Short Introduction

This project showcases platform game mechanics such as the player using button sequences and pressure plates to open doors, moving platforms, falling platforms, lasers that kill the player if touched, and rotating objects. The purpose of this project is to show that I have a grasp of how to navigate through the Unreal Engine IDE and implement logic using Blueprints and C++.

Chamber One: Button Sequence Opens Door Mechanic

How was this implemented?

I implemented this logic by creating two classes:

1. **FPButton**

- int ButtonId: Initially used for door sequence, but I later deprecated the idea and used pointers to buttons instead.
- bool IsUsing: Used as a flag to prevent activation of the button from firing multiple times.
- bool IsDisplayingError: Used as a flag to prevent error visuals and sound effects from firing multiple times.
- float ErrorDurationTimer: Controls how long the error visual is active before returning to the normal button state.
- AFPDoor* AssignedDoor: Pointer to the door this button is associated with.
- void ActivateButton(): Called when the player walks on the button.
- void DisplayError(): Called when IsDisplayingError is first set to true. The color of the button changes, and the error sound effect is played.
- void DisplaySuccess(): Called when AssignedDoor's CompareSequences() returns true. The color of the button changes, and the success sound effect is played.
- void DisplayCompleted(): Called when AssignedDoor's OpenDoor() is run. The button's color changes permanently, and it can no longer be interacted with.
- bool CanInteractWith(): Used as a flag to avoid interaction with the button if IsUsing or IsDisplayingError is true.

2. **FPDoor**

- int DoorType: Stores what type of door this is. 0 stands for button sequence door, 1 stands for pressure plate door.

- TArray<AFPButton*> CorrectButtonSequence: Stores the button sequence the player must enter to activate the door.
- TArray<AFPButton*> CurrentButtonSequence: Stores the button sequence the player is currently entering.
- void OpenDoor(): Called to open the door after the correct sequence is entered.
- bool ProcessButton(): Handles what happens when the player walks on a button and returns true if activated.
- bool CheckCurrentValues(): Checks if current buttons entered are correct so far.
- bool CheckForIdenticalSequenceLengths(): Checks if the lengths of CorrectButtonSequence and CurrentButtonSequence match.
- bool CompareSequences(): Compares all values in CurrentButtonSequence and CorrectButtonSequence after their lengths are confirmed to match.
- void AddButton(AFPButton* Button): Adds a button to the CurrentButtonSequence array.
- TArray<AFPPressurePlate*> PressurePlatesRequired: Stores the pressure plates the player must activate for the door to open.
- bool ProcessPressurePlate(AFPPressurePlate* PressurePlate, bool IsAdding): Handles what happens when a pressure plate associated with the door is activated or deactivated.
- bool CheckIfAllPressurePlatesAreActive(): Checks if all required pressure plates are activated; returns true if they are.

I then created a Blueprint class based on the C++ class. I used shapes and box collisions to set up the Blueprint-derived classes named BP_Button and BP_Door. I gave Blueprint logic to the functions listed above, placed the actors into the world, and set the default values required (such as AssignedDoor and CorrectButtonSequence).

Any specific design goals or constraints?

My main goal was to have a button and door set that could be dragged and dropped and used anywhere. Initially, I planned to use button IDs, but that felt tedious when using many buttons over time. Tracking each ID and ensuring no duplicates (avoiding human error) led me to scrap that idea and use pointers instead. Now, I can drag and drop a few buttons and a door, rename them for clarity, and set default values to have it working in seconds. I wanted a quick setup with an unlimited number of buttons.

Chamber Two: Platforming Challenge

How was this implemented?

I implemented this logic by creating four classes:

1. **FPMovingPlatform**

- FVector CurrentLocation: Stores the actor's location for moving and updates via SetActorLocation().
- float MovementSpeed: Controls how fast the platform moves.
- TArray<FVector> NavigationLocations: Positions the platform navigates to in order.
- int CurrentNavigationIndex: Tracks the current target location.
- bool bIsReversing: Determines if navigation index increments or decrements.
- bool bIsUpdatingNavigationLocation: Prevents UpdateNavigationLocation() from firing multiple times.
- void MovePlatform(): Moves the platform to the current target location.
- void UpdateNavigationLocation(): Updates the navigation index and reversing flag.

2. **FPFallingPlatform**

- float FallTimer: Controls how long before the platform starts falling.
- float WarningBlinkTimer: Duration of the warning effect before falling.
- float FallSpeed: Controls how fast the platform falls.
- bool bIsFalling: Indicates if the platform is currently falling.
- bool bIsWarning: Indicates if the warning state is active.
- void PlayFalling(): Plays visual and sound cues for falling.
- void EndFalling(): Stops the falling cues.
- void PlayWarning(): Plays warning cues before falling.
- void EndWarning(): Stops the warning cues.

3. **FPKillZone**

- No additional members.

4. **FPGoalPlatform**

- No additional members.

I then created Blueprint classes for each of these. I also created widget Blueprints for the death and victory screens. I added logic for overlap events for the KillZone and GoalPlatform.

Any specific design goals or constraints?

My main goal was creating moving platforms that could have unlimited positions. The falling platform was simple: an actor that performs an action after a set timer triggered by interaction. I felt constrained

creatively. I wanted, for example, a crumbling effect where the platform broke into pieces as it fell—opening the door for split-second player decisions.

Chamber Three: Pressure Plate Puzzle

How was this implemented?

I created three classes:

1. **FPGroundItem**
- bool bIsCurrentlyPickedUp: Tracks if the object is picked up.
2. **FPPressurePlate**
- bool bIsCurrentlyActive: Tracks if the plate is active.
- bool bDoorHasBeenActivated: Tracks if the door has been opened.
- AFPDoor* Door: Pointer to the door this plate controls.
- TArray<AActor*> CurrentActors: Stores overlapping actors.
- void RemoveActor(AActor* Actor): Removes an actor from CurrentActors.
- void AddActor(AActor* Actor): Adds an actor to CurrentActors.
3. **FPGroundFloor**
- No additional members.

In Blueprint:

- **BP_GroundItem**: Cube and collision box, drop/pickup events, StartingLocation saved at spawn, Timer for auto-reset, bIsReturning flag, pointer to PressurePlateHit.
- **BP_PressurePlate**: Shapes/collisions, overlap logic, material updates in Tick.
- **BP_GroundFloor**: Collision box, overlap logic to reset game when touched.

Any specific design goals or constraints?

I wanted scalable doors that could require any number of plates. A tricky part was handling dropped objects that became unretrievable. My solution: a GroundFloor actor that resets objects on overlap. This works, but could be improved further.

Chamber Four: Synthesis Puzzle

How was this implemented?

I combined previous classes with two new ones:

1. **FPLaserBeam**

- float Timer: Controls delay between laser toggles.
- bool bIsActive: Tracks if laser is on/off.
- void ToggleLaserBeam(): Logic for toggling laser.

2. **FPRotatingPlatform**

- float RotationSpeed: Speed of rotation.
- FVector RotationAxis: Rotation axis.
- bool bIsRotating: If the object is rotating.
- bool bLimitRotation: If rotation should be limited.
- float MaxRotationAngle: Max allowed rotation.
- float AccumulatedRotation: Tracks rotation so far.
- void StartRotating(): Starts rotation.
- void StopRotating(): Stops rotation.
- void SetRotatingSpeed(float NewSpeed): Sets speed.
- void SetRotationAxis(FVector NewAxis): Sets axis.
- void TickRotate(float DeltaTime): Rotation logic called in Tick.

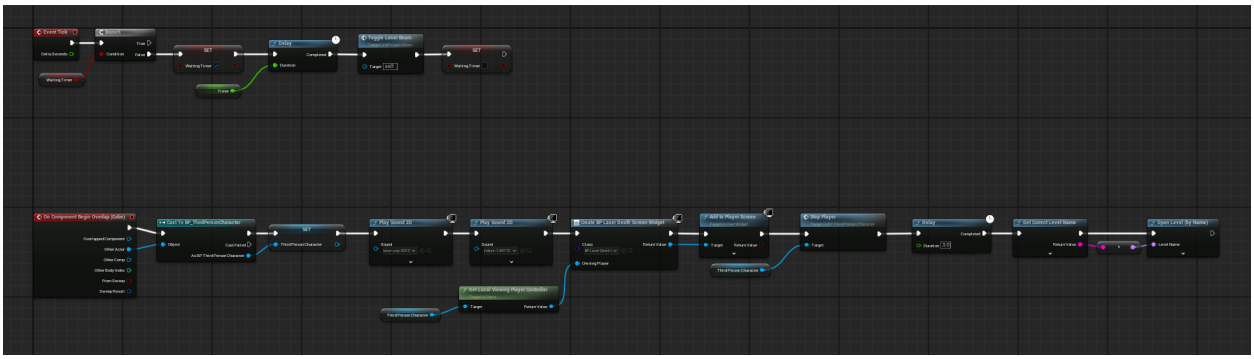
In Blueprint:

- **BP_LaserBeam**: Cube + material, overlap logic, toggle timer.
- **BP_RotatingPlatform**: Cube base, starts rotating on BeginPlay.

Any specific design goals or constraints?

My goal was to combine mechanics simply and effectively. Lasers and rotating platforms provided that opportunity. I also experimented with combining mechanics (e.g., button + rotating platform, falling platform + button).

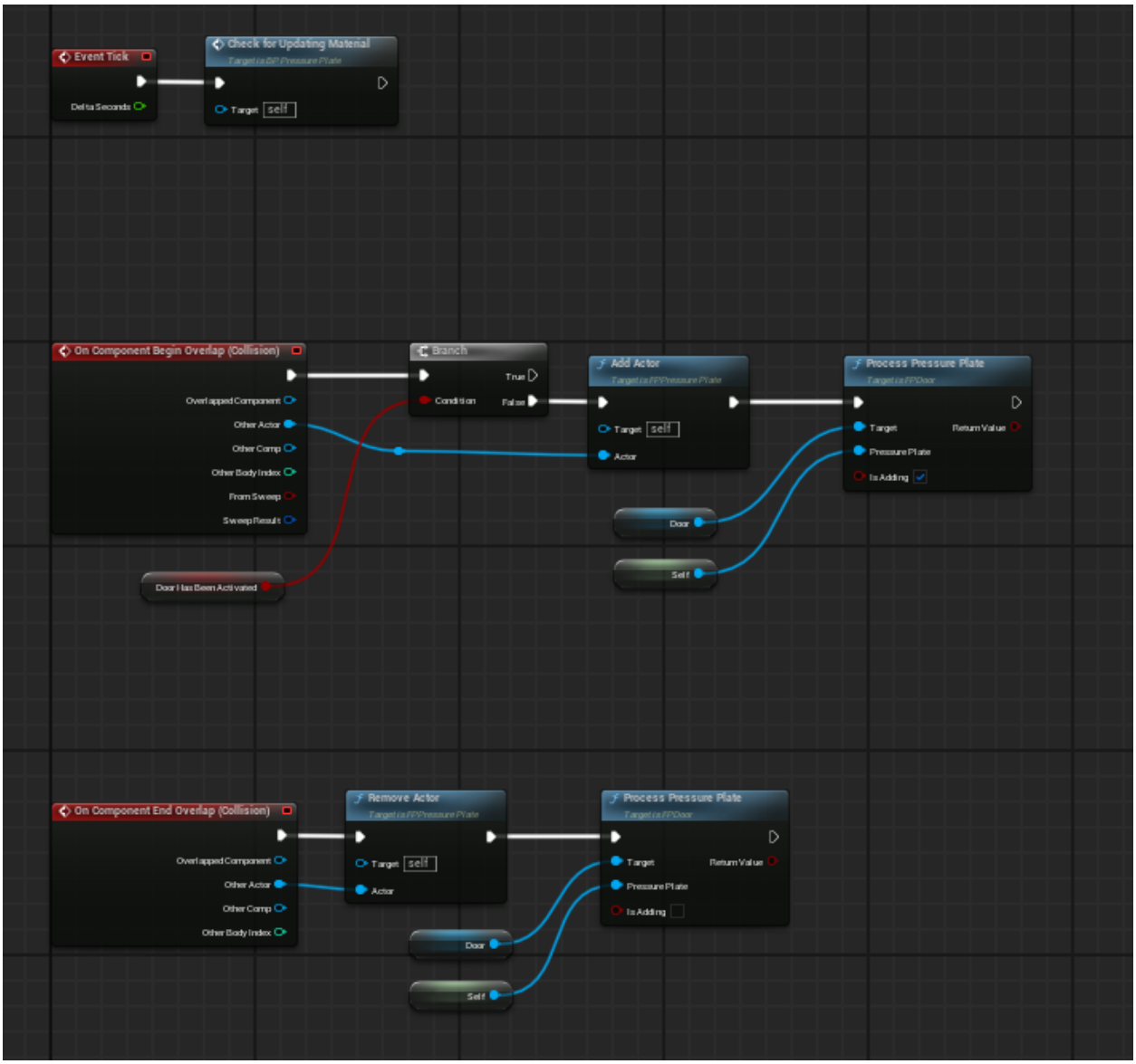
Annotated screenshots of key Blueprint or C++ implementations



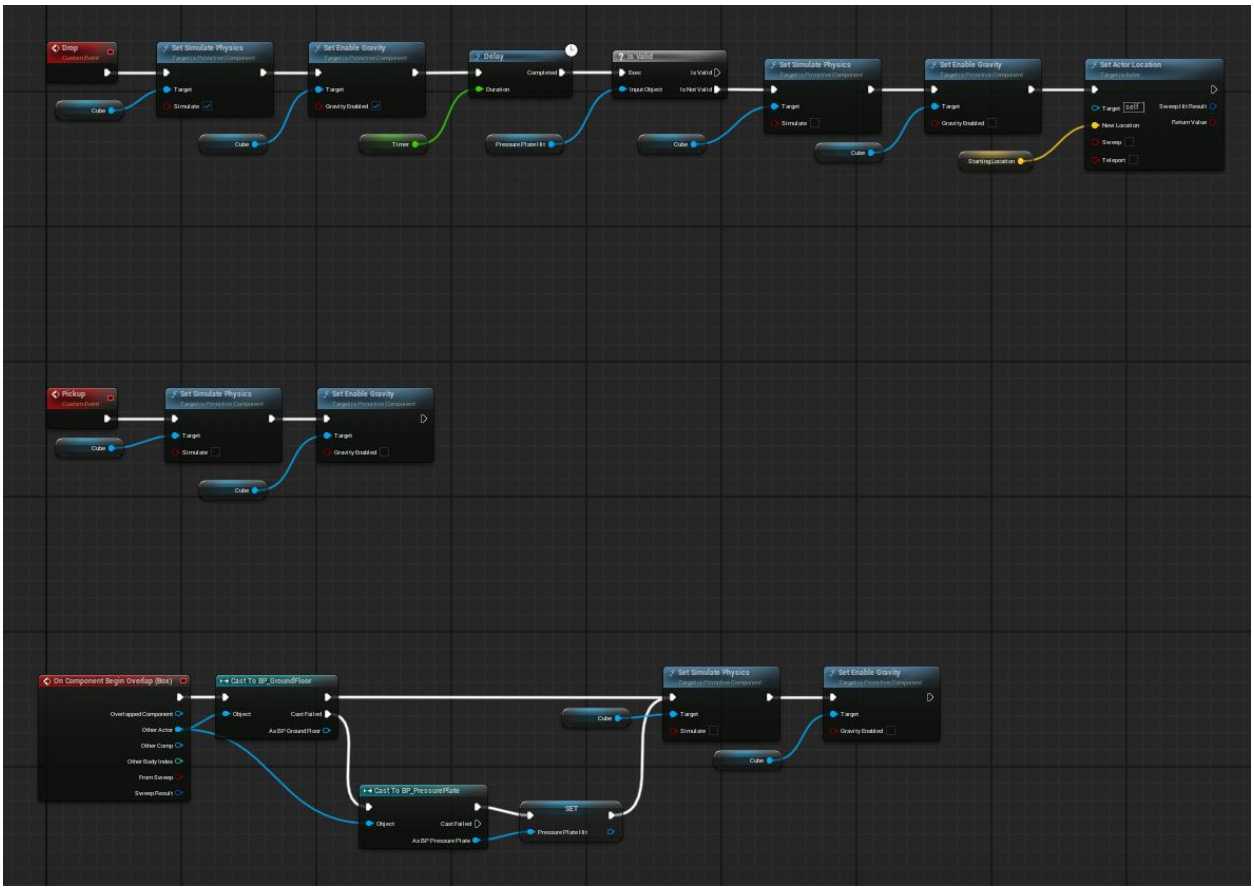
Laser beam logic in blueprints. Checks if overlapped actor is the player then handles death action and restarts level. In the tick method, toggles laser beam's visibility and resets flag after.

```cpp
void AFPRotatingPlatform::TickRotate_Implementation(float DeltaTime)
{
    FRotator DeltaRot = FRotator::ZeroRotator;
    FVector LocalAxis = RotationAxis.GetSafeNormal();

    float DeltaAngle = RotationSpeed * DeltaTime;

    DeltaRot += FQuat(LocalAxis, AngleRad: FMath::DegreesToRadians(DeltaAngle)).Rotator();

    AddActorLocalRotation(DeltaRot);

    if (bLimitRotation)
    {
        AccumulatedRotation += DeltaAngle;
        if (AccumulatedRotation >= MaxRotationAngle)
        {
            StopRotating();
        }
    }
}
```

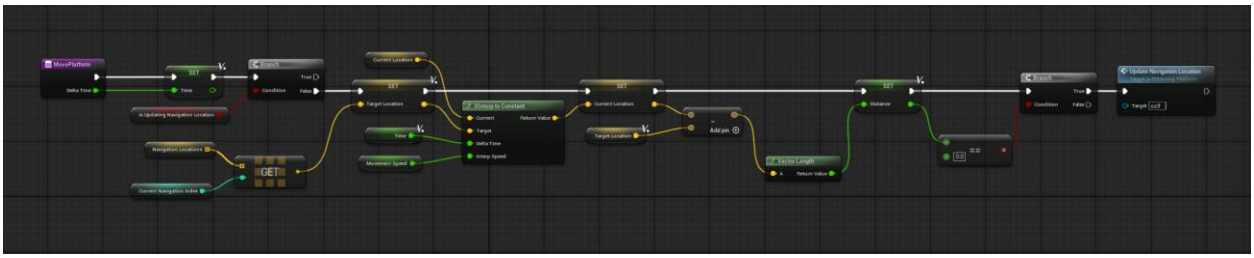
This function controls how the platform spins during the game. Each frame, it figures out how far to rotate based on the speed and time, and then applies that rotation. If the platform is supposed to stop after turning a certain amount, it keeps track of the total rotation and stops when it hits the limit.



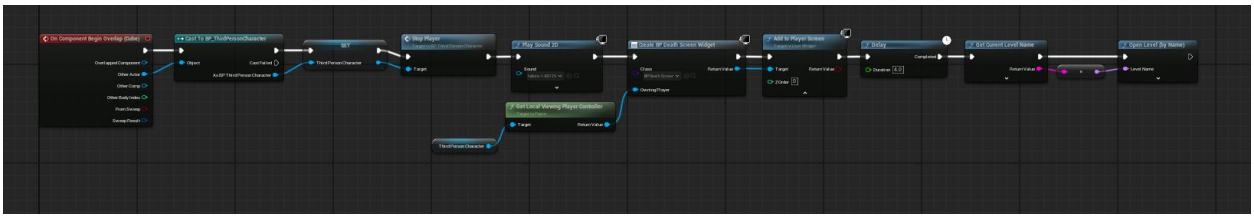
The logic above for the pressure plate attempts to add/remove an actor to the pressure plate then use the door's process method. Each tick checks if the pressure plate's characteristics such as color needs to be updated.



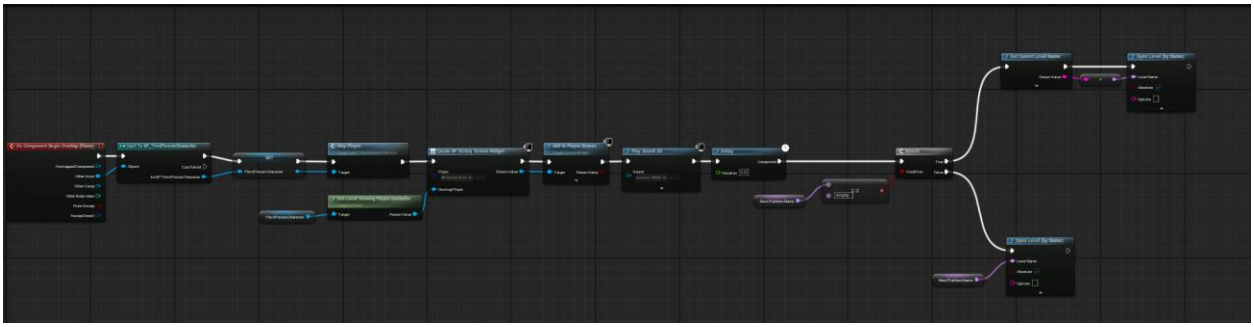
This is the ground item logic. Basic pickup and drop events that affect the item's gravity, physic, and collision settings.



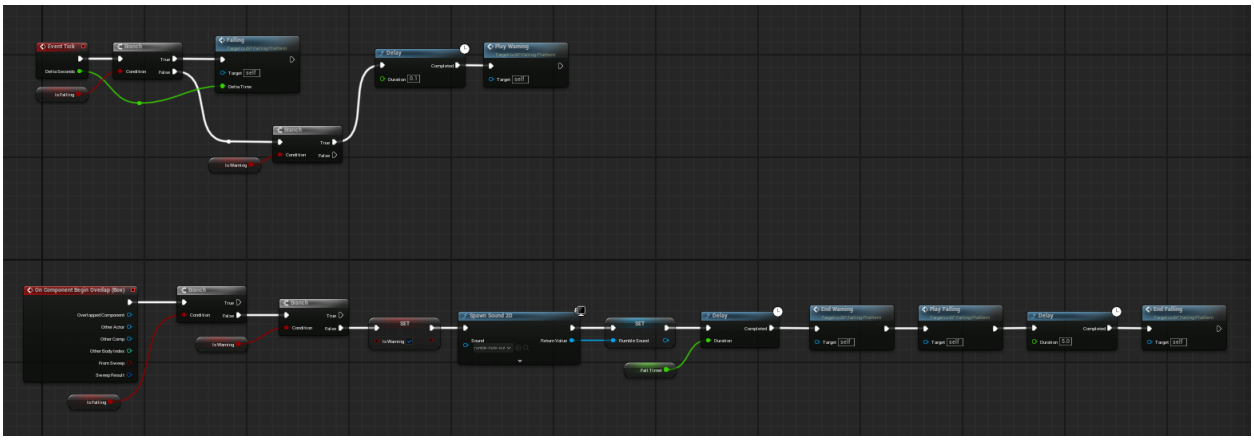
The image above contains logic for the moving platform. Current location is set, location traveling to is set, and the length between the two are calculated to find out if the platform needs to an update for a new location.



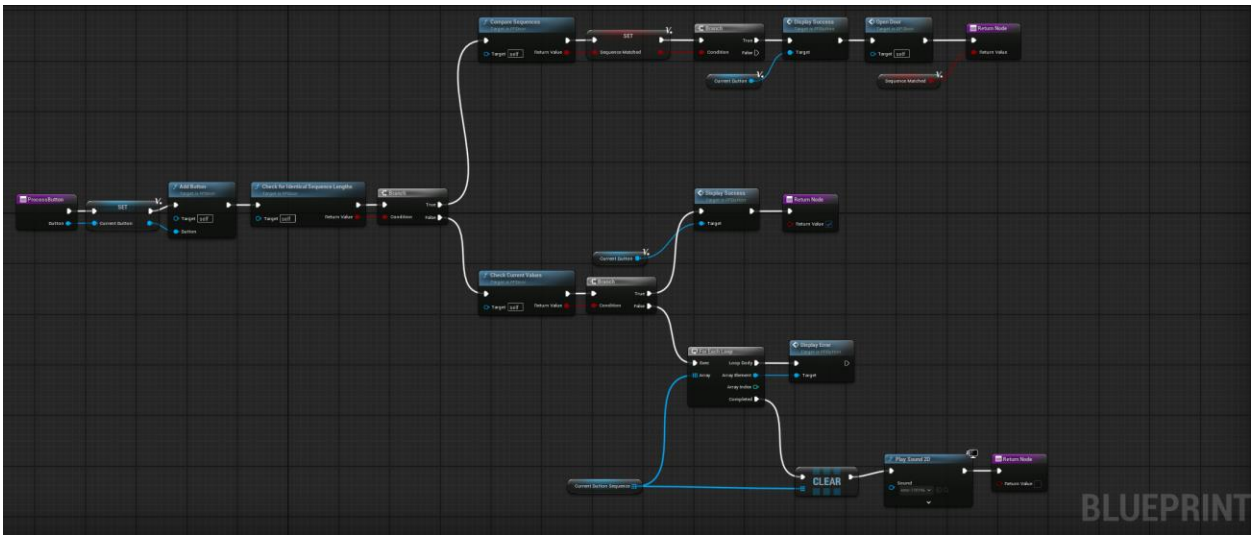
This is the logic for the kill zone. Simply checks if the player is overlapping the kill zone and halts player then resets level after delay ends.



This is logic for the goal platform. Checks if player has overlapped it and moves to the next level based on a variable.



This is the logic for the falling platform. When the player overlaps it, starts the warning function then a delay is set for the falling function. Once the function to end the falling function is called, the falling platform is destroyed.



This is logic for the ProcessButton method under BP_Door. Very straightforward. Attempts to add button, checks if sequence length for correct & current is identical. If they are, checks if all of the values are correct. If they are not, checks if the player is entering the correct values. Success and Failure methods are called depending on the results.

What did I learn?

I learned how to implement logic with scalability in mind, maintaining simplicity, and allow room for modifications to be made.

What went well?

I made sure each mechanic requested was implemented correctly with visual and sound feedback for the player as they navigated through the game.

What would you improve on or expand in the next stage?

I would like to improve on more flexibility when it comes to adding support for mechanics supporting one another.

A list of all prompts given to AI tools (if any were used)

Used an AI prompt in ChatGPT to create rotation code for FRRotatingPlatform.

References to any additional learning resources utilised (e.g. Youtube tutorials)

None, but if AI was not present, I would have searched up further information on rotating objects. I found majority of what I needed by searching through blueprint's catalog of functions in Unreal Engine. For example, I was having issues with Moving Platform's speed slowing down as it reached destination but tested using VInterp To Constant and it worked.