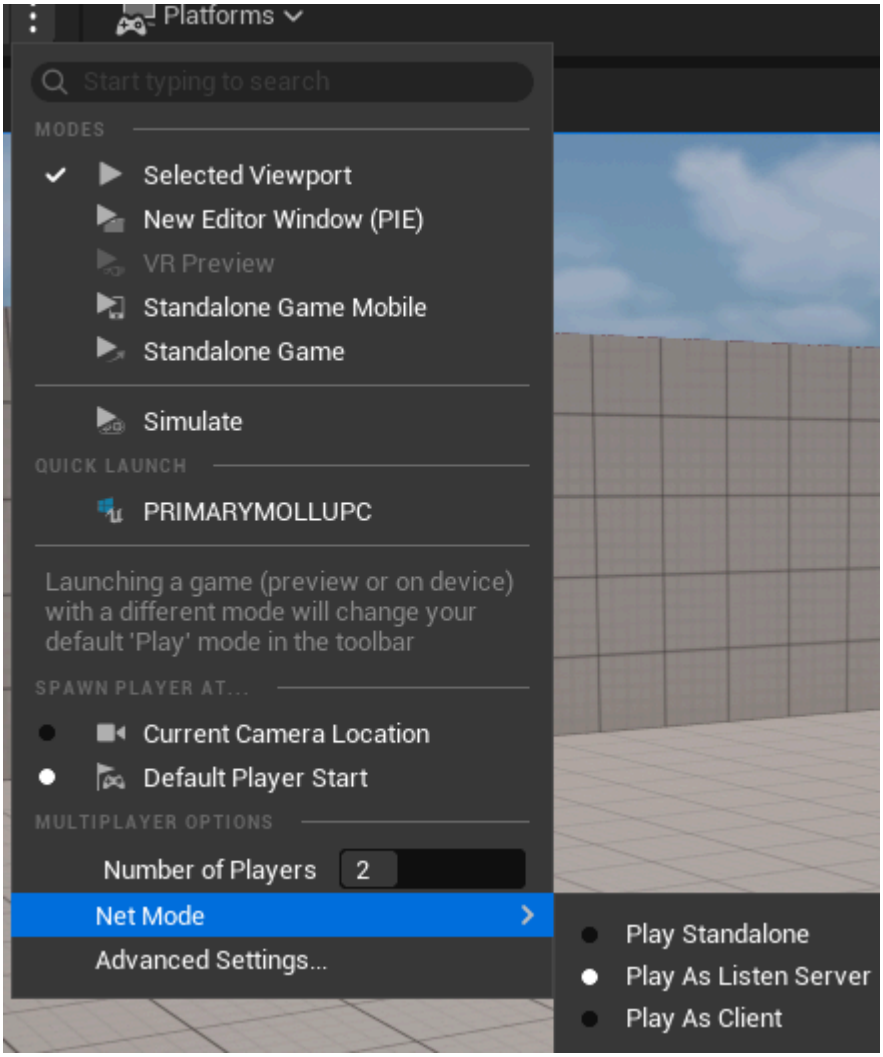
Specification

- ***Unreal Engine 5.6*** is required for running the project.
- Best work in the ***multiplayer setup*** - either listen server or dedicate server.
- 100% written in blueprint as requested.
- Total **5 days** spent to build the game, including the 3 days to build the logic and 2 days to refine the codes, testing and documenting

How To Play

WASD - Walk, Moving the character.
Mouse - Look
Dribble is enabled when plates move close to the ball while the ball colour changes to the yellow.
C (Hold to charge/Release to shoot) - Shoot the ball when the player dribbles the ball.
Left Shift (Hold) - Run

Game starts when minimum players logged in the session (Default - 2 players, can change the minimum players in Gamemode)



Set number of players as **2** and net mode either ***Play As Listen Server*** or ***Play As Client***

Name Conventions

All of the project related classes have the **FP_** prefix in the blueprint.
All of the member variables in the class have the **m** prefix in the variables.
Delegate/Dispatcher binded functions have the **On** prefix on its name.
Other than prefixes, the name conversion must follow the standard unreal engine naming conventions.

Approaches

To achieve a basic multiplayer football game prototype, those following features must be implemented.

- *Game mode to control the player in the session.*
- *Game stats to track the game status (e.g. players in each team, team scores, status of the game), control the game flow and control the player reaction by the event (e.g. goal event)*
- *Interactions among actors and call the dispatch to game stats if necessary.*

- *World and user interface to show game stats (e.g. score and goal indicator) to the players*

Class Structure

[FootballPrototype]
– [Actors]
– [Enums]
– [GameMode]
– [GameStates]
– [Input]
  – [Actions]
– [Map]
– [PlayerCharacters]
– [PlayerController]
– [PlayerStart]
– [PlayerStates]
– [UI]
  – [HUD]
  – [ScreenWidget]
  – [UserWidgets]

FP_A(Classname) - Field actors in the football field.
FP_E(Classname) - Enum class.
FP_IMC(Classname) - Input mapping class for the input mappings.
FP_IA(Classname) - Input action for the player action.
FP_GMFieldGameMode - Counts the number of logged in players and starts the game when the minimum players are logged in. Setting ball spawn position and ball to spawn. *mMaxPlayerForSession* sets the minimum player to start the game (2 as default).
FP_GSBField - Game state base for the field, controls the flow of the game and sends the event dispatcher when the game state changes. Also it sends the signal or directly calls to the specific team pawns (through player states).
FP_PlayerFieldCharacter - main player character in the football field. Contains the basic movement and ball interactions such as shooting and dribbling. Has ABall as the reference to verify the ownership of the ball.
FP_PCField - Mapping the input context to control the pawn.
FP_PSField - Take the logics and variables for the players state, in the prototype build it only contains the player team information (mPlayerTeam)
FP_FieldHUD - Main hud class to show in the player screen.
FP_W(Classname) - Widget class contains multiple widgets.
FP_UW(Classname) - Widget class as the single widget.

Tactics

Event dispatcher is great for calling the event at a specific time without the strong connection of each class. In this project, an event dispatcher is used to call the function in multiple classes or at the moment the event happens.

For instance, OnGoal dispatcher sends the dispatcher calls the function in the game state and ball class at the moment the ball collides with the goal post collision.

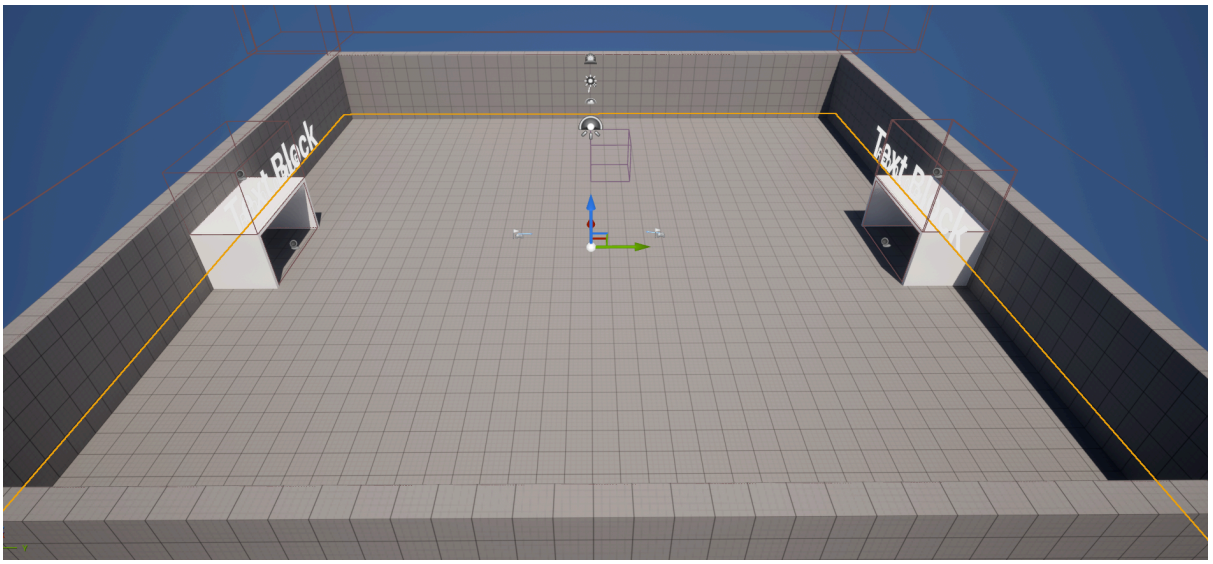
Game States is the core class to send the event dispatchers which take the game session status as the EGameStatus, representing the specific time for the game events. This guarantees to get the time the event happens.

Otherwise, if the class has the strong reference of the other class it may use the direct call. PlayerFieldCharacter, for instance, makes the direct call when the player shoots the ball from the ball reference the player character has.

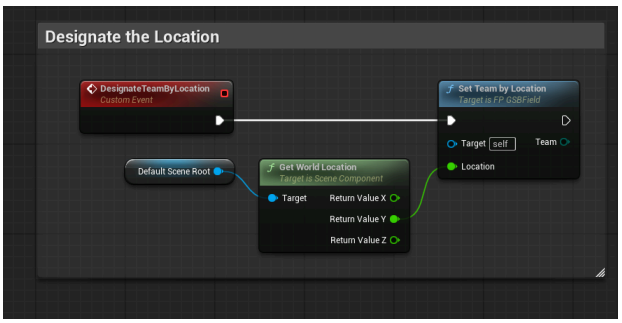
This hybrid approach gives the flexibility of the structure to both minimize the unnecessary strong references and accurate function calls.

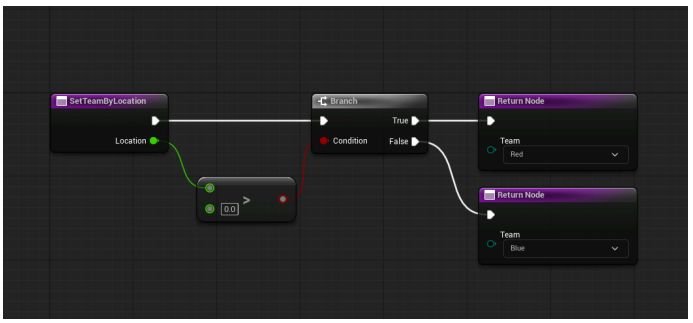
Timing is crucial in multiplayer development. Actors may spawn in different times if the network delay exists. Due to the nature of the network delaying, event dispatchers might be a great approach.

Basic Map setup



To set up the football field on the map, each actor must be designated the team owner. Each team must have its goal post and spawn positions.





Game stats has the logic to designate the owner team of the actors based on the world position. Any actors placed negative by the y-axis in the world coordinate (which is the left side of the world centre) is designated to the blue team, otherwise the red team.

Once an actor has been spawned with the owner team, ***game mode*** can spawn the player on the right side of the pitch. In the prototype game, the blue team player is spawned in the left side of the pitch and the red team player is spawned in the right side of the pitch.

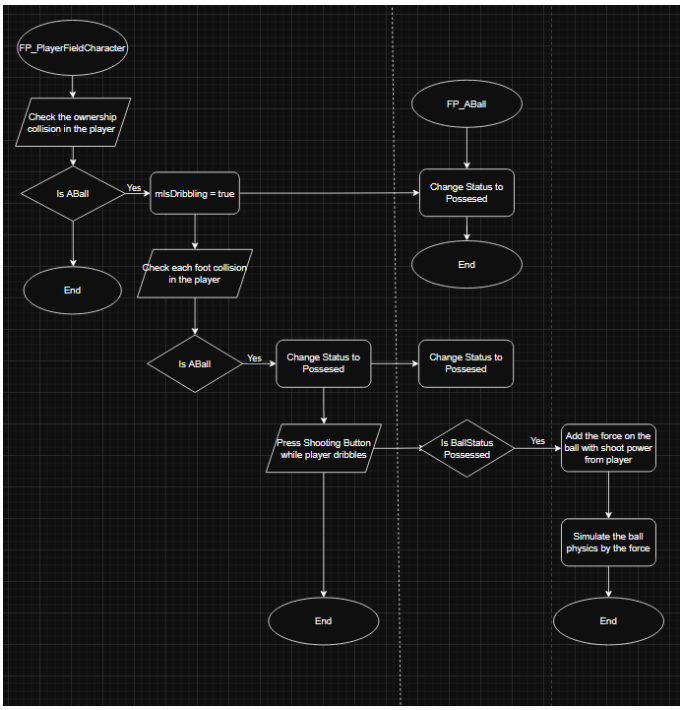
Ball and Goal Behavior

For implementing the dribbling and ownership (who dribbles the ball or can player shoot the ball), player character has two types of the collision: Ownership collision and foot collisions on each side of the ball. Ownership collision is for verifying the ball's ownership, which can make the player shoot. Foot collision, otherwise, is for the dribbling automatically gives the force while the player running for smoother dribbling experience. (This is the bare minimum physics interaction for the prototype)

The player has the ball reference while the collision has been collided by the ball for interacting with the right ball actor in the world.

EBallState is for verifying the ball status and internally verifying it can interact with the player. Ball state must be replicated to the other local player, which controls the movement and visual effect of the ball.

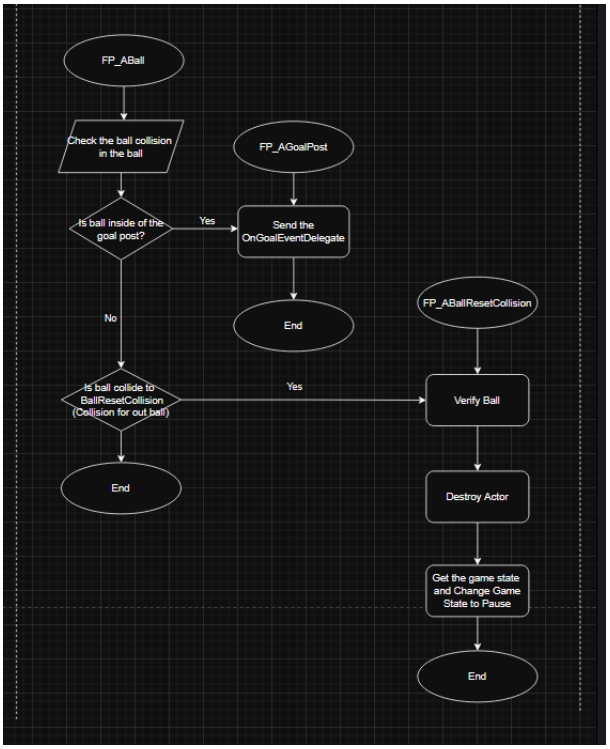
Character and ball logic flow is like below.



Goal post is the key actor to verify and send the goal event.

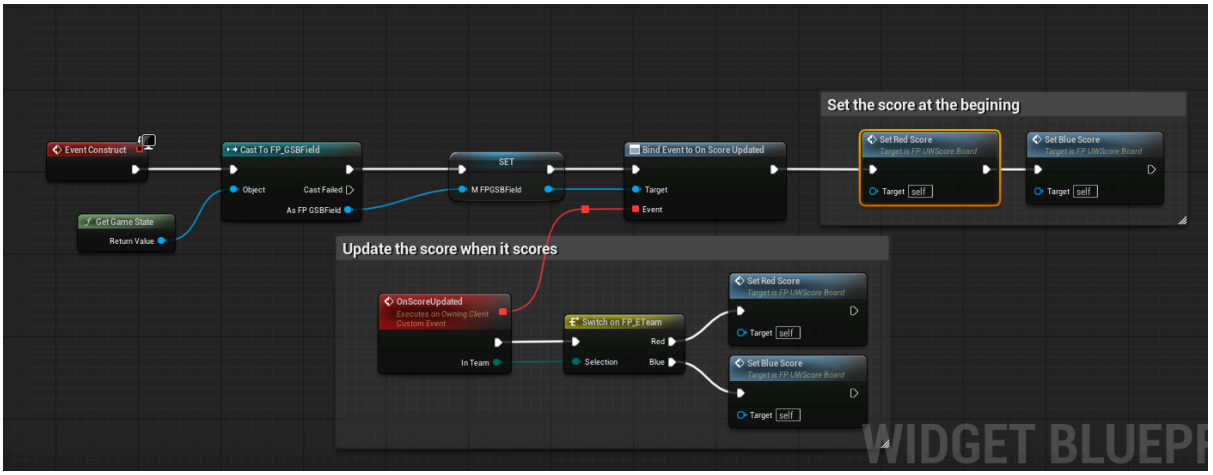
Goal post has its own collision to activate the goal event when the ball is in the goal post collision. I used the dispatcher event to send the on goal event to other classes and actors to run the relevant events in the moment the goal event happens.

Balls can goes outside of the pitch if the player shoots the ball too strong, I put ABallResetCollision for every side of the pitch to destroy the ball and reset the game as the initial position (in the prototype) it triggers on pause event and changes the game status to pause game in the game states.

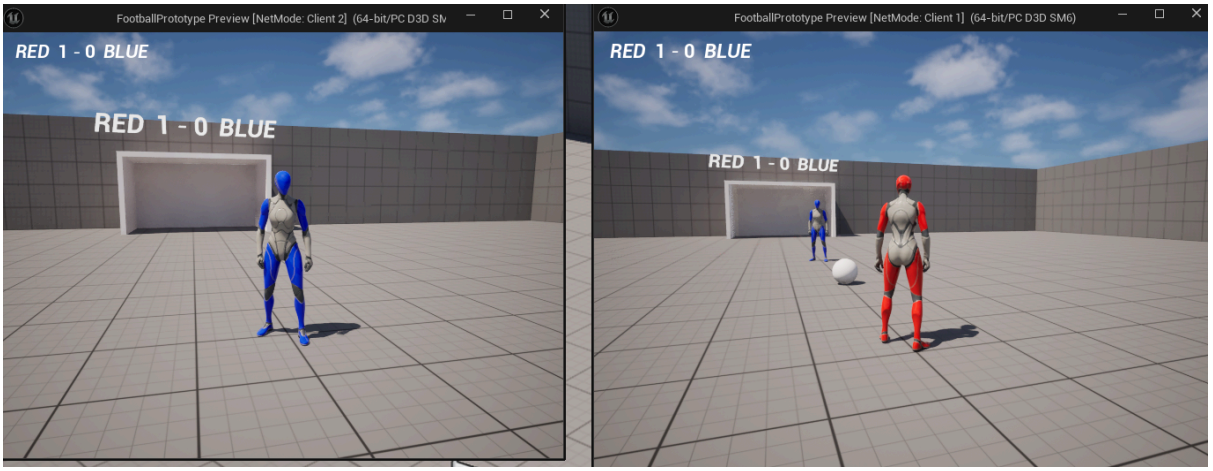


Scoreboard Widget and Replication

The score board widget cooperates with the game stats event. Game stats has each team's score and increments the score when the goal event happens. CallOnScore dispatch is called out when either team's scores are changed and score and goal indicator widgets synchronize the score.
On Score update event sends ETeam enum parameter and only works for updates scored team score to specify which team score must update in the widget.

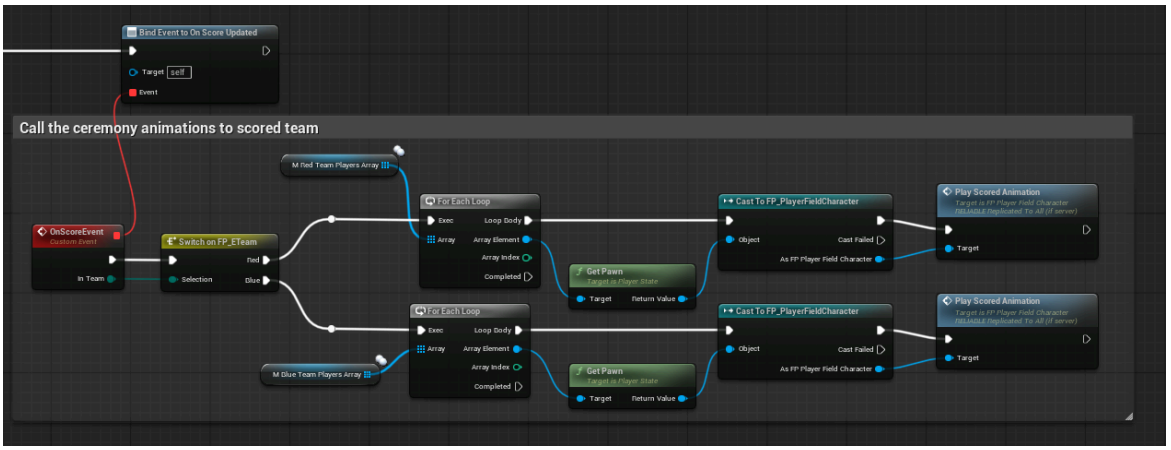


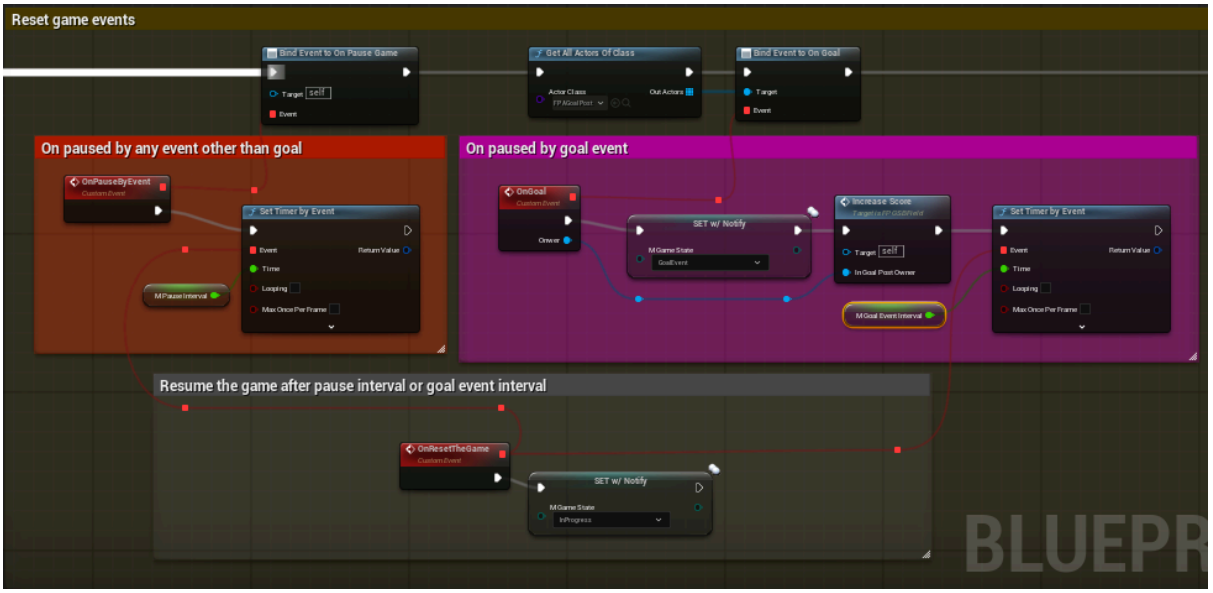
Meanwhile, I made the direct call of the set score function in event construct (on initialization of the game) to make sure the client side player gets the initial score information of each team in the goal post score widget in the world.



Goal Reaction

Everybody wants to celebrate when the player makes a notable goal. GoalPostEvent in the gamestats sends the dispatcher to the goal post widget to show the goal and calls every player which is in the scored team to make the jump. In the goal event status, game is paused and waits for the resetting the pitch. Players and balls reset to the initial position when it resets.



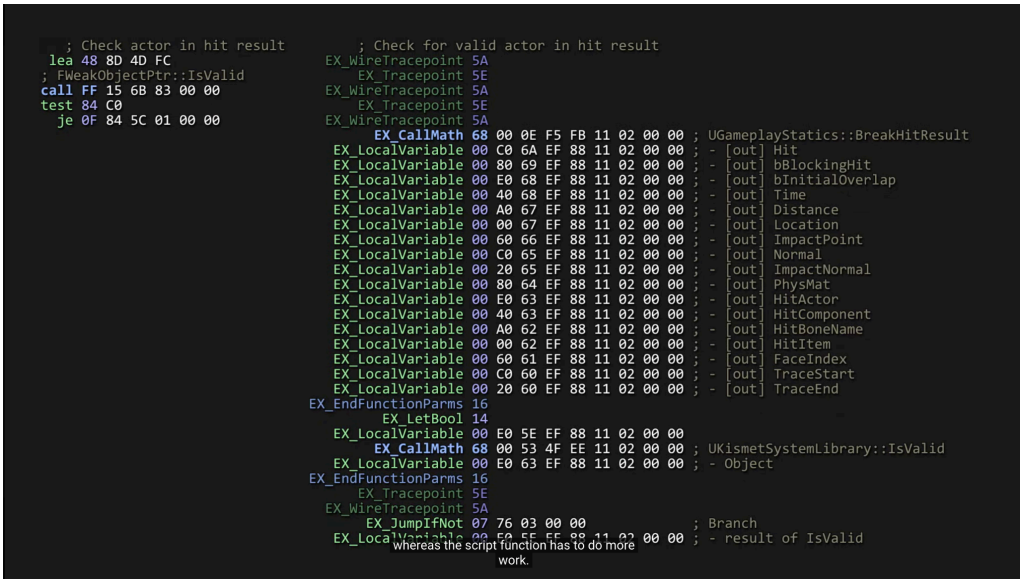


Further Improvements

In the architectural perspective, I'd like to work further to separate the logic with C++ and blueprint as per the level of the scopes and it gives multiple benefits.

In general, C++ performs better than blueprints, as C++ functions and classes have less overhead and drawcalls than blueprints.

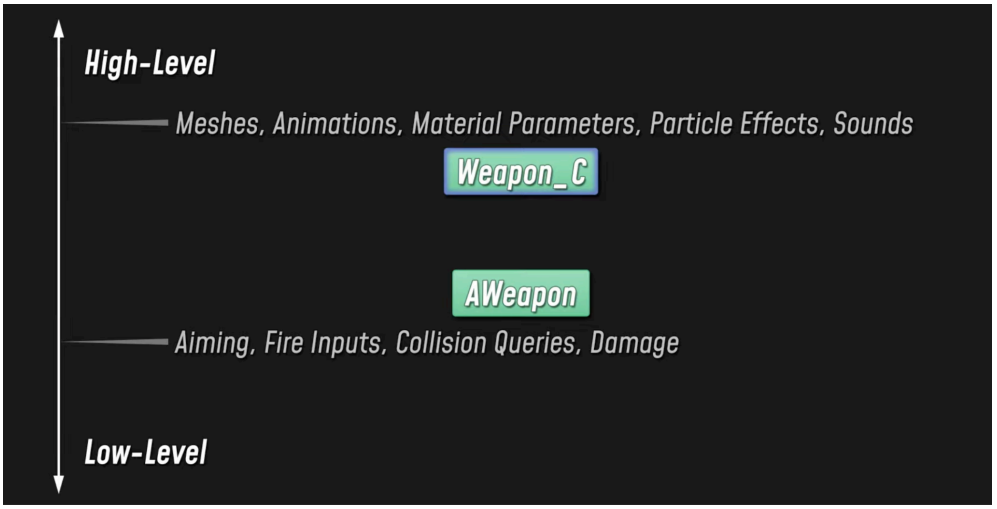
For instance, to verify the hit result of the collision box (or capsule), C++ only calls and test to get the results while the blueprints through the multiple kismet libraries to execute.



Source - Alex Forsythe (2021)

Also, as the ideal structure of the unreal project, it would be best to separate the high level blueprint class and low level core C++ class, especially, cooperative with designers who need a high level approach to iterate.

As the prototype project example, I'd like to keep the player core logic, like shooting and calculation of the movement physics to the C++ which is exclusively for the programmers and doesn't need to be iterations. Otherwise, it is good to keep a few logics or variables for fast prototyping of the game design.



Source - Alex Forsythe (2021)

This hybrid approach gives great flexibility to the designer while the programmers keep working in the core logic in C++ scripts. The designer does most of the work in editor but can keep the core logic and must keep it safe from the unintended iterations.

Otherwise, in the prototype game logic, Game mode could have the room for improvements to give the flexibility of the game mode, which may support the multiple goals, player login controls and further modes other than the small 1v1 football session with the single ball. Player and ball physics are the list of the further improvement list, implementing the real physics mechanism instead of the simplified one.

Also, collision classes can be generalized or make it as the interface if it is possible to deal with multiple collision scenarios to make different results.

References

Alex Forsythe. (2021, March 13). Blueprints vs C++ : How They Fit Together and Why You Should Use Both [Video]. Youtube. https://www.youtube.com/watch?v=VMZftEVDuCE