Magnetism

Matteo Curti
Oliver Chapman
Iasmina Leagan
Omar Latreche
Joshua Feria De La Torrie

REPOSITORY DIRECTORY

Host Domain - https://dev.azure.com/Group3-FinalProject/

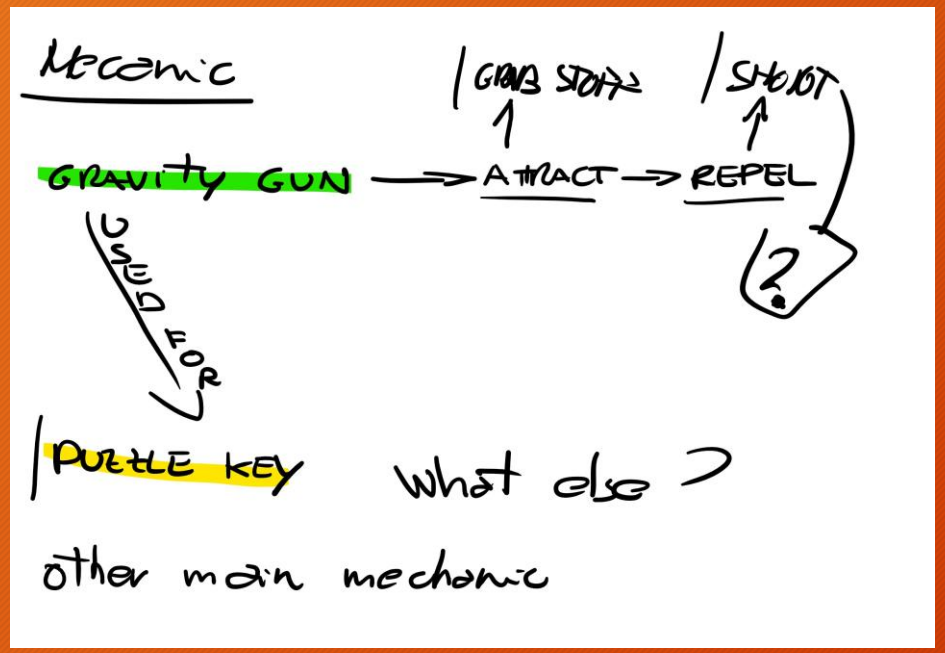
Personal Accesso Token - 7dbfqm2fohj7fvrpfgetdclntlgla7njbwzixj2fp7jy4zlkfqja

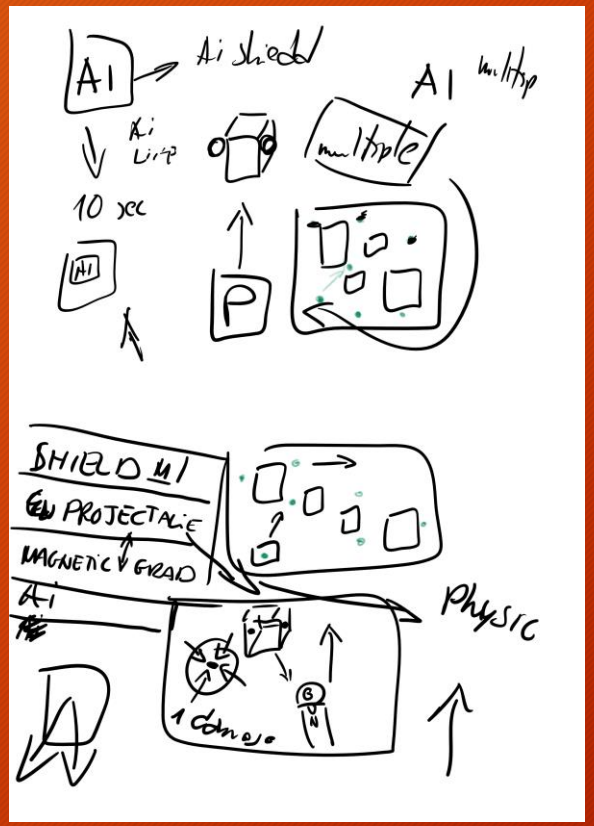
Repository to Clone - MagnetMechanicsTPREPO

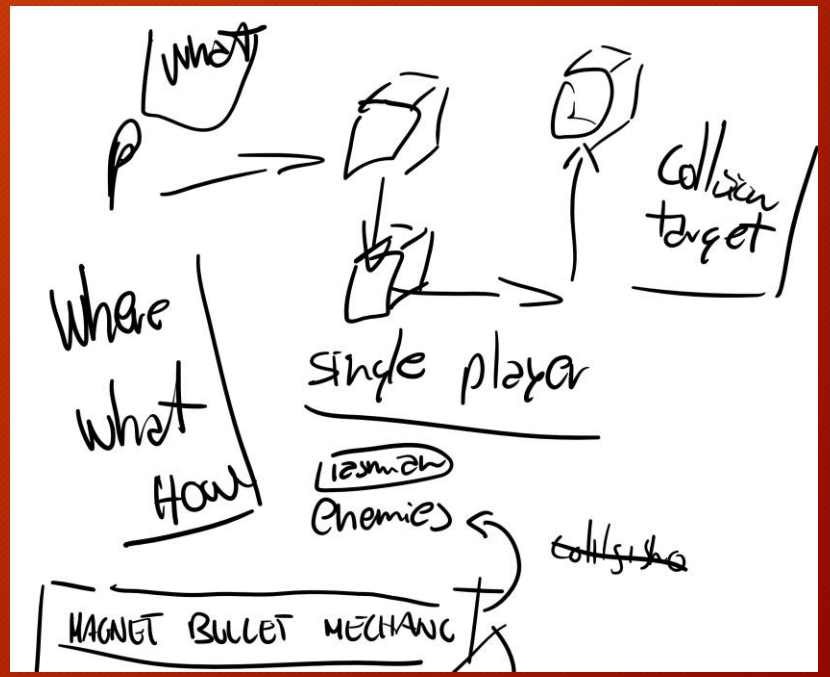
Moodle project uploaded by: Iasmina Leagan

If the token doesn't work you can re-create another one from AzuraDevops

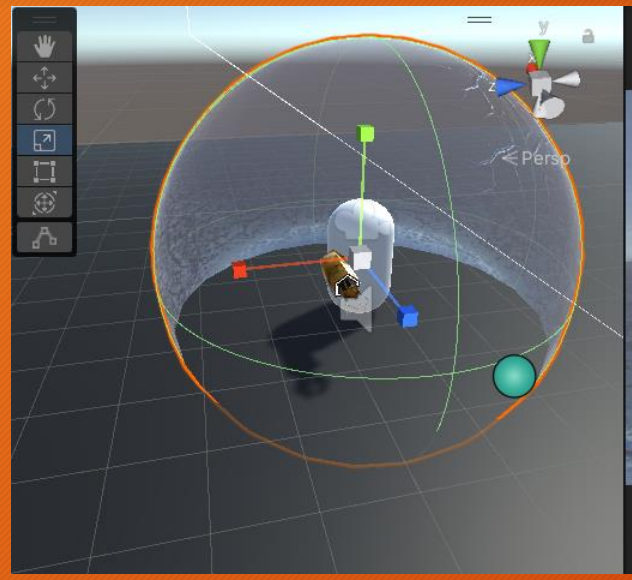
The first idea

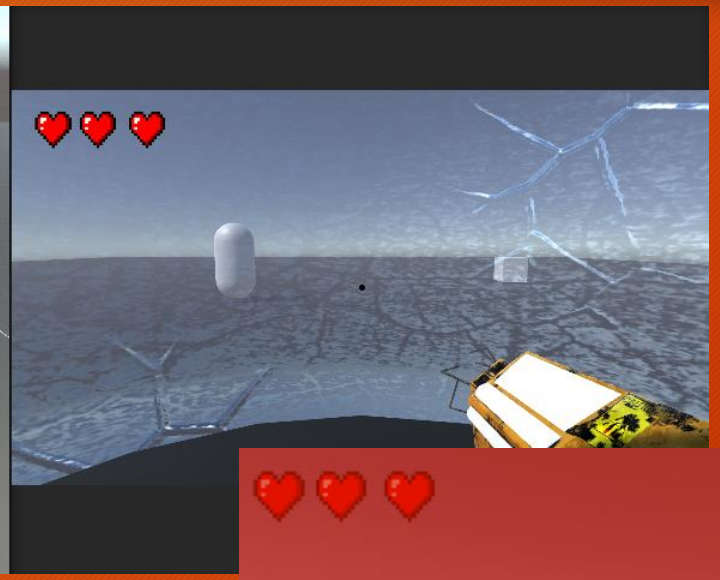


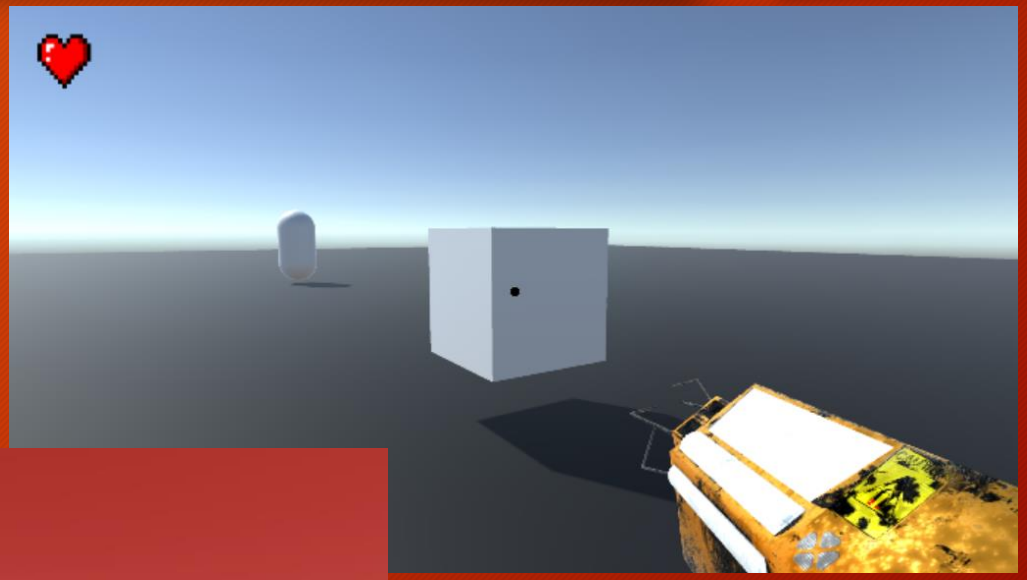


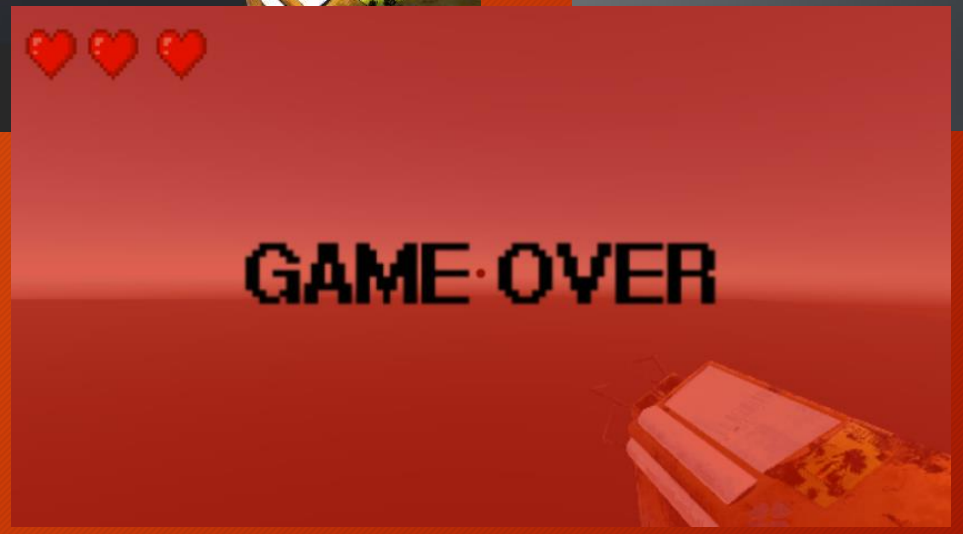


First Development in Unity









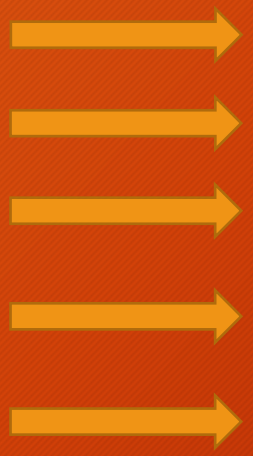
The changes throughout the development

Gameplay Changes

DEMO

The Group Specialism

- Matteo Curti
- Oliver Chapman
- Iasmina Leagan
- Omar Latreche
- Joshua Feria De La Torrie



- Accessibility
- Physics
- Networking
- User Interface
- Optimization

Matteo Curti

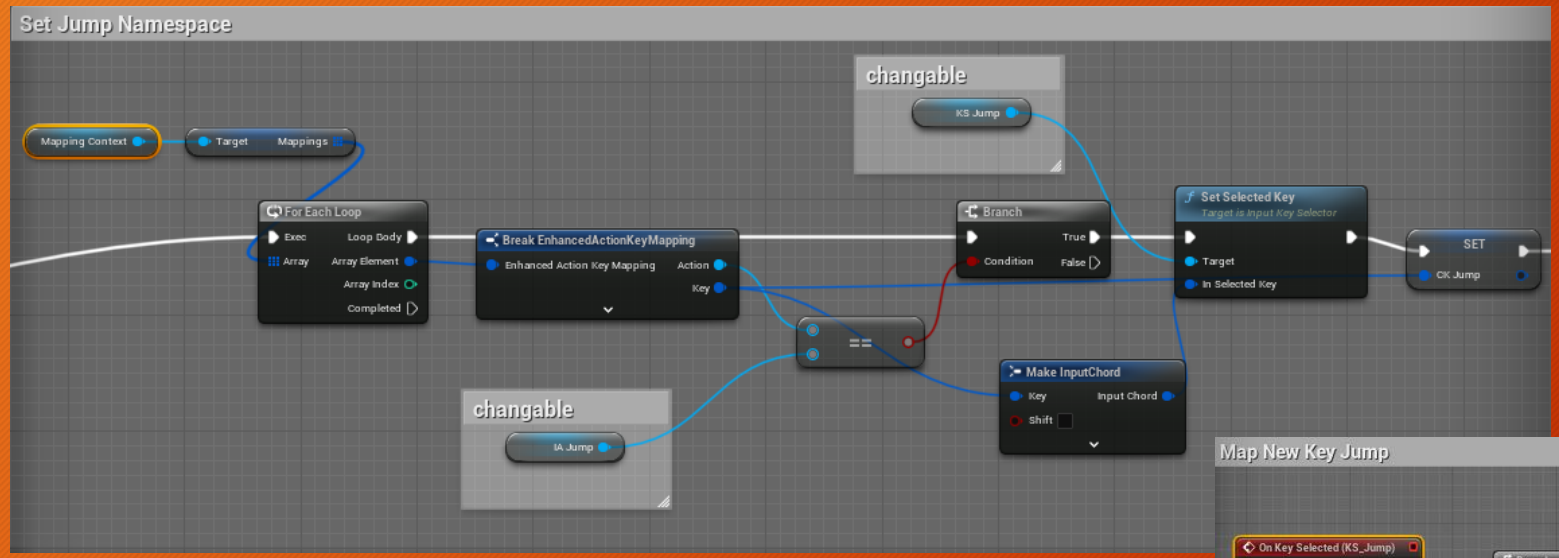
Accessibility

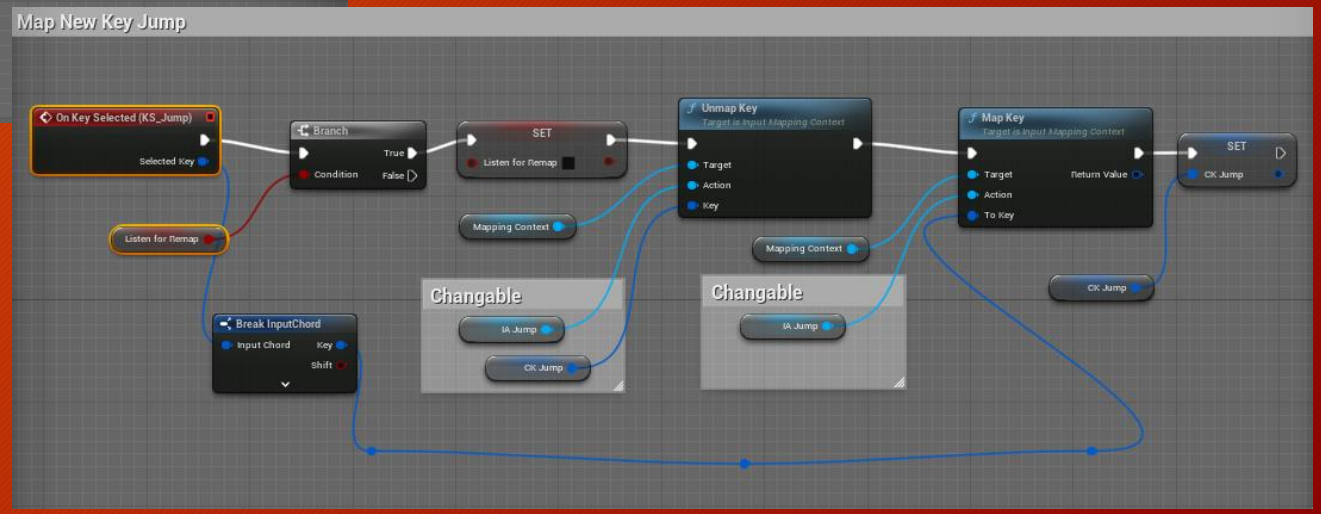
Specialism Implementations

ACCESSIBILITY MENU:

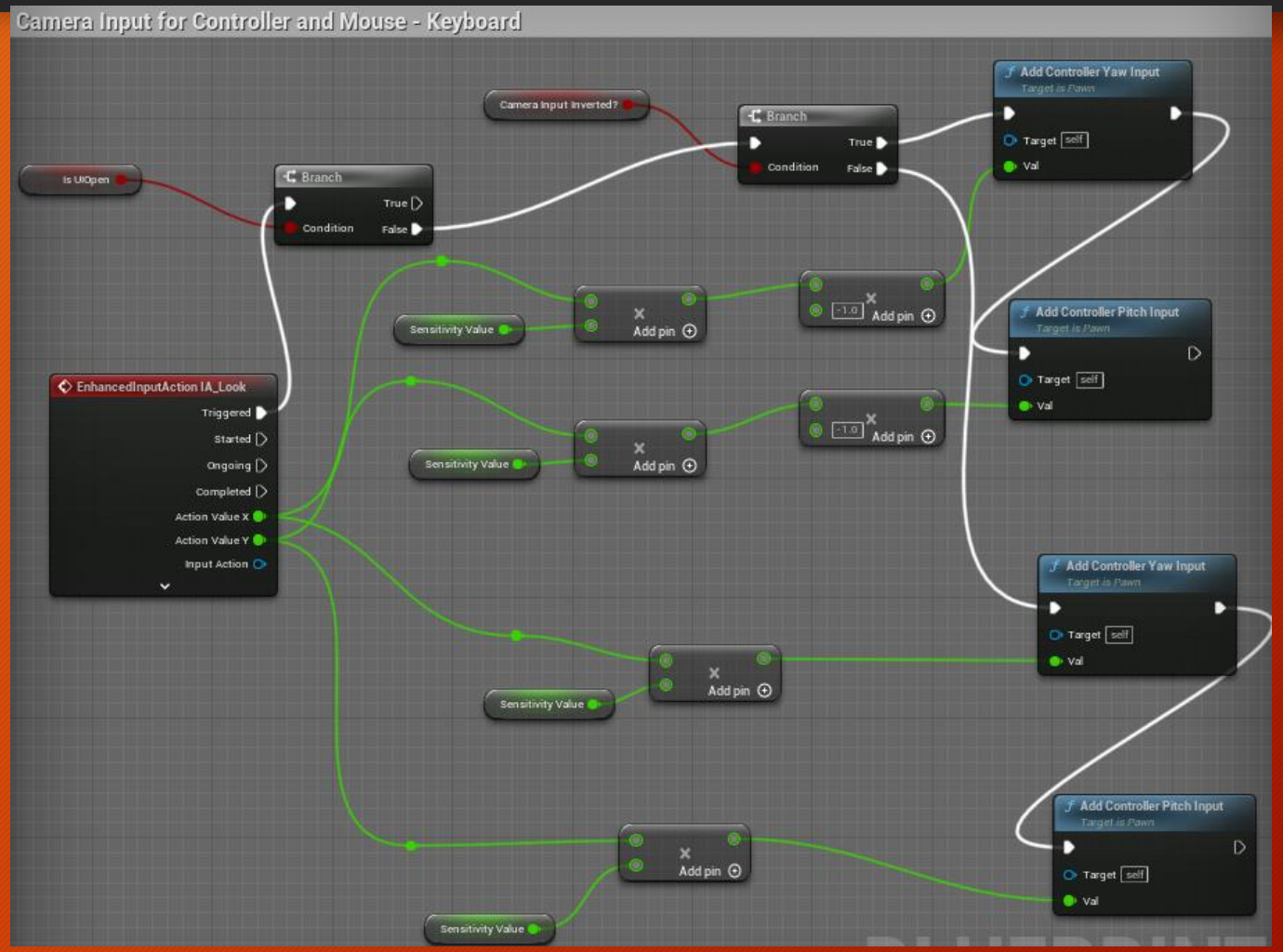
- Re-mappable controls system, with mouse sensibility.
- Invertible camera inputs.
- Colour Blindness functionality with multiple types of variation and the severity level of the colour blindness.
- Save and Load the system for the Accessibility Menu.
- Helped with the scoring system.

Re-mappable Controls

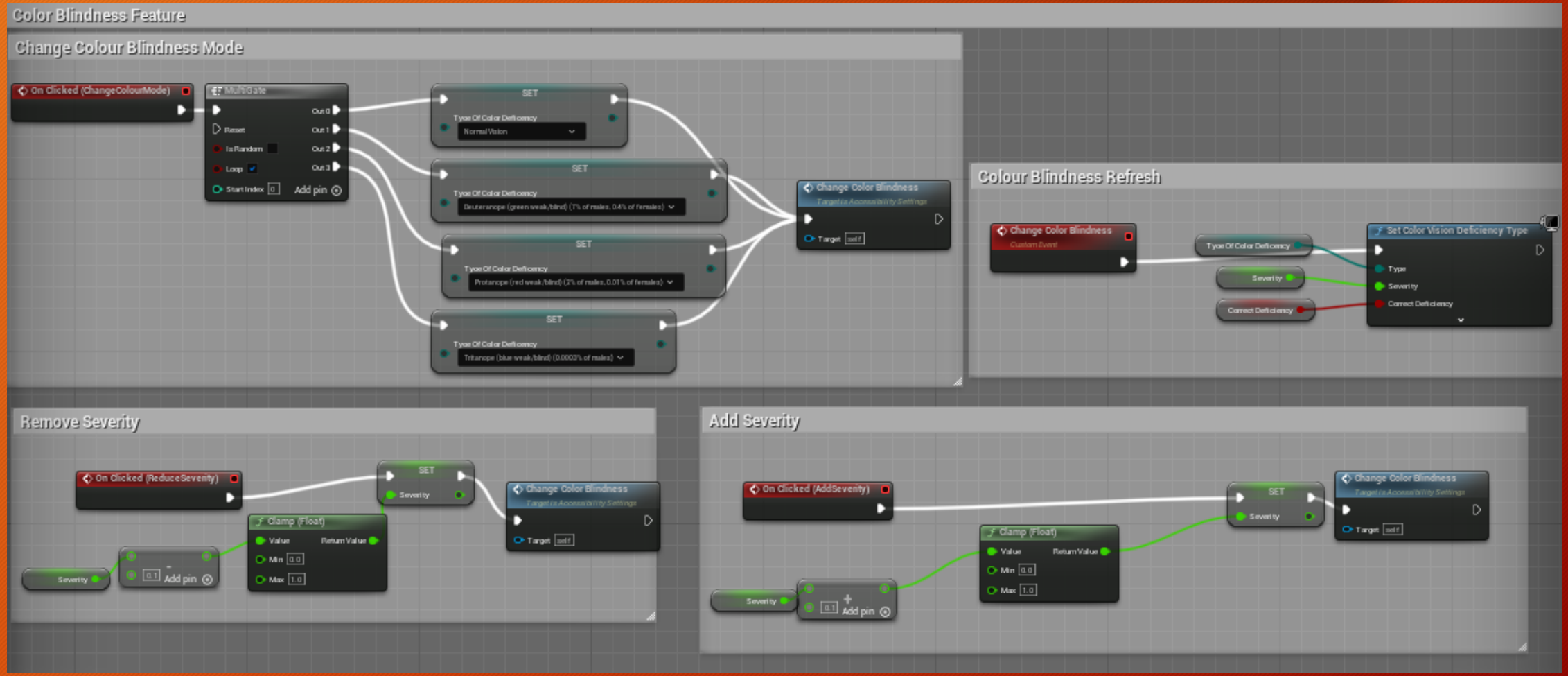




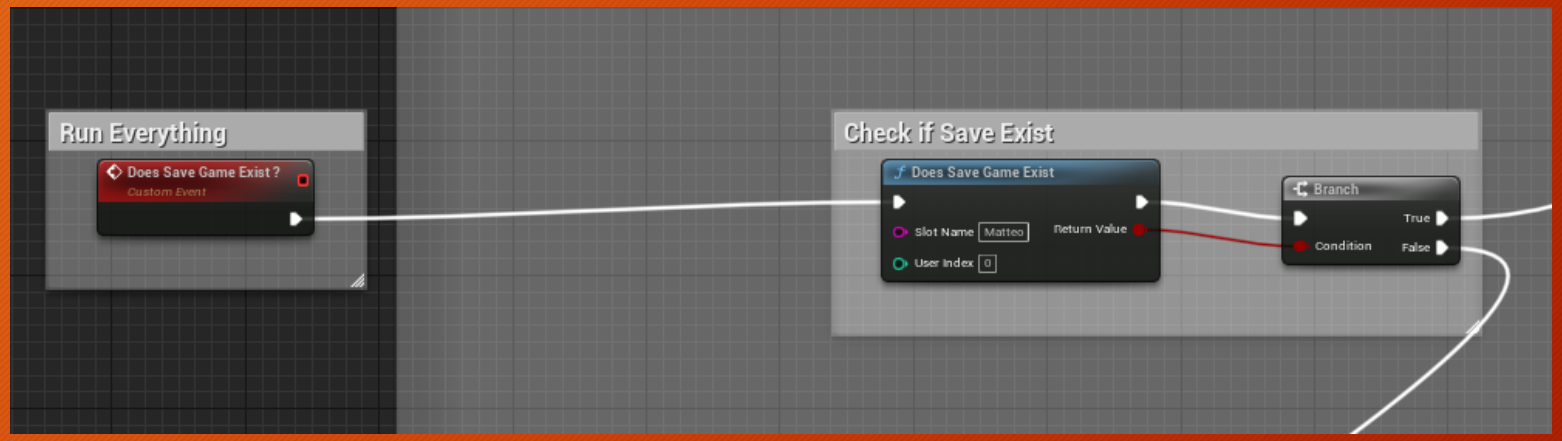
Camera Sensibility and Inverted Camera



Colour Blindness



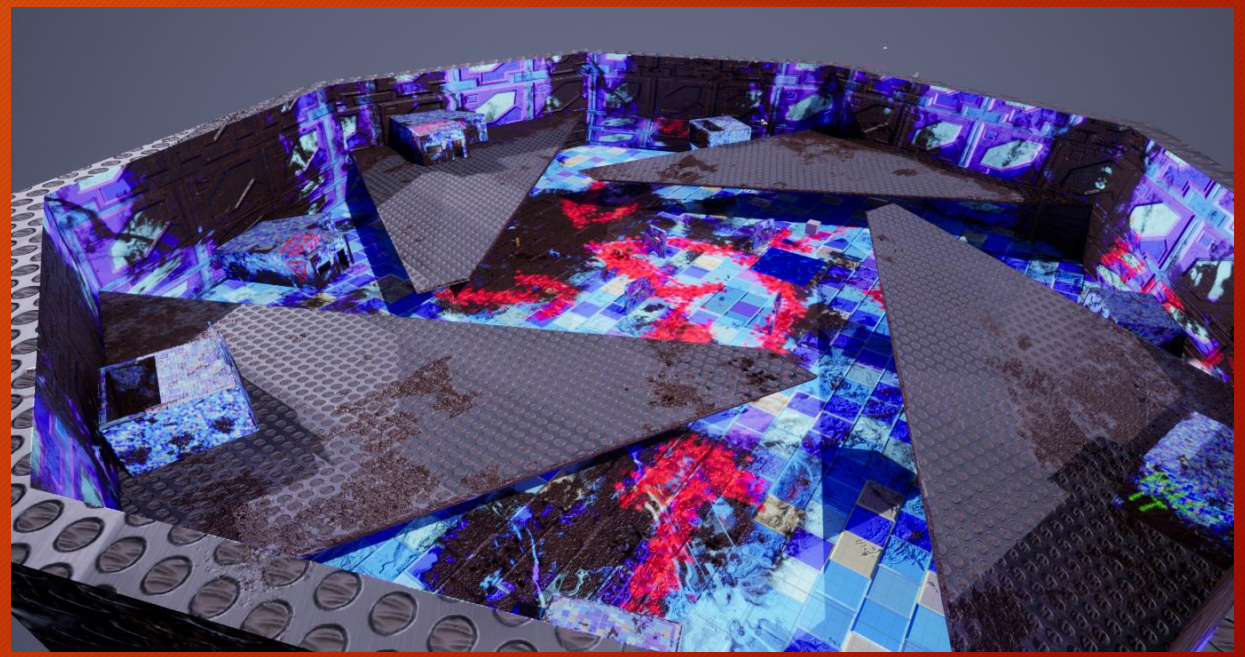
Save and Load System



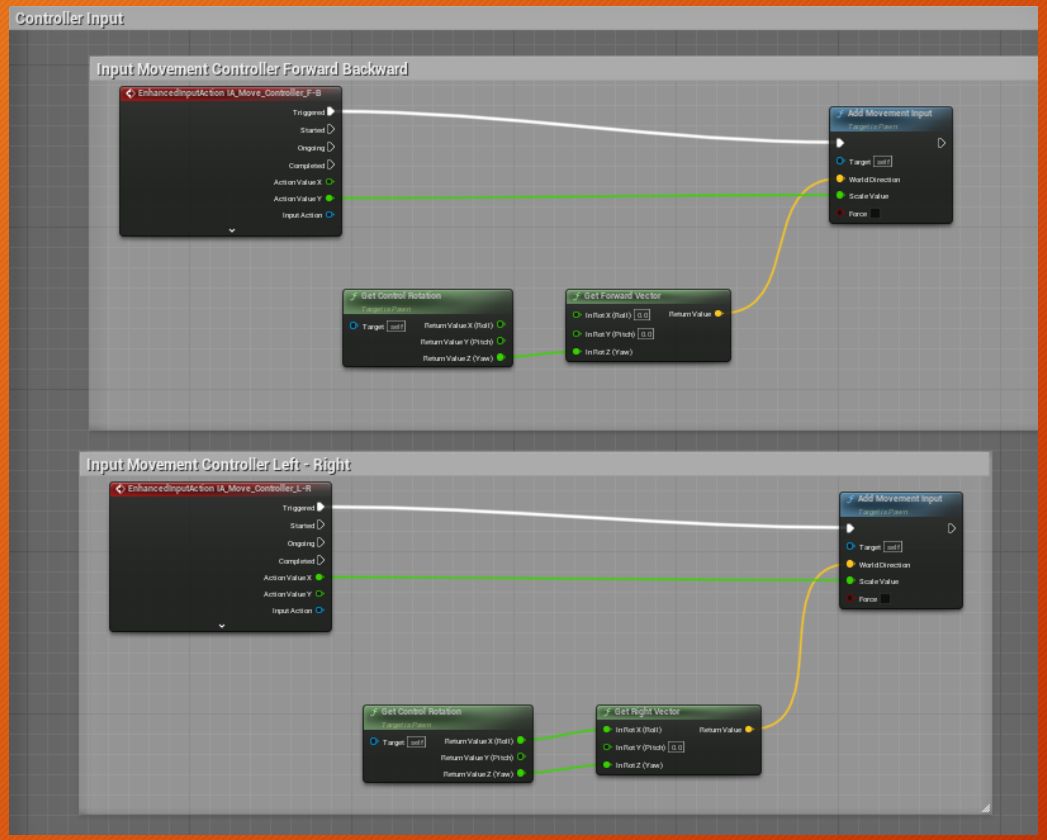
Generalism Implementations

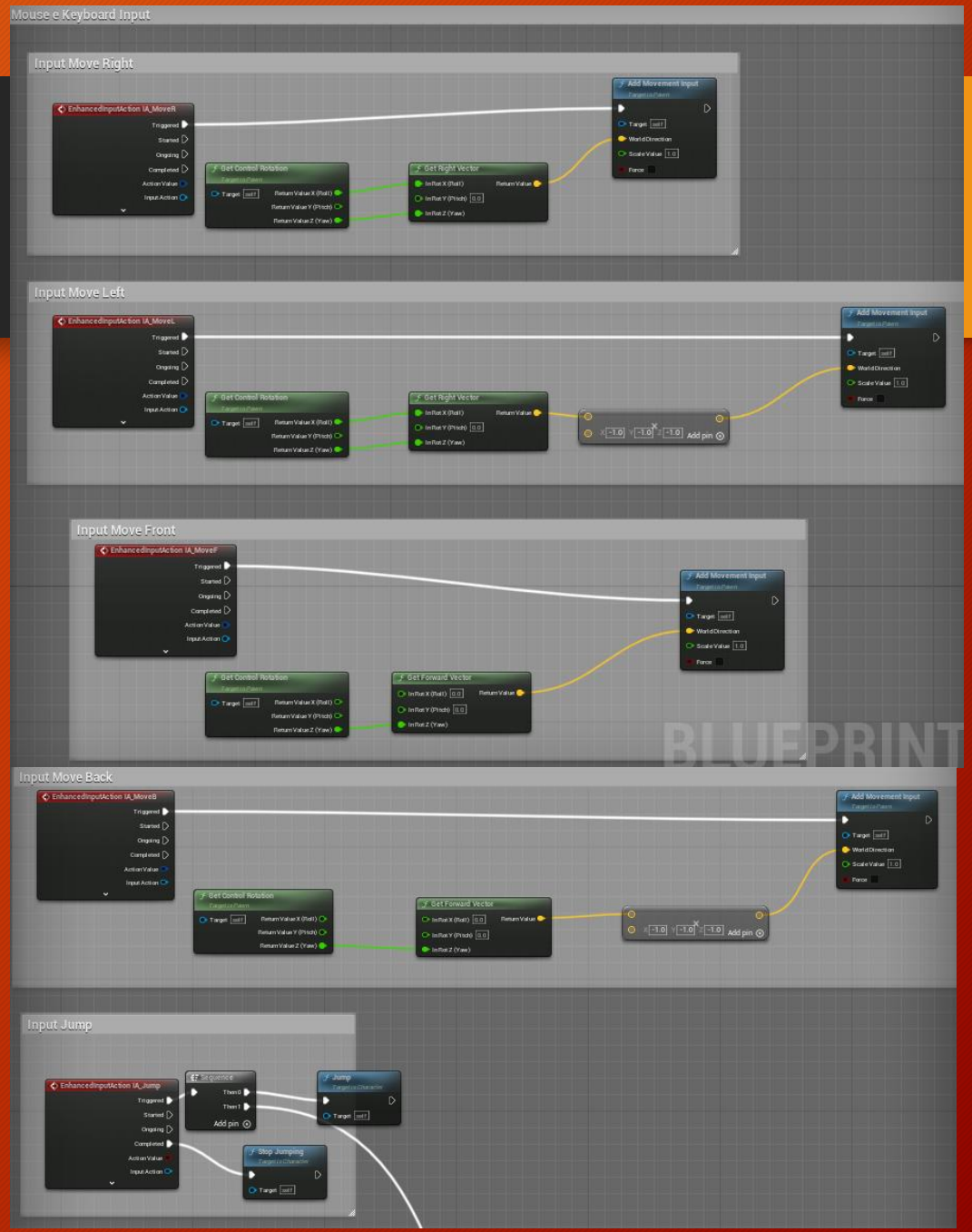
- Modelled and Textured all the assets in the environment.
- Developed the movement script for the character and all the new inputs with Mouse & Keyboard, and Controller.
- Developed a script to give each Player a different Colour when spawned.
- Developed Rotating Planet (Not the Textures).
- Postprocessing (Skybox, Lighting).
- All the character animations.

Environment Modelling and Texturing

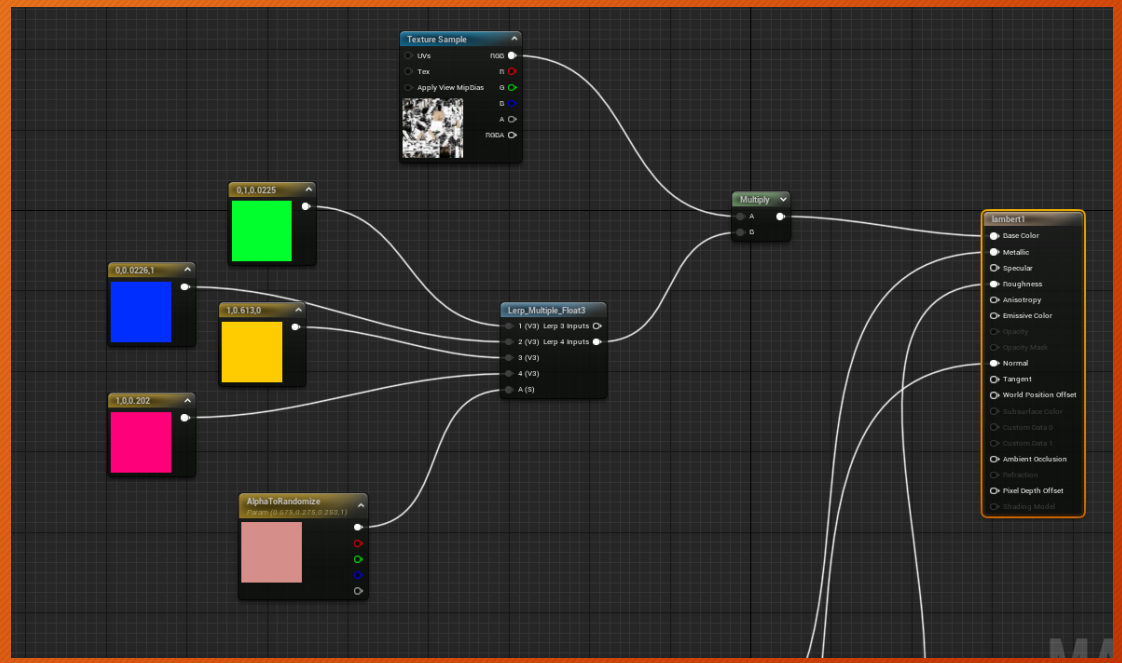


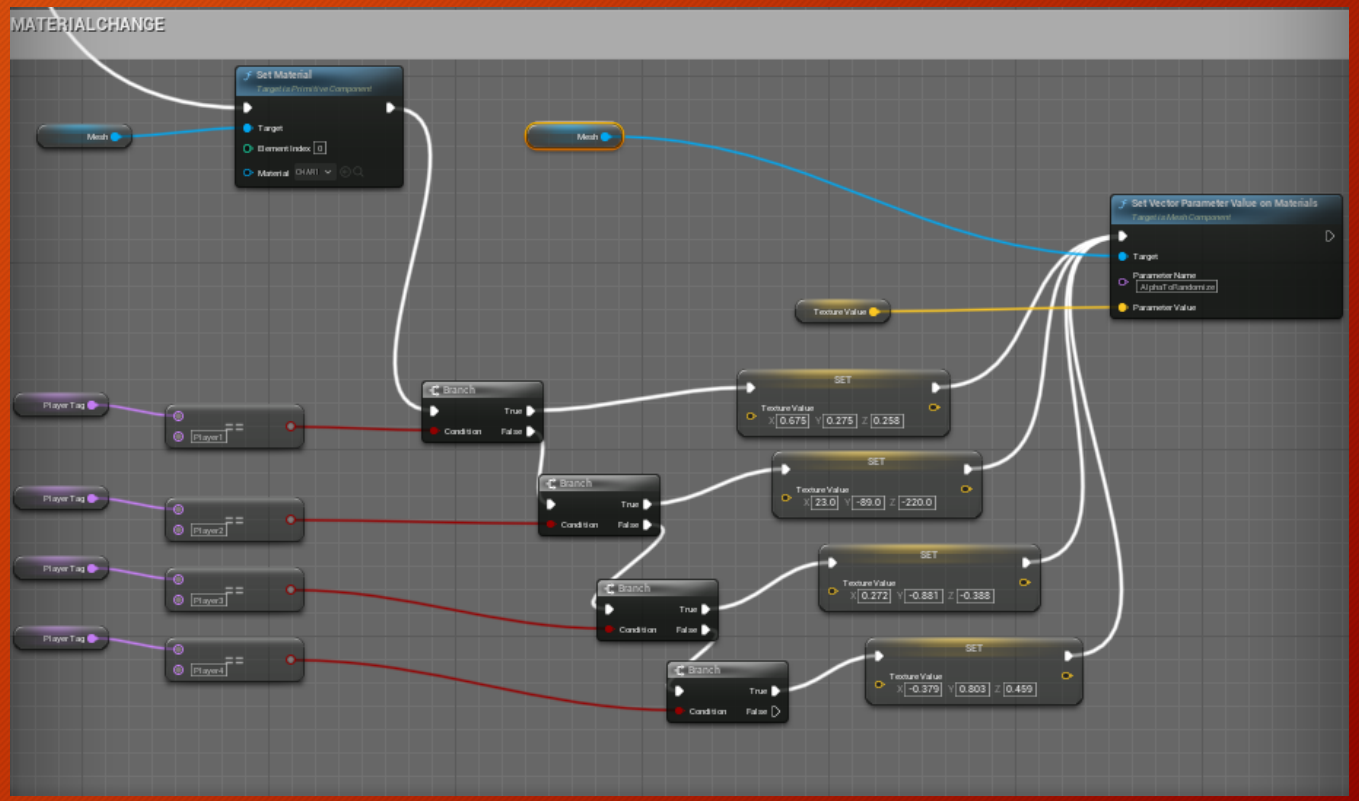
Movement Scripts



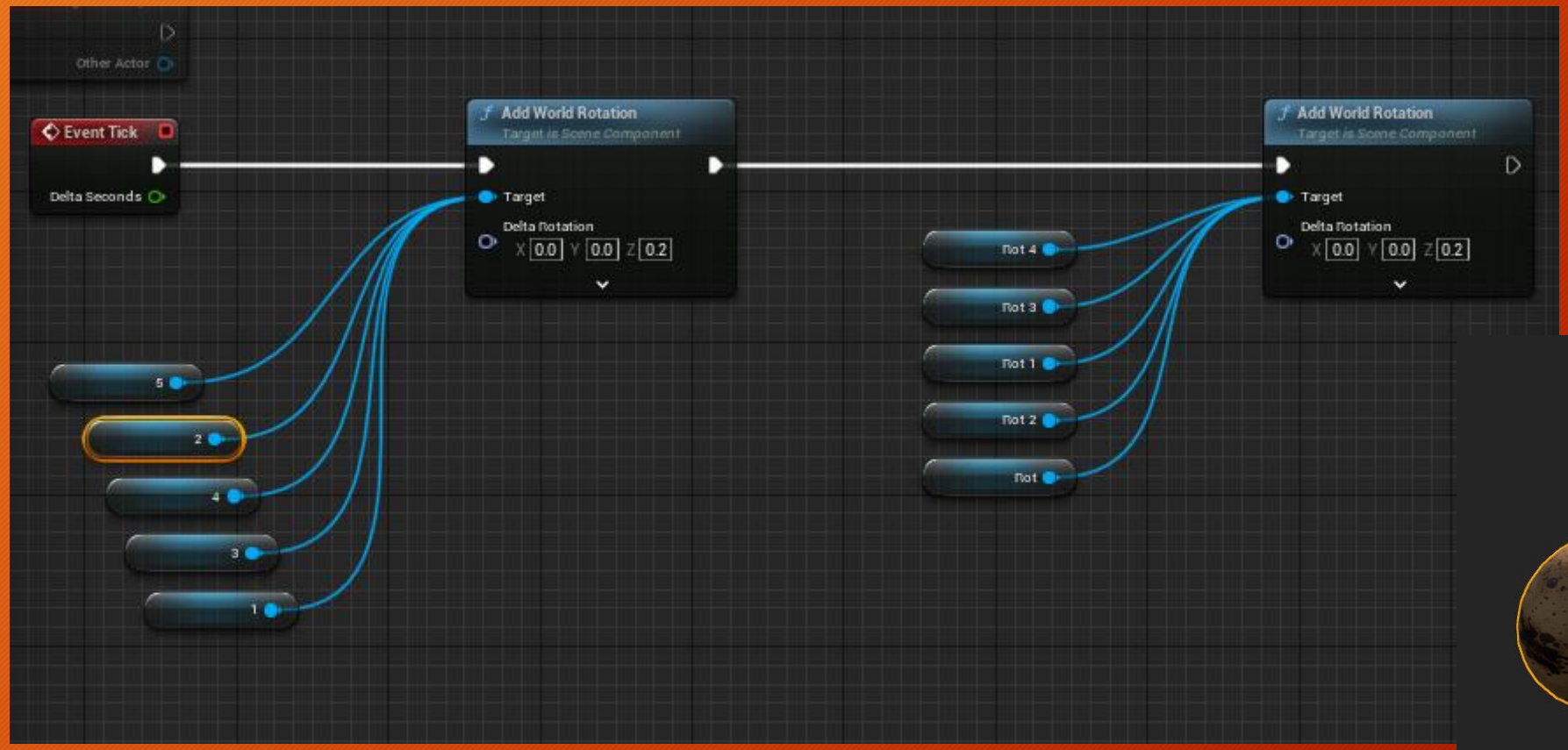


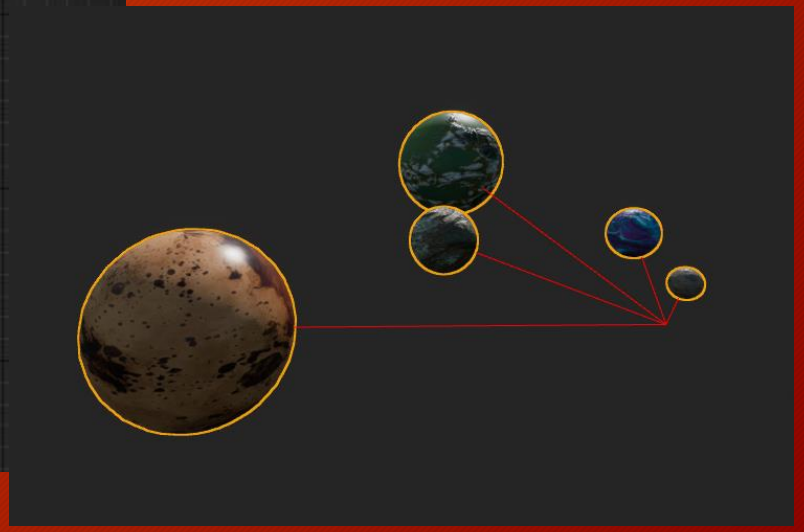
Change Texture Colour depending on the Character





Rotating Planets





Oliver Chapman

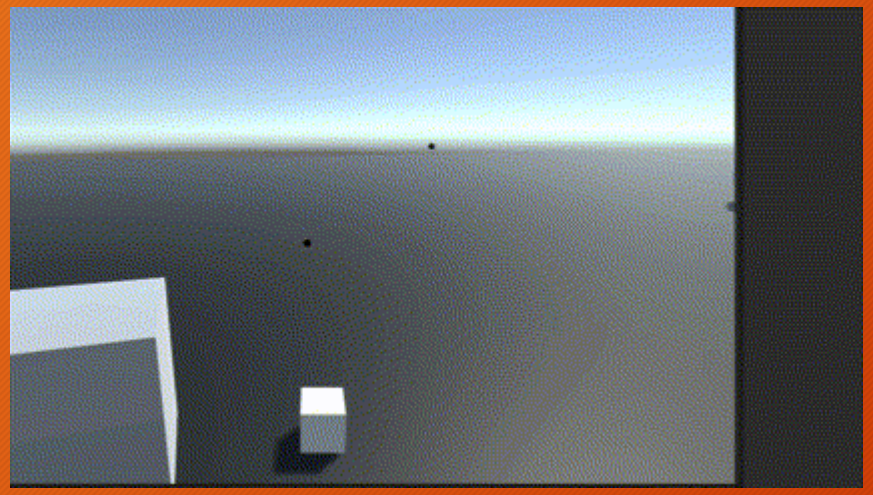
Physics

Specialism Implementations

- **Character Movement**
- Characters with more advanced movement mechanics
- Double Jump
- Wall Running
- **Physics Mechanics**
- Gravity Gun
- Rope Bridge

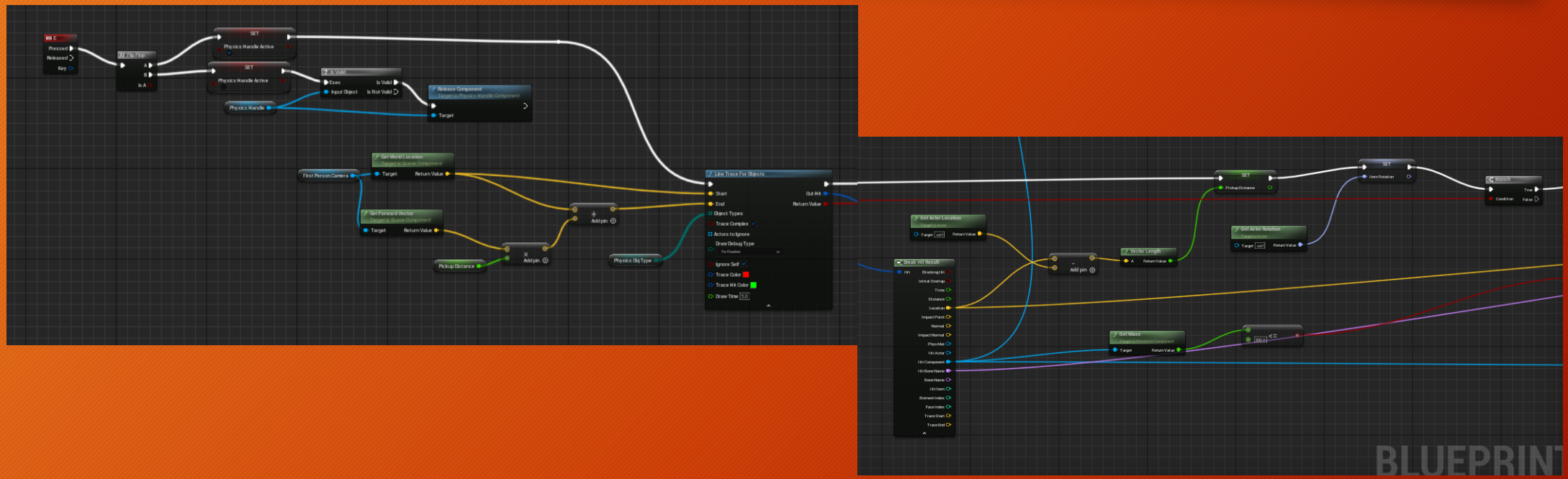
Original Version

- Built in Unity

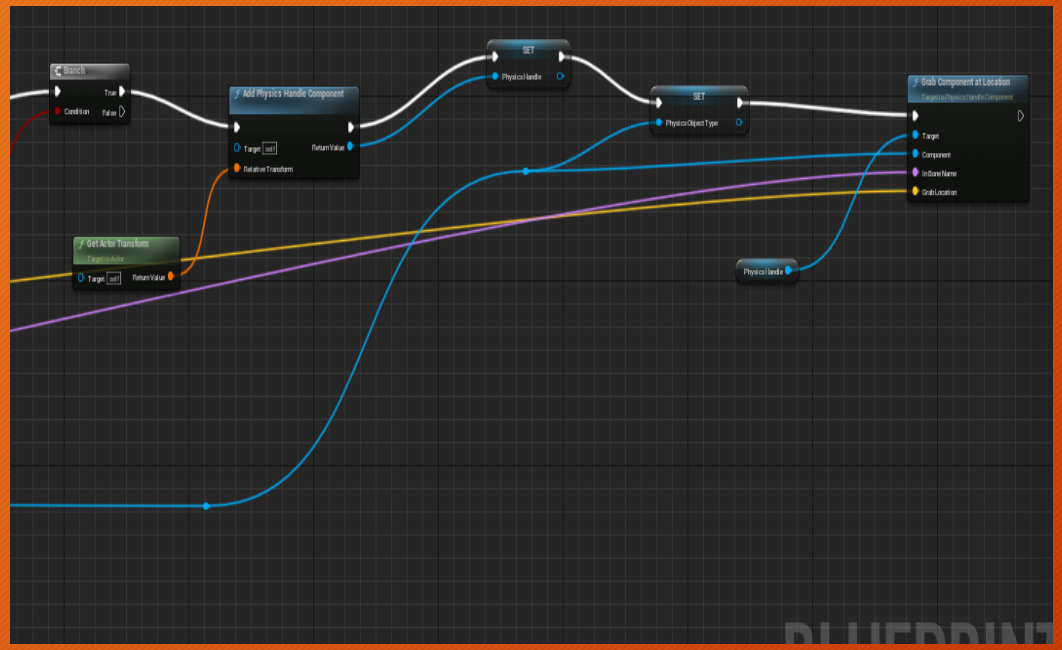


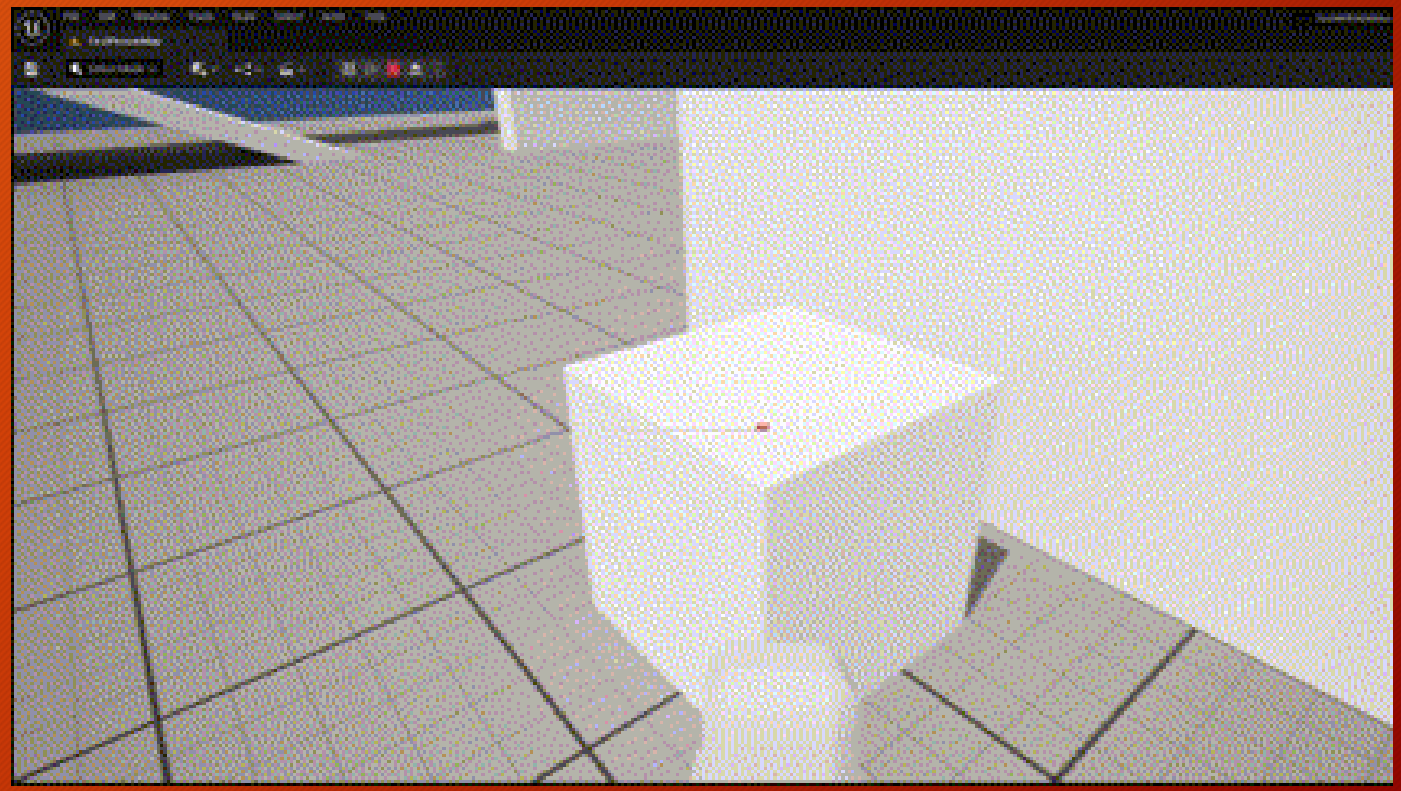
```
Unity Script (1 asset reference) | 0 references
public class GravGun : MonoBehaviour
{
    [SerializeField] private Camera cam;
    [SerializeField] private float maxGrabDistance, throwForce, lerpSpeed;
    [SerializeField] private Transform objectHolder;

    private Rigidbody grabbedRB;

    Unity Message | 0 references
    void Update()
    {
        if (grabbedRB)
        {
            grabbedRB.MovePosition(Vector3.Lerp(grabbedRB.position, objectHolder.transform.position, Time.deltaTime * lerpSpeed));

            if (Input.GetMouseButtonDown(0))
            {
                grabbedRB.isKinematic = false;
                grabbedRB.AddForce(cam.transform.forward * throwForce, ForceMode.VelocityChange);
                grabbedRB = null;
            }
        }

        if (Input.GetKeyDown(KeyCode.E))
        {
            if (grabbedRB)
            {
                grabbedRB.isKinematic = false;
                grabbedRB = null;
            }
            else
            {
                RaycastHit hit;
                Ray ray = cam.ViewportPointToRay(new Vector3(0.5f, 0.5f));
                if (Physics.Raycast(ray, out hit, maxGrabDistance))
                {
                    if (hit.collider.gameObject.tag == "Grabbable")
                    {
                        grabbedRB = hit.collider.gameObject.GetComponent<Rigidbody>();
                    }
                    if (grabbedRB)
                    {
                        grabbedRB.isKinematic = true;
                    }
                }
            }
        }
    }
}
```

Initial Development – Gravity Gun V2



Initial Development – Gravity Gun V2

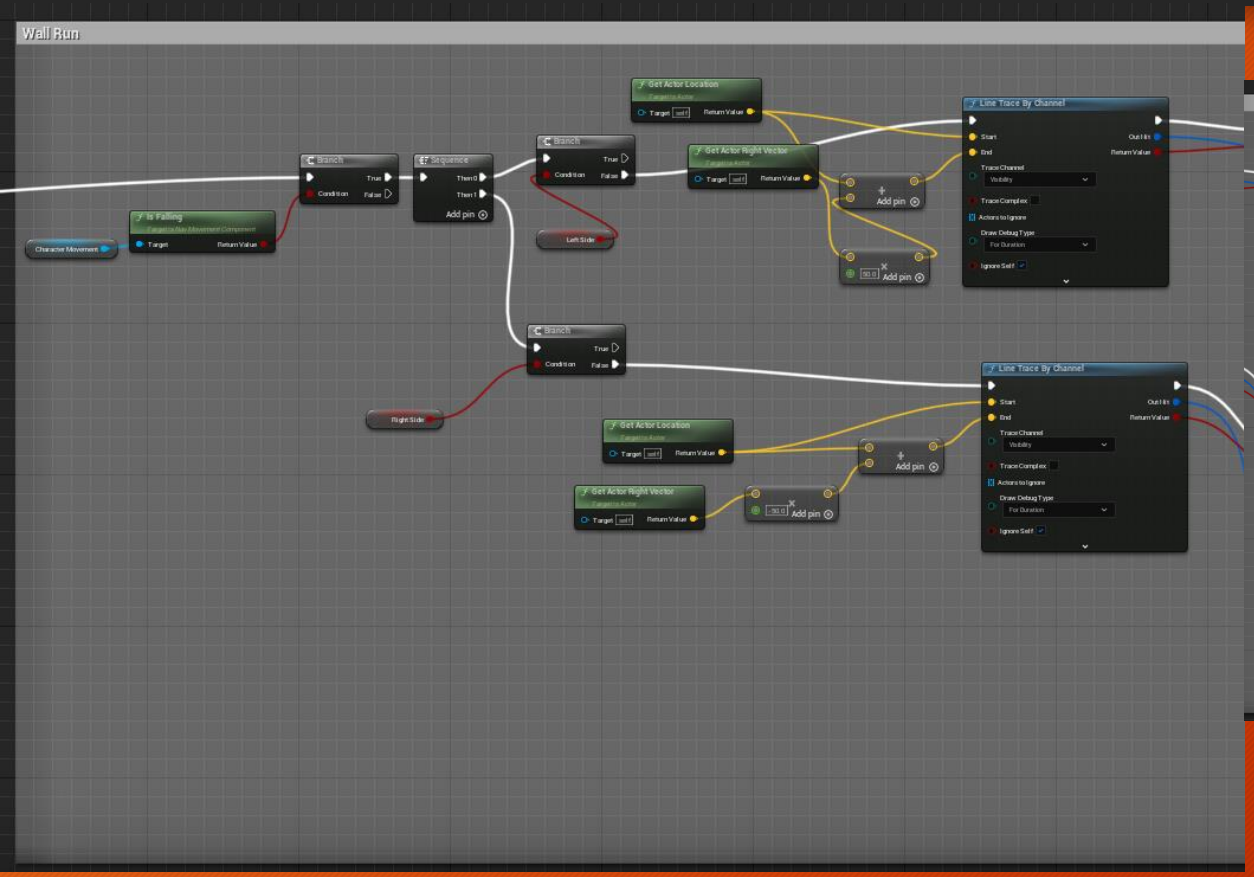


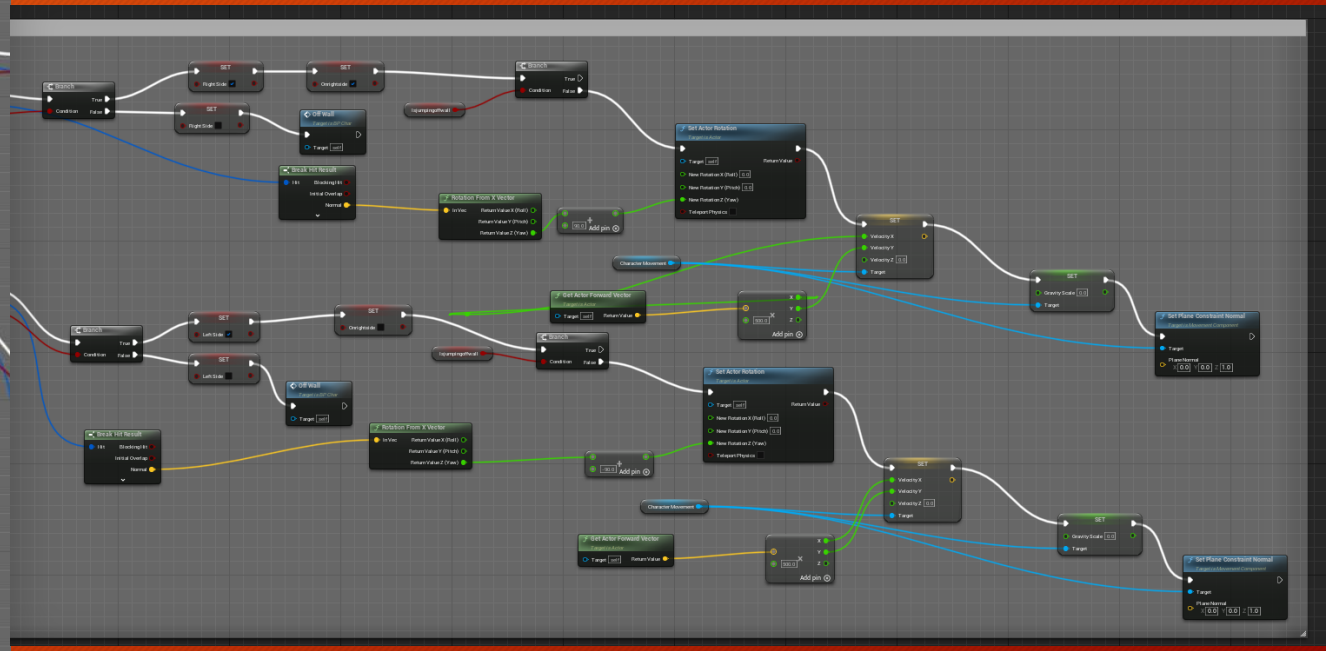


Character Movement Key Points

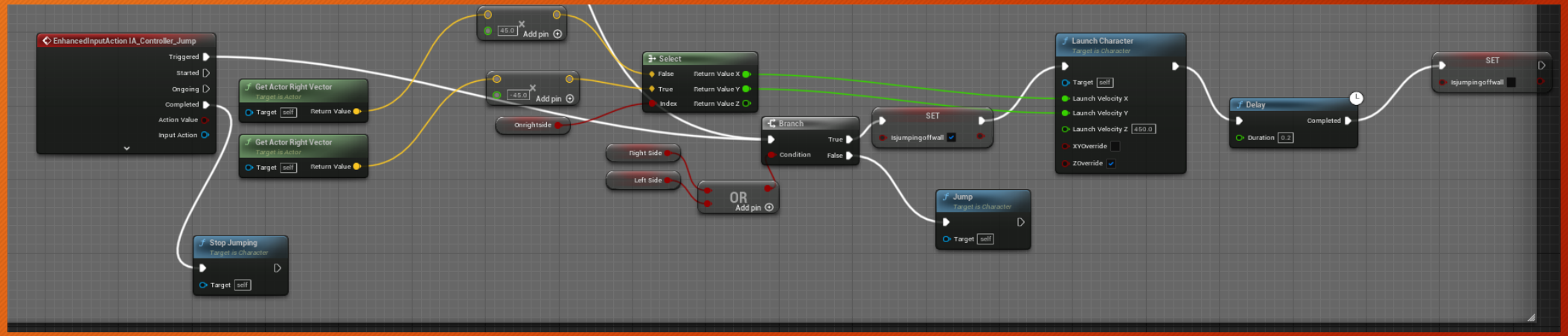
- Advanced character movement can make players feel more connected to their in-game avatar, allowing them to inhabit the game world. Fluid movement can help create a sense of realism and can make players feel as though they are truly controlling the character.
- Advanced movement mechanics can open up new avenues for gameplay, such as allowing players to explore previously unreachable areas or to access hidden items or secrets.
- In multiplayer games players who have mastered advanced movement mechanics can gain an edge over their opponents, allowing them to out maneuverer and outplay them.

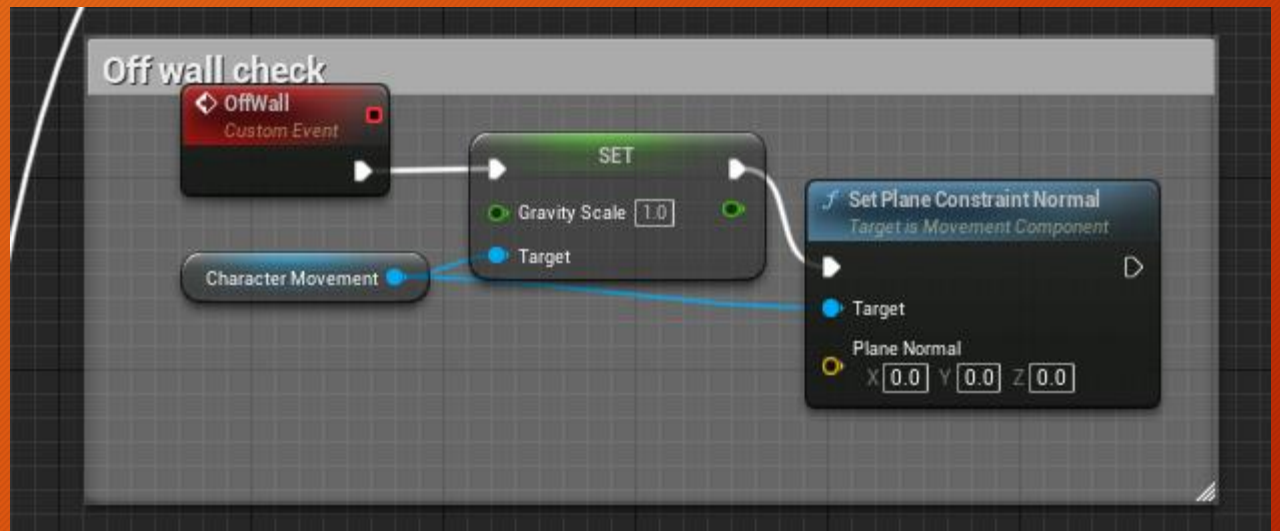
Wall Running





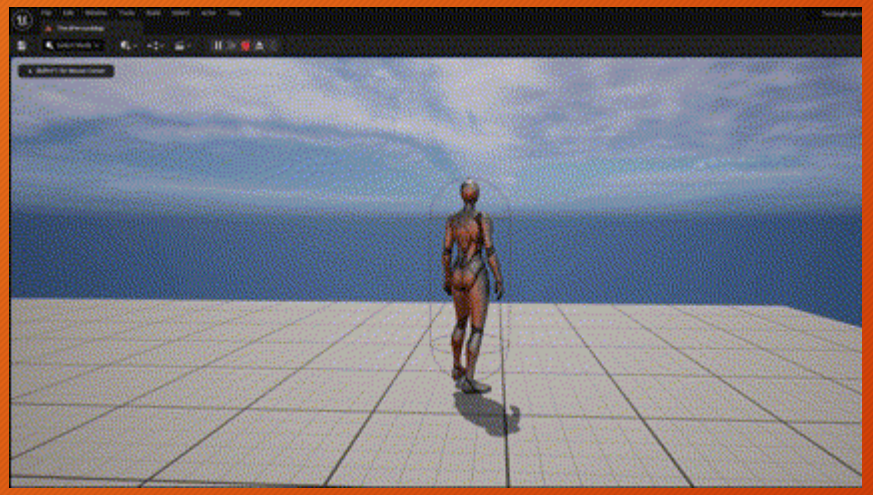
Wall Running – Jump off Wall

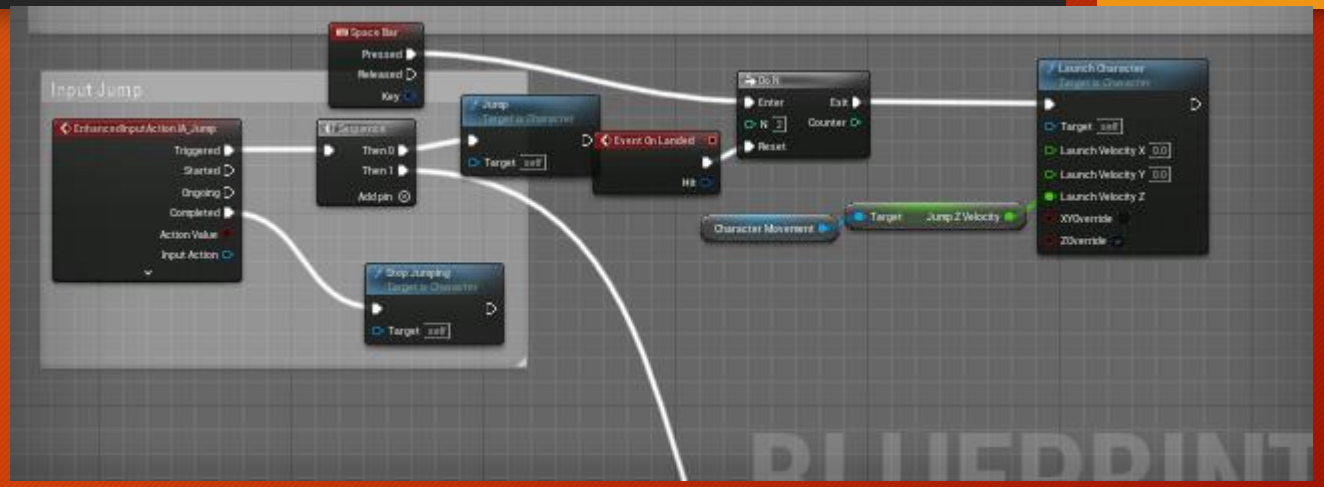


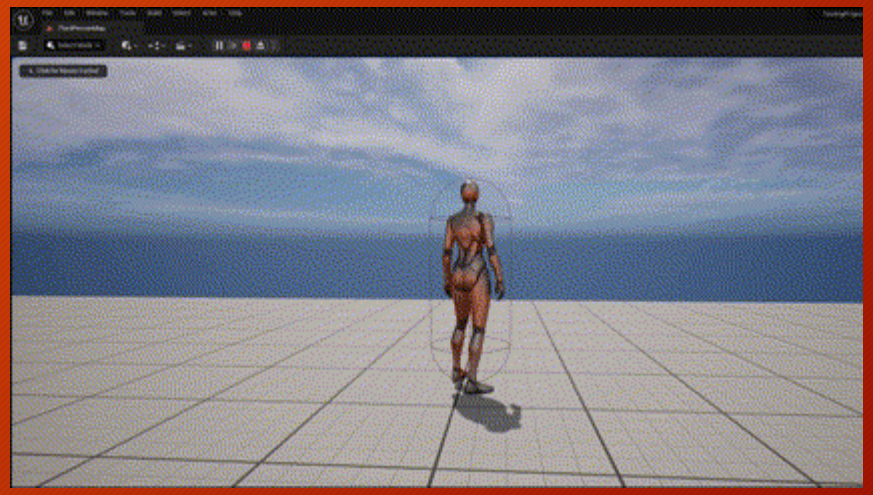


Advanced Character Movement

- Double Jump / Crouching





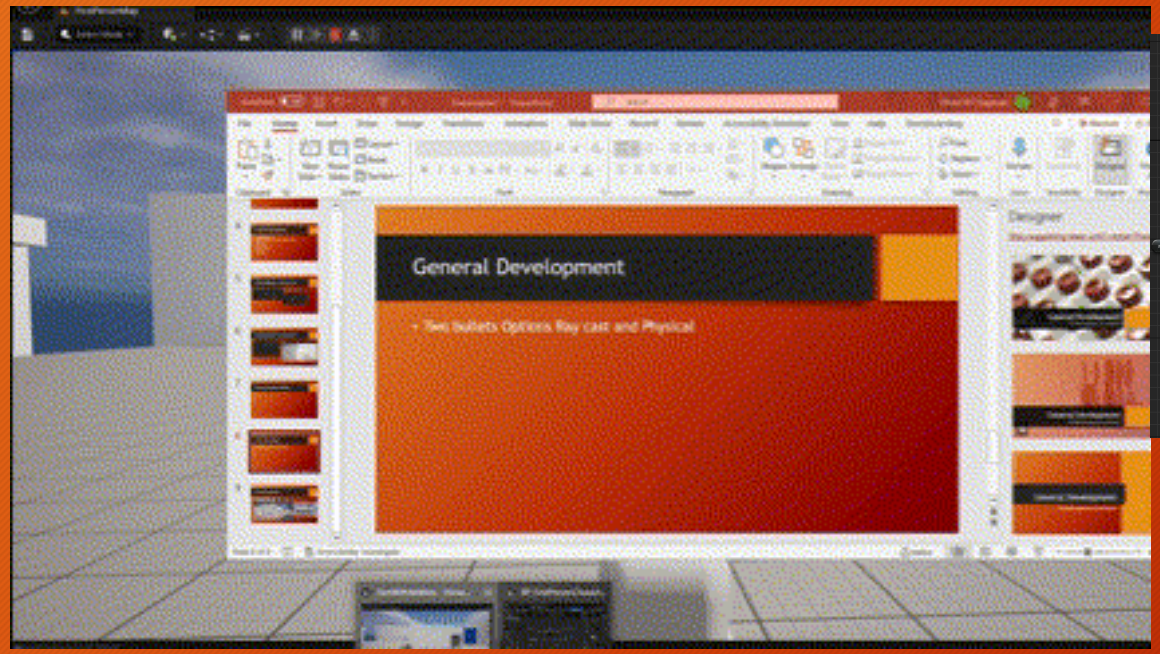


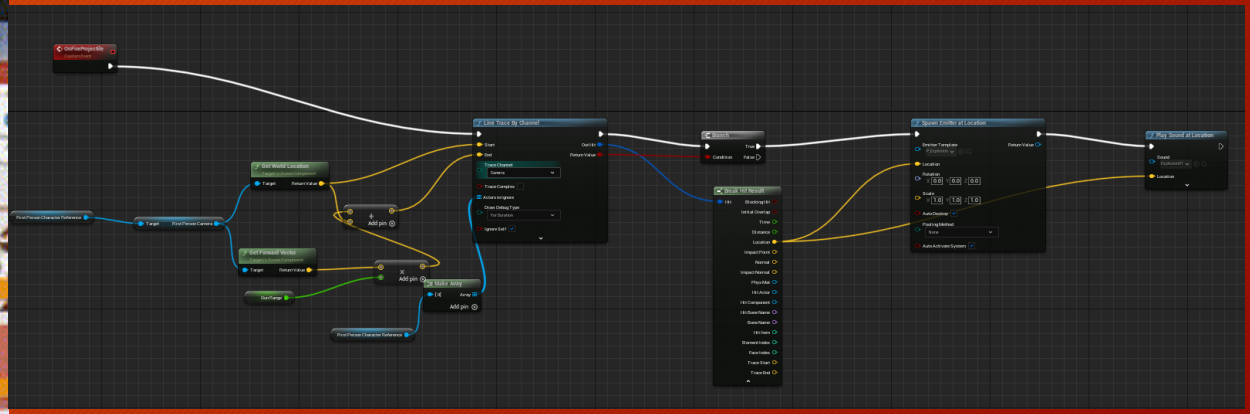
Physics Testing

Testing was carried out on a rope bridge and a jump pad but after the changes to the game had been made the development of these stopped

General Development

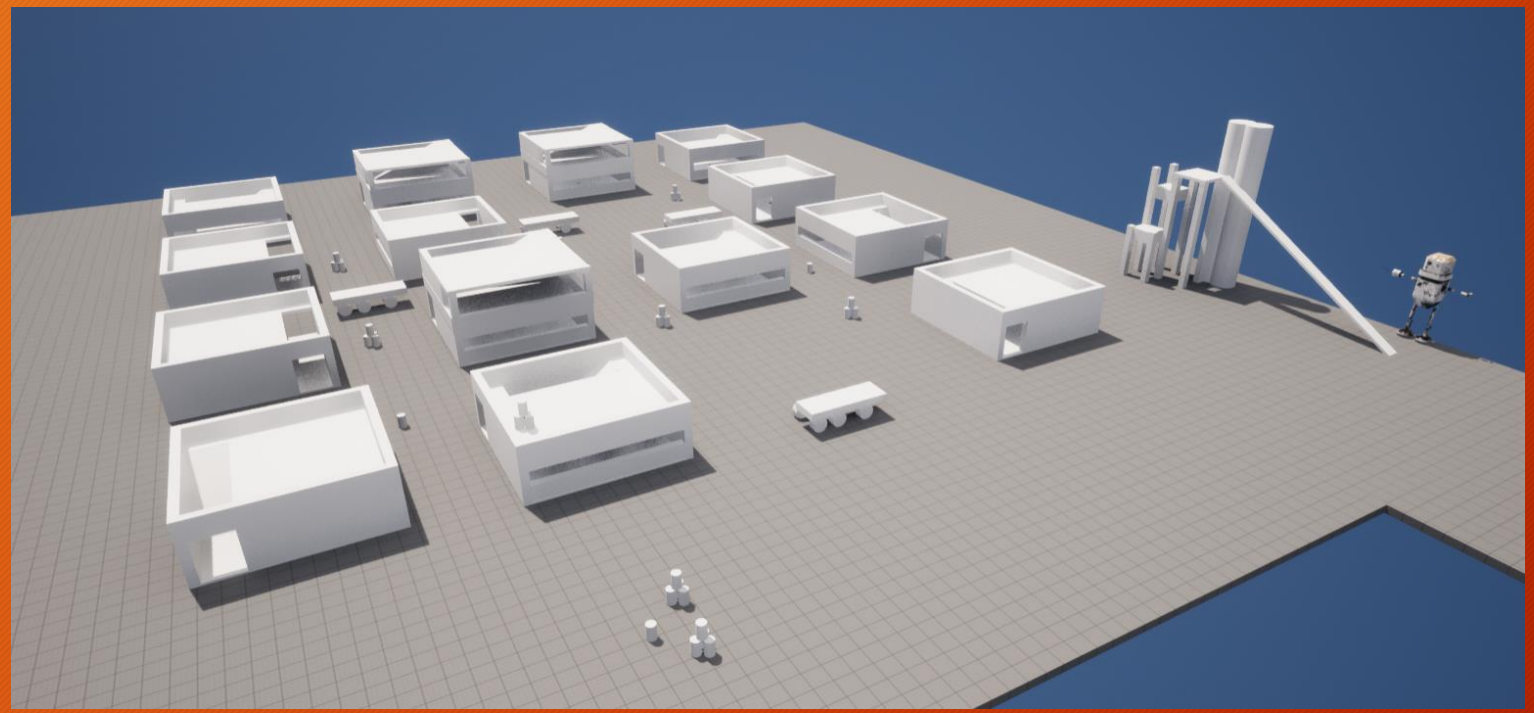
- Two bullets Options Ray cast and Physical

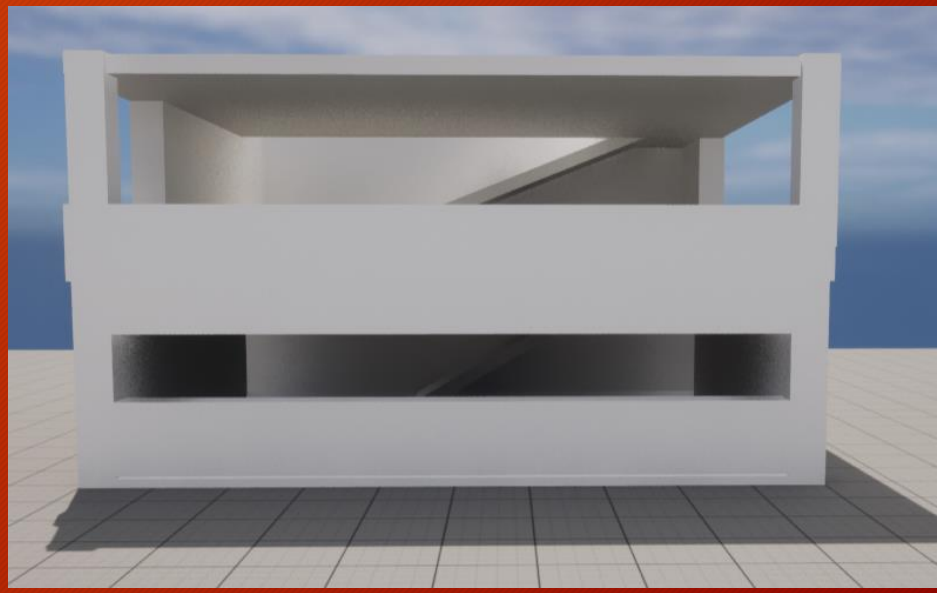




General Development

- Early Map Creation Ideas





Omar Latreche

User Interface

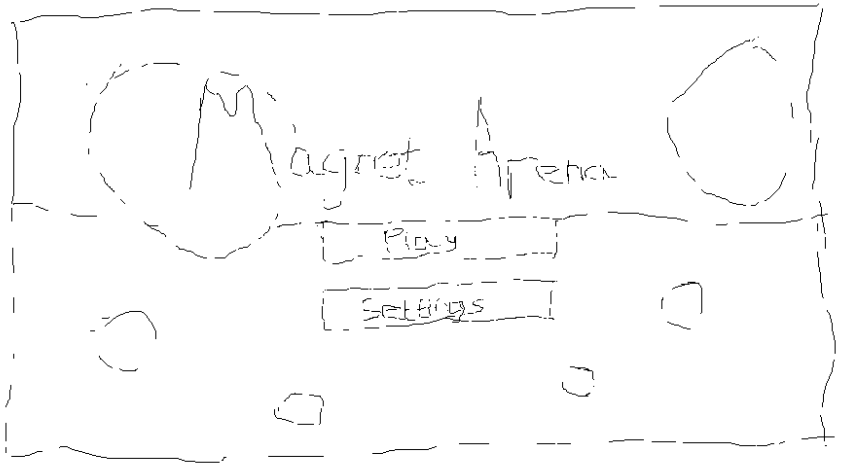
UI Specialism - Design Principles

Clarity and simplicity: Ensure that the UI is easy to understand, with minimal cognitive load for players. Use clear and legible fonts, icons, and colours to convey important information.

Consistency and feedback: Maintain a consistent visual language throughout the UI. Provide real-time feedback to the player's actions to enhance the gaming experience.

Flexibility and efficiency: Design the UI to adapt to various screen resolutions and input devices. Optimize the UI to reduce performance impact on the game.

Aesthetically appealing: Create a visually pleasing UI that complements the game's overall art style and theme.



[Image: Sketch of initial UI design Idea]

UI Components - Game HUD

Health and energy bars: Display the player's health and energy levels prominently on the screen, allowing them to monitor their status easily during gameplay.

Magnetic power indicators: Show the strength and direction of the magnetic fields generated by the player or the environment, providing crucial information for solving puzzles and overcoming challenges.

Interactive map and objectives: Present an in-game map with clearly marked objectives and points of interest, helping the player to navigate the game world.

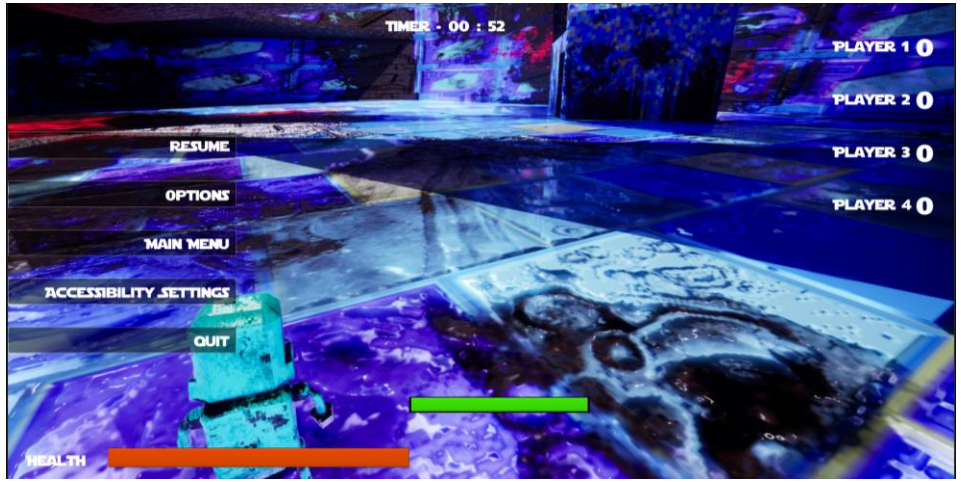
[Image: Example of HUD]

UI Components - Pause and Settings Menu

Intuitive navigation: Organize the menu options in a logical and consistent manner, enabling players to find and adjust settings with ease.

Responsive controls: Ensure that the UI responds quickly to player inputs, providing a seamless and frustration-free experience.

Customization options: Offer various settings for players to customize their gameplay experience, such as audio and video preferences, control schemes, and difficulty levels.



Accessibility Features

Colorblind-friendly options: Include alternative color schemes or icons to accommodate players with color vision deficiencies.

Adjustable text size: Allow players to increase or decrease the text size of the UI elements, ensuring readability for all users.

Customizable control schemes: Provide options for players to remap controls or adjust sensitivity settings, catering to their individual preferences and needs.

Iasmina Leagan

Networking

Specialism Implementations

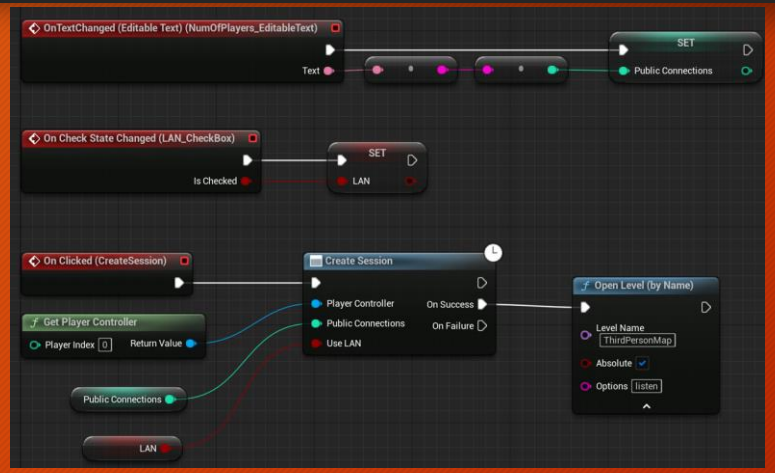
- Sessions
- Host/Join a game
- Spawn/Respawn
- Player colours
- Bullet damage
- Health system

Additional (general):

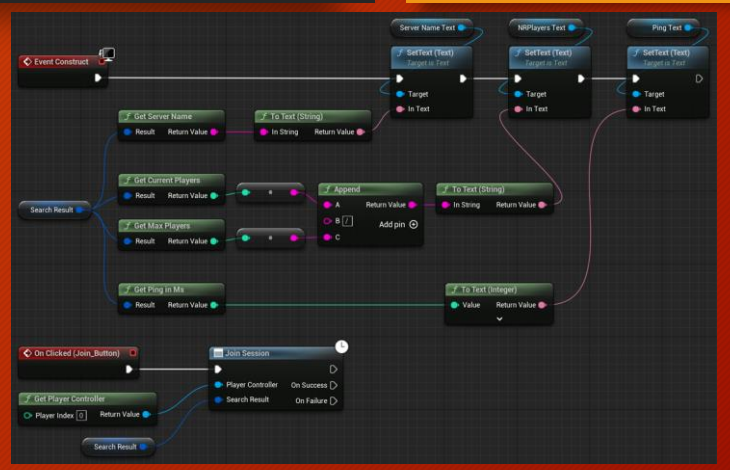
- Score system
- Pressure plates
- Moving platforms

Sessions - rejected functionalities

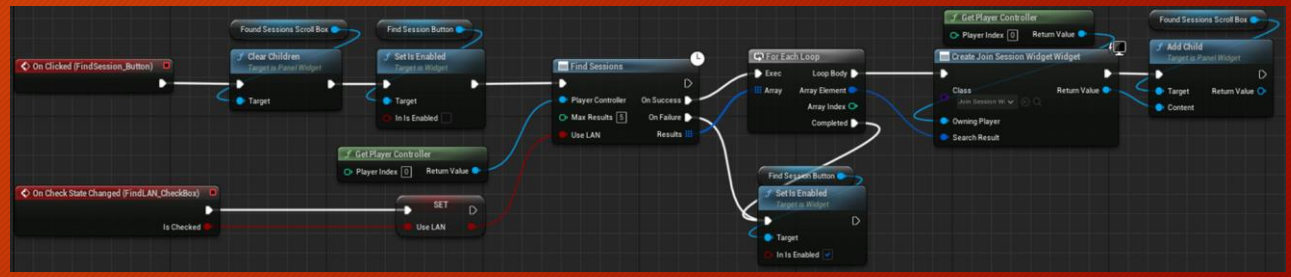
- Sessions were initially created with blueprints using subsystems and Unreal Engine's default system.
- Multiple failed tests resulted in its rejection.
- The reason could possibly be that the packet broadcast to the LAN's subnet was blocked.



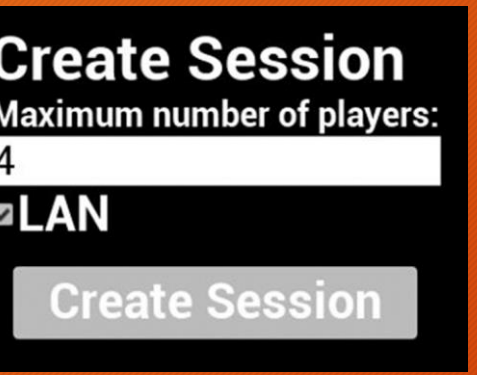
Create Session



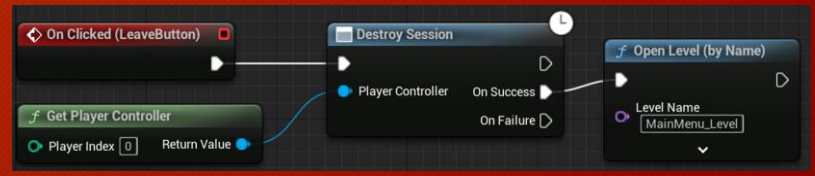
Join Session



Find Session



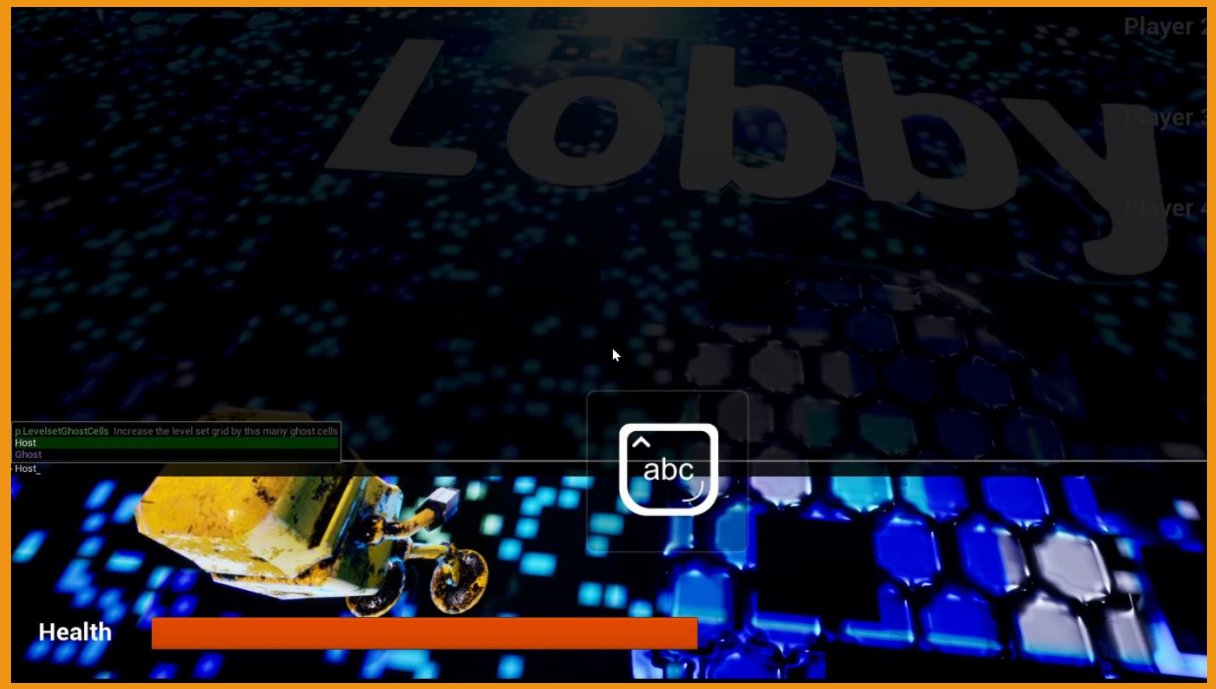
Initial Host/Join system



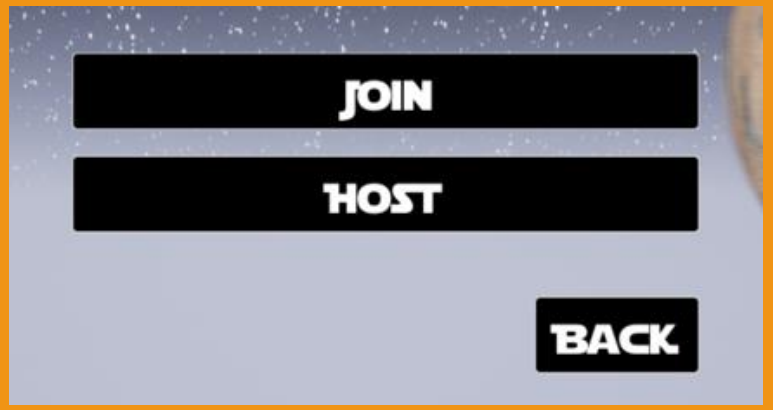
Delete Session

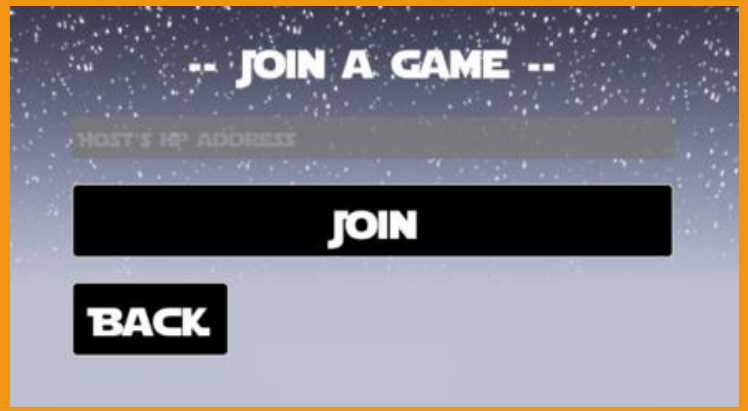
Updated Host/Join

- This was done using C++ classes and Widget blueprints.
- In the initial implementation, the players would spawn into a lobby and could host/join games using in-game console commands.
- The final version does not use the lobby anymore, since the players connect directly to the map through user interface buttons; the join functionality also takes the host's IP address as an input. The functionality of the multiplayer part of the menu system was coded using C++ classes.



Initial Lobby using in-game console commands





Final version

Implementation using C++ classes

The functionalities use a custom Game Instance.

The host function uses "Server Travel" which:

- Operates at world level;
- Can be called at server level or engine;
- Moves all participants.

The Join function uses "Client Travel" which:

- Operates at player control level (called on PlayerController);
- Tells the individual player controller to move (one at a time).

Additional: The Multiplayer Menu system uses an interface so that it can be attached to any other project; this uses a method called "Inversion of dependencies".

```cpp
void UMagnetMechanicsTPGameInstance::Host()
{
    //First deactivate the menu so it does not have control over the input
    if (MpMenu != nullptr)
    {
        MpMenu->DeactivateMenu();
    }

    UEngine* Engine = GetEngine();
    if (!ensure(Engine != nullptr)) return;

    Engine->AddOnScreenDebugMessage(0, 5, FColor::Green, TEXT("Hosting"));

    UWorld* World = GetWorld();
    if (!ensure(World != nullptr)) return;

    World->ServerTravel("/Game/ThirdPerson/Maps/MultiplayerMap?listen"); //The ServerTravel is called on
}

void UMagnetMechanicsTPGameInstance::Join(const FString& Address)
{
    //First deactivate the menu so it does not have control over the input
    if (MpMenu != nullptr)
    {
        MpMenu->DeactivateMenu();
    }

    UEngine* Engine = GetEngine();
    if (!ensure(Engine != nullptr)) return;

    Engine->AddOnScreenDebugMessage(0, 5, FColor::Green, FString::Printf(TEXT("Joining %s"), *Address));

    APlayerController* PlayerController = GetFirstLocalPlayerController();
    if (!ensure(PlayerController != nullptr)) return;

    PlayerController->ClientTravel(Address, ETravelType::TRAVEL_Absolute); //The ClientTravel is called o
}
```

The Host and Join functions in the new Game Instance C++ class

Replication

- How Unreal Engine handles replication:

Step 1 – The setting "Replication" must be turned on for each class so that the server has the ability to overwrite variables.

Step 2 – To update the server's variables to the clients, these must have one of the two configurations:

- Replicated
- RepNotify (ex: the player's colour)

Step 3 – The clients must use Remote procedure calls (RPSc), so they can communicate back to the server. There are 3 types of RPCs:

- Run on Server (ex: applying damage)
- Owning client (ex: updating the player's health for the health-bar)
- Multicast – broadcasts an event on the server and all connected clients (but it is executed from the server)

Replication – necessary blueprints

Q: What can be replicated?

A: Everything except the GameMode (it only exists on the server) and widgets that only exist on the owning clients.

Server
Game Mode

Variables:
- Array with connected players

Server & Owning Clients
Player Controller

Functions:
Spawn/Respawn

Server & Clients
Character (Pawn)

Widgets:
- Health-bar

Variables:
- Health

(Ammo, fire rate)

Functions:
- Spawn projectile
- Take damage
- Timer

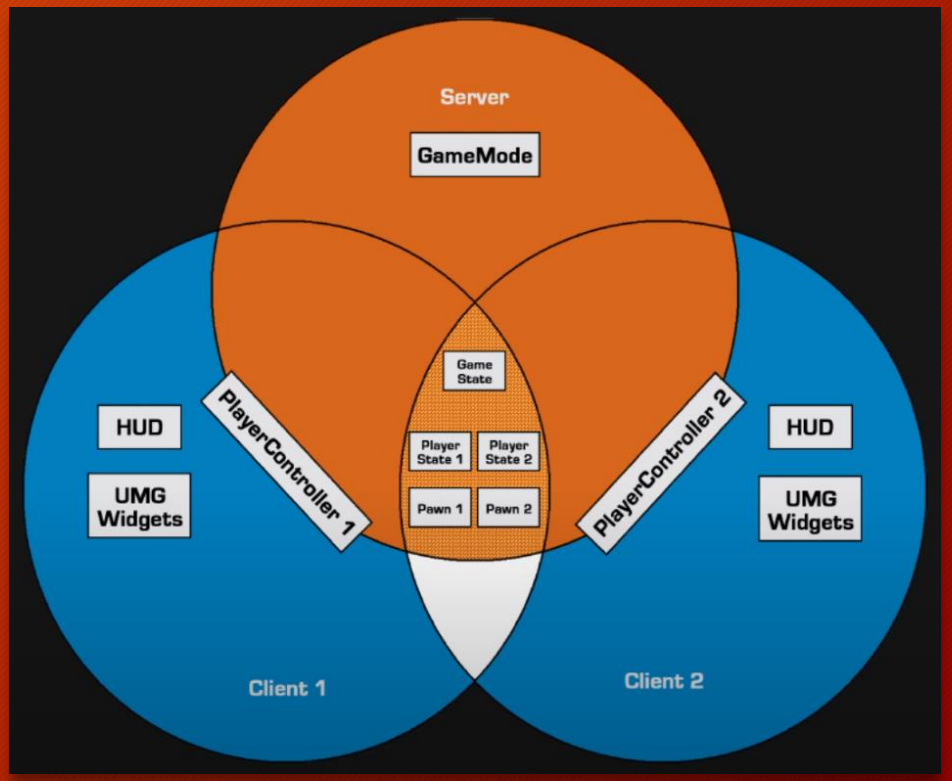
(Reload)

Server & Clients
Game State

Variables:
- Player scores

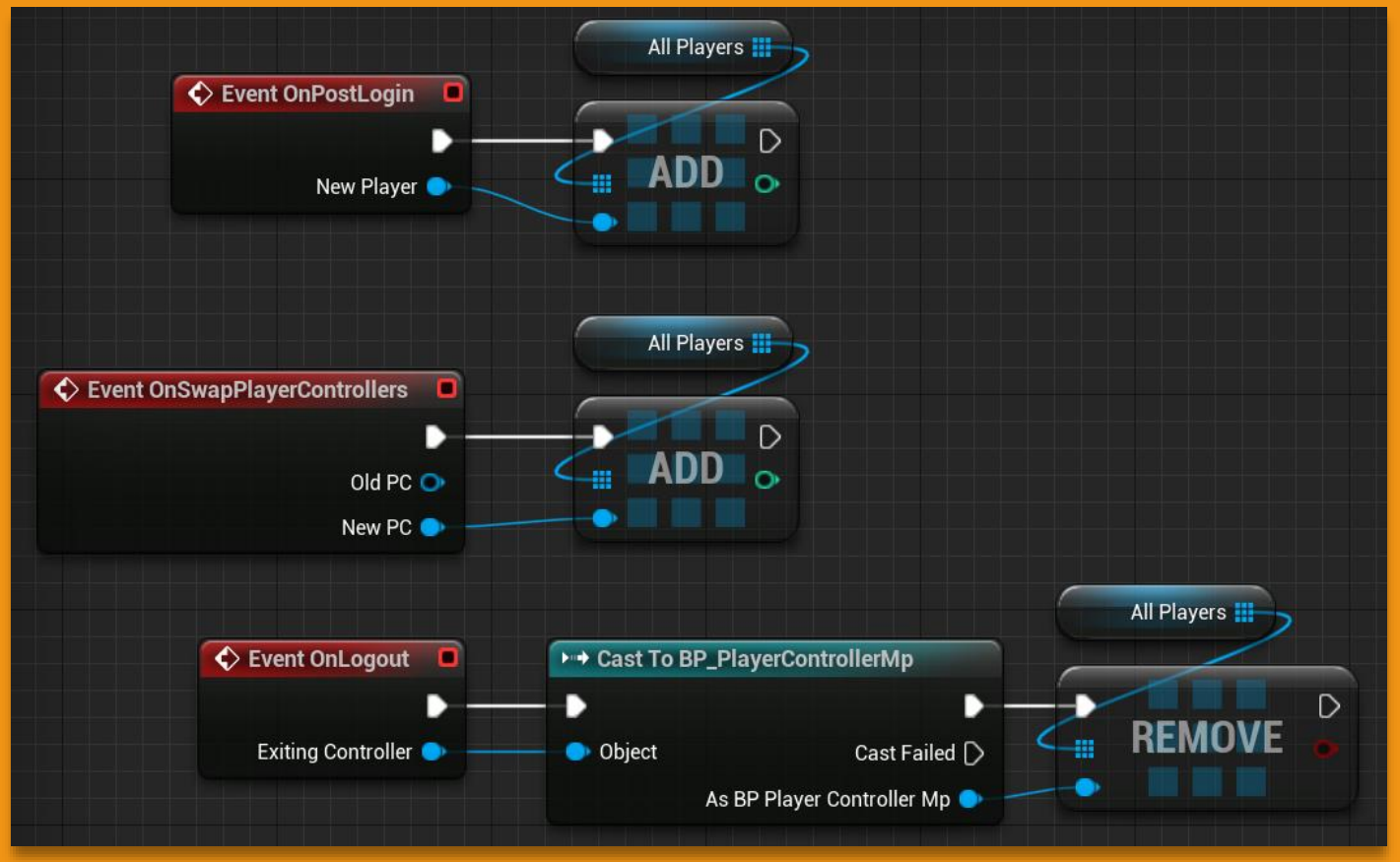
Server & Clients
Player State

Variables:
(Player name, username, ID)



Replication – Game Mode

- The GM contains all the classes that define the rules and logic for the game.
- When each player connects, a copy of their Player Controller is stored in an array. This gets updated when the players change levels or disconnect.



Storing the connected players in an array

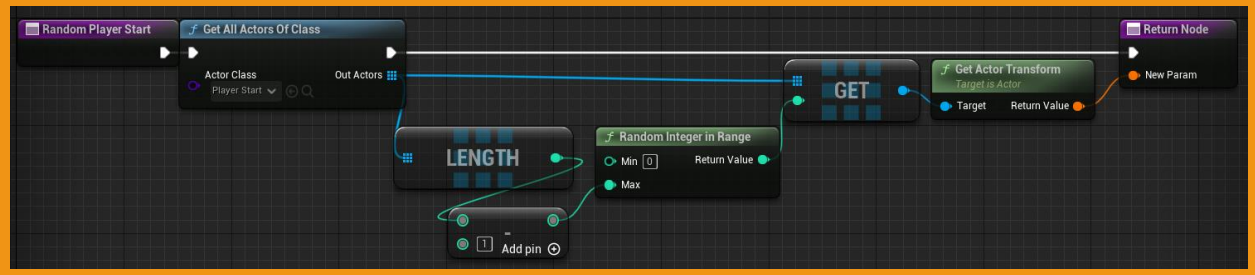
Spawn/Respawn

The Pawn (character) is not set in the world setting "Default Spawn Class" because the player is spawned through code for the respawn system.

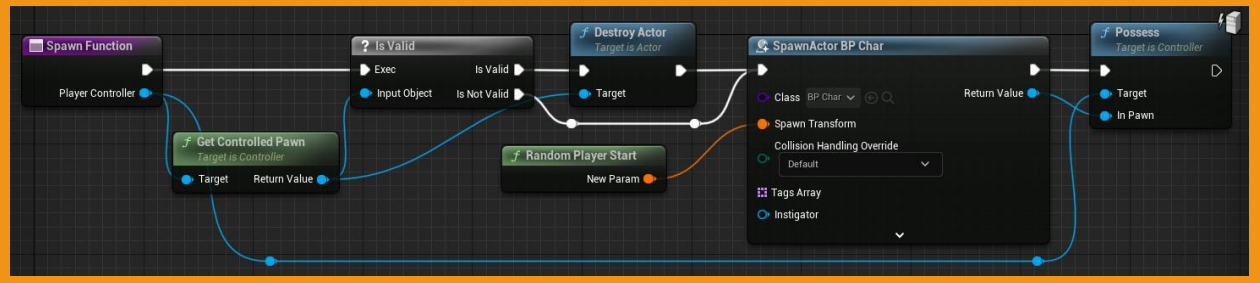
The Game Mode contains two functions:

- The first one gets a random location to spawn the player;
- The second one Spawns the Pawn, but this pawn needs to be controlled by a PlayerController, therefore it gets assigned through a function called "Possess".

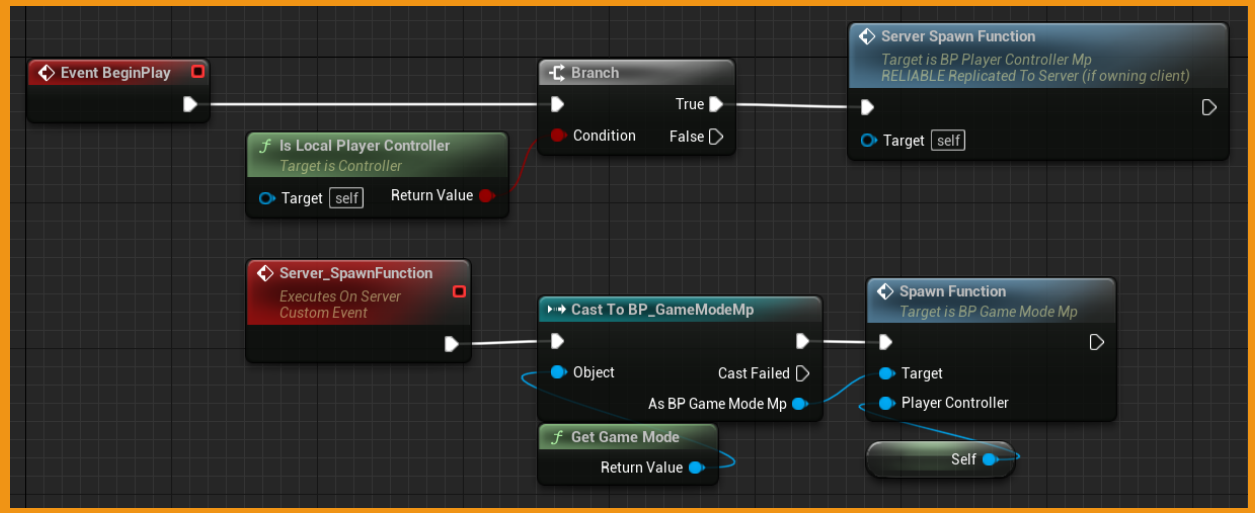
Although the spawn function is on the Game Mode, the Player Controller is actually the one calling the function to spawn it.



Random spawn point (Game Mode)



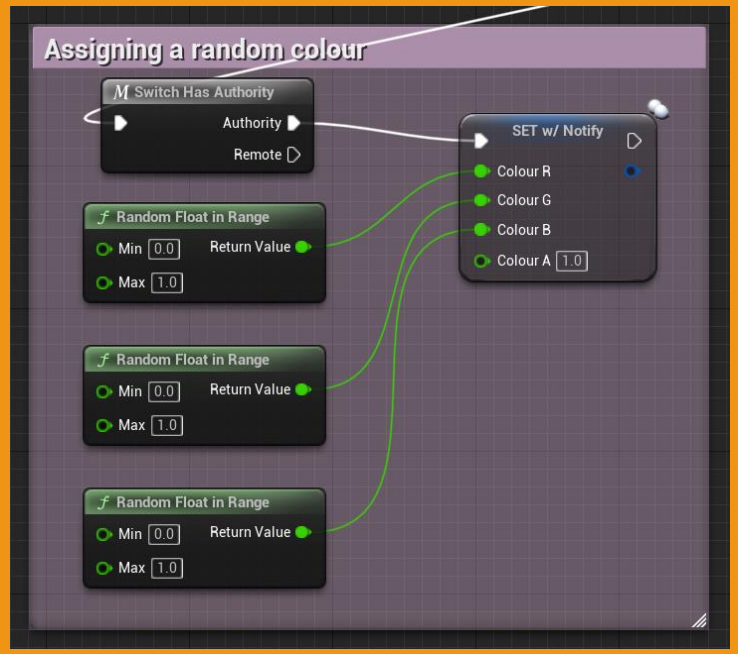
Spawn function (Game Mode)



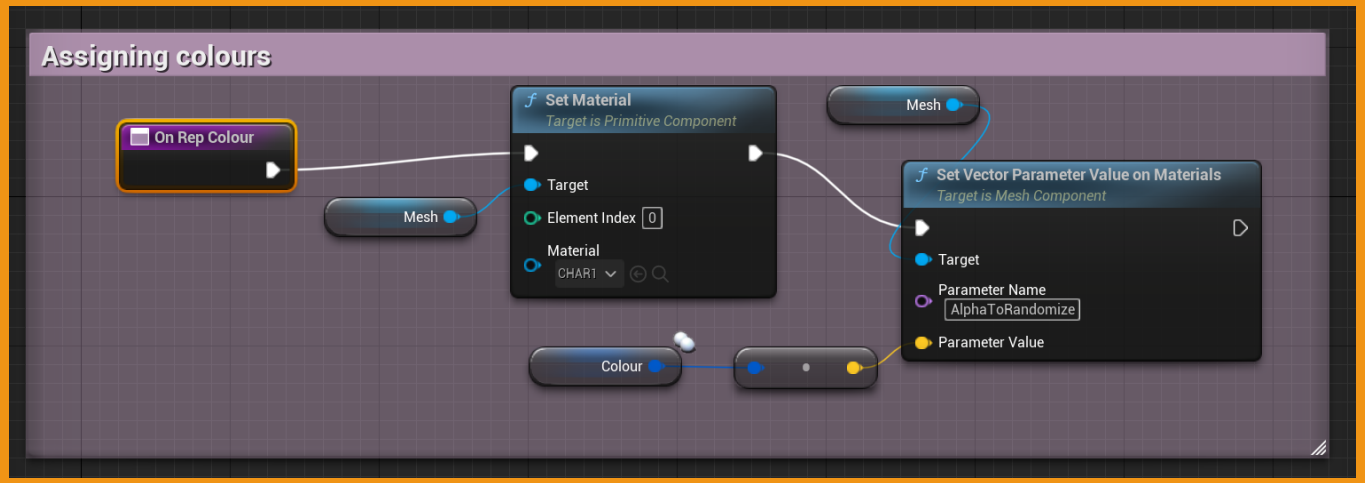
Spawn function (Player Controller)

Replication – Different colours for players

- The colour of the player's material is stored in a RepNotify variable, which is set only by the server, since it has authority.
- The OnRep_Colour function is called each time the variable changes and it takes the colour and assigns it to the Pawn's material.



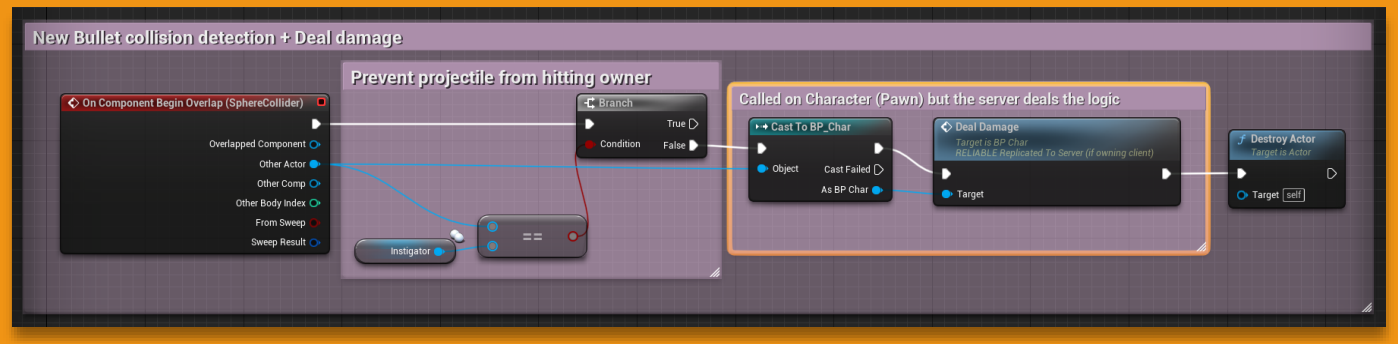
Setting a random colour



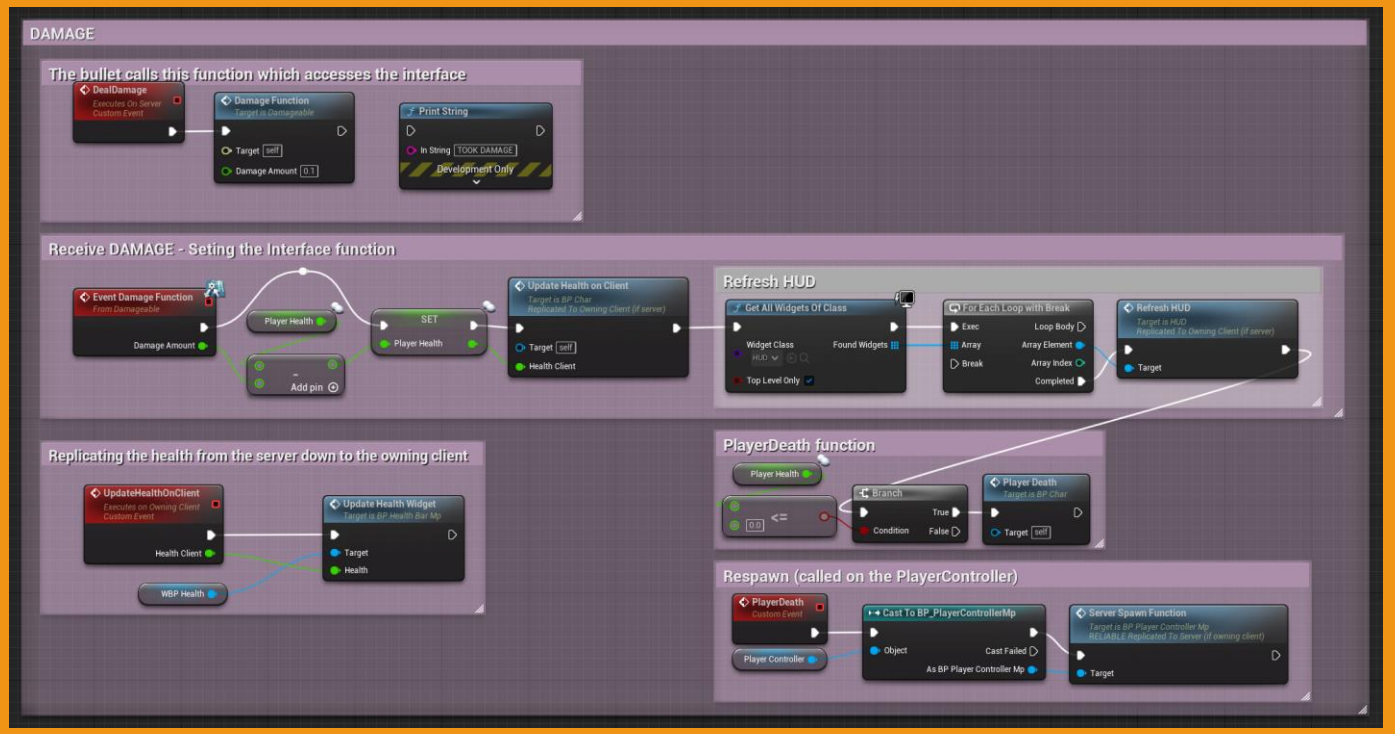
OnRep_Colour

Replication – Bullet + Health

- Using Overlap functions, if the bullet collided with a Pawn, a function is called on the Pawn to decrease the health.
- Also, the bullet checks the name of person who spawned it to avoid dealing damage to themselves.
- The logic happens on the server to make sure all clients sync, but the value is further transmitted to an "Owning client" type function, which updates the specific health-bar UI of that player.



Calling the dmg function on collision (inside bullet actor)



The "receiving damage" logic (inside Pawn)

Additional features

- A moving platform object that moves in between 2 points. This is replicated so that all players can see it moving.
- A pressure plate (C++ class and blueprint inherited from it) that activates the moving platform (it can be used for future team collaboration mode).

```cpp
void APressurePlate::OnOverlapBegin(class UPrimitiveComponent* OverlappedComp, class AActor* OtherActor, class UPrimitiveComponent* OtherComp,
    int32 OtherBodyIndex, bool bFromSweep, const FHitResult& SweepResult)
{
    if (HasAuthority())
    {
        UE_LOG(LogTemp, Warning, TEXT("Activated"));
        for (AMovingObject* Platform : ObjectsToTrigger)
        {
            Platform->AddActivePpTrigger();
        }
    }
}

void APressurePlate::OnOverlapEnd(UPrimitiveComponent* OverlappedComp, AActor* OtherActor, UPrimitiveComponent* OtherComp, int32 OtherBodyIndex)
{
    if (HasAuthority())
    {
        UE_LOG(LogTemp, Warning, TEXT("Deactivated"));
        for (AMovingObject* Platform : ObjectsToTrigger)
        {
            Platform->RemoveActivePpTrigger();
        }
    }
}
```

Using OnOverlap events to move the platform

```cpp
void AMovingObject::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);

    if (activePpTriggers > 0)
    {
        if (HasAuthority())
        {
            //object's location
            FVector Location = GetActorLocation();

            //the distance = the length of the Vector
            float totalJourneyLenght = (GlobalTargetLocation - GlobalStartLocation).Size();
            float travelledJourneyLenght = (Location - GlobalStartLocation).Size();

            if (travelledJourneyLenght >= totalJourneyLenght)
            {
                FVector ContainerForSwap = GlobalStartLocation;
                GlobalStartLocation = GlobalTargetLocation;
                GlobalTargetLocation = ContainerForSwap;
            }

            FVector Direction = (GlobalTargetLocation - GlobalStartLocation).GetSafeNormal();

            Location += speed * DeltaTime * Direction;
            SetActorLocation(Location);
        }
    }
```

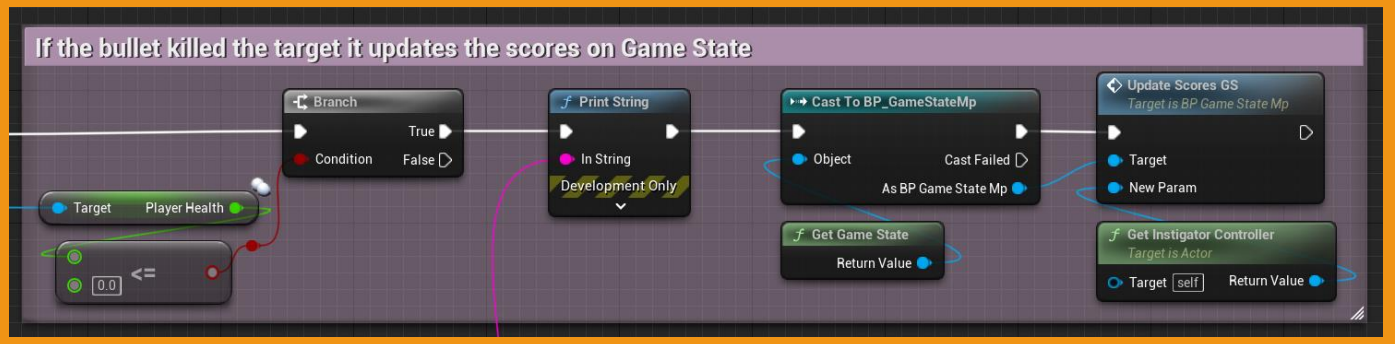
Code to make the object move

```cpp
void AMovingObject::BeginPlay()
{
    Super::BeginPlay();

    if (HasAuthority())
    {
        SetReplicates(true);
        SetReplicateMovement(true);
    }

    //GlobalStartLocation = GetLocalLocation();
    //GlobalTargetLocation = GetTransform();

    GlobalStartLocation = GetActorLocation();
    GlobalTargetLocation = GetTransform().TransformPosition(TargetLocation);
}
```

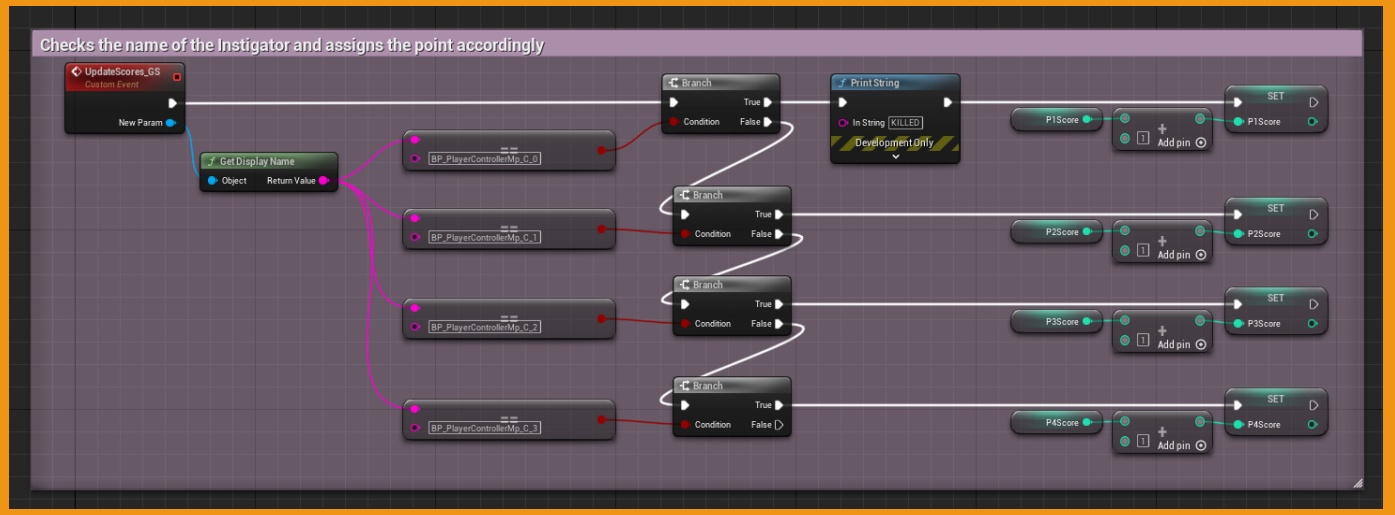
Setting the replication so that all clients can see the object moving

Additional functionalities - Score system

- Whenever a bullet hits a player, it checks if their health is less or equal to 0.
- If this is true, it checks who shot the bullet using the Instigator and assigns them a point.
- All points are stored in the Game State, which can be accessed by the server and the clients.



Score update if target is killed
(in Bullet actor)



Updating player scores
(on Game State)

Rejected functionalities and future implementation

Rejected:
- Sessions

Future:
- The players spawn into a lobby system instead of directly in the game
- They all get connected once a specific number of players connected
- Countdown before spawning a player
- The timer should be moved into the Game State

References

- Planets Textures - https://substance3d.adobe.com/assets/allassets/7145ed862594a618610d6efba6a07776b337d85f?assetType=substanceMaterial&assetType=substanceAtlas&assetType=substanceDecal&q=planets%20major_planet%20satellite%20planet%20sci%20fi&u=planet%20sci-fi
- Sound Effects from www.Pixabay.com & www.FreeSounds.com