001

Star Menders

Development and Design Document

Daniel George Mark Dore

Brief

In this project, I have created a

Contents

List of Figures

Development

In this section, I explain what I created in each version, any methods or prototypes I created to complete the task and go further in-depth about certain features

1 EnhancedInput Implementation

Commit

My first task for the project was implementing the new EnhancedInput system, introduced in Unreal Engine 5. This system is described as a 'powerful system for reading and interpreting user input', and provides more features than the previous system such as 'like radial dead zones, chorded actions, contextual input and prioritization'.

In my previous Unreal Engine 4 projects, I used the Action and Axis system to bind inputs to events in the character themselves via the SetupPlayerInputComponent function. From there, inputs from a wide range of hardware could be set, such as controller, keyboards and other external devices.

```
// Example of Action/Axis binds //
// Add Axis Binds
PlayerInputComponent->BindAxis("MoveX", this,
    &ABaseCharacter::MoveX);
PlayerInputComponent->BindAxis("MoveY", this,
    &ABaseCharacter::MoveY);
PlayerInputComponent->BindAxis("CameraX", this,
    &ABaseCharacter::CameraX);
PlayerInputComponent->BindAxis("CameraY", this,
    &ABaseCharacter::CameraY);
PlayerInputComponent->BindAxis("Zoom", this,
    &ABaseCharacter::ZoomThirdPersonCamera);
PlayerInputComponent->BindAxis("Zoom", this,
    &ABaseCharacter::FocusCamera);
PlayerInputComponent->BindAxis("InteractModeScroll", this,
    &ABaseCharacter::InteractModeScroll);
```

Instead, the EnhancedInput system uses Input Mapping Context and Input Action data assets to define and describe the rules for triggering one or more inputs. Users can set an array of different trigger types for inputs such as Release, Hold and Pulse to control when the inputs are fired, while Modifiers can change how the values produced from the inputs are changed.

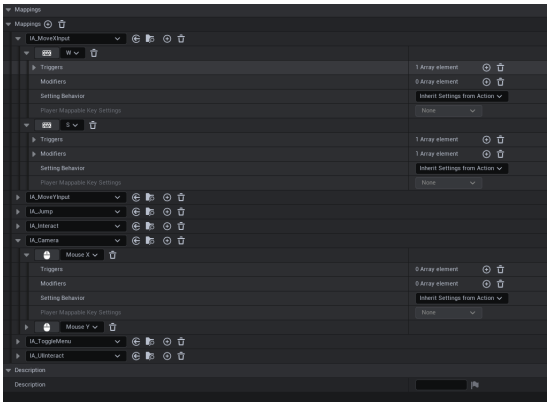

Figure 1. The InputMappingContext. Users can create a table of different inputs with varying trigger types and modifiers

I began by creating a separate UDataAsset class to hold all of my InputActions together, rather than finding them one by one via FObjectFinder. I also created a InputMappingContext to setup the input types and modifiers for each input. Both of these object are then found in the ControllerPlayer and implemented via the SetupInputComponent function.

```
UCLASS()
class STARMENDERS_API UInputConfigData : public UDataAsset
{
GENERATED_BODY()

public:
/// -- Movement Inputs --
// Pointer to the forward/backwards player input
UPROPERTY(EditDefaultsOnly, BlueprintReadOnly)
class UInputAction* MoveXInput = nullptr;

// Pointer to the strafe player input
UPROPERTY(EditDefaultsOnly, BlueprintReadOnly)
class UInputAction* MoveYInput = nullptr;

// Pointer to the mouse player input
UPROPERTY(EditDefaultsOnly, BlueprintReadOnly)
class UInputAction* CameraInput = nullptr;

// Pointer to the toggle menu player input
UPROPERTY(EditDefaultsOnly, BlueprintReadOnly)
class UInputAction* MenuInput = nullptr;

// Pointer to the jump player input
UPROPERTY(EditDefaultsOnly, BlueprintReadOnly)
class UInputAction* JumpInput = nullptr;

// Pointer to the interact player input
UPROPERTY(EditDefaultsOnly, BlueprintReadOnly)
class UInputAction* InteractInput = nullptr;

// Pointer to the UI Left click player input
UPROPERTY(EditDefaultsOnly, BlueprintReadOnly)
```

```
class UInputAction* UIInteractInput = nullptr;
};
```

Also unlike my past projects, I implemented my inputs directly into the controller. Previously, I would never implement my inputs directly into the controller class, rather the Pawn/Character. However, I learnt that the best practice to handle inputs is to have any similar inputs used by all controllable characters implemented into the controller, while unique inputs implemented on the character.

As the player will only ever control two characters with identical inputs via a parent class, I decided to implement all my inputs into the PlayerController class. If in future builds this changes, I will move any unique functions to their respective classes.

```
void AController_Player::SetupInputComponent()
{
Super::SetupInputComponent();

// Check if the input object pointers are valid
if (InputConfig && InputMapping) {
        // Get and store the local player subsystem
        auto EnhancedInputSubsystem =
     ULocalPlayer::GetSubsystem<UEnhancedInputLocal
     PlayerSubsystem>(GetLocalPlayer());

        // Then clear any existing mapping, then add the new
        ↪  mapping
        EnhancedInputSubsystem->ClearAllMappings();
        EnhancedInputSubsystem->AddMappingContext
     (InputMapping, 0);

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

        // Bind the inputs
        EnhancedInputComponent->BindAction(InputConfig->
     MoveXInput, ETriggerEvent::Triggered, this,
     &AController_Player::MoveX);
        EnhancedInputComponent->BindAction(InputConfig->
     MoveYInput, ETriggerEvent::Triggered, this,
     &AController_Player::MoveY);
        EnhancedInputComponent->BindAction(InputConfig->
     JumpInput, ETriggerEvent::Triggered, this,
     &AController_Player::Jump);
        EnhancedInputComponent->BindAction(InputConfig->
     InteractInput, ETriggerEvent::Triggered, this,
     &AController_Player::Interact);
        EnhancedInputComponent->BindAction(InputConfig->
     CameraInput, ETriggerEvent::Triggered, this,
     &AController_Player::RotateCamera);
        }
}
```

Finally, I implemented basic X/Y movement and camera rotation in the character.

2 Picking Up Objects

Commit

Next, I worked on picking up certain objects in the world, such as cubes to solve puzzles. I began by creating a prototype object to test any pickup scripts that I implement. It is simply a static mesh component with physics enables, allowing the player to push it around the world

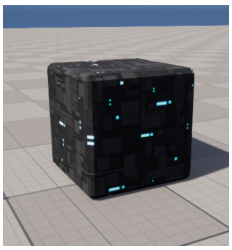

Figure 2. The PROTO WorldObject actor. It's simply a cube static mesh with physics enabled and its property set to

I implemented a system to pickup and attach the object to the player next. My first attempt was fire a trace, testing for objects with a "CanBePickedUp" GameplayTag (which currently is only the PROTO WorldObjects). If the trace hit an object with the tag, it is attached to a spring arm to keep it at a certain length.

```
TraceHit.GetActor()->AttachToComponent(PickUpSpringArm,
    FAttachmentTransformRules(EAttachmentRule::SnapToTarget,
    ↪  true), USpringArmComponent::SocketName);
ActorHeld = TraceHit.GetActor();
```

However, problems arose with this system. Firstly, for the objects to be movable in the world, they needed to have "SimulatePhysics" enabled which is not allowed when attached to character or other actors, which made them require their own base class to be casted to. Casts use additional processing power, so this system would be slightly ineffective.

My second implementation used Unreal's PhysicsHandleComponent to hold the object in place infront of the player. It started by also checking if the trace actor had a "CanBePickedUp" GameplayTag and if it did was 'grabbed' by the PhysicsHandleComponent. Dropping was also implemented, where the component

was checked to see if something was currently grabbed and if there was it was released.

```
void ACharacter_Parent::Interact()...
// Check if this character is currently holding an object
if (ObjectPhysicsHandle->GetGrabbedComponent()) {
        // If so, stop grabbing the component
        ObjectPhysicsHandle->ReleaseComponent();

}
else {
// Fire a line trace infront of this character
FHitResult TraceHit;
FCollisionQueryParams TraceParams;
TraceParams.AddIgnoredActor(this);

bool InteractTrace = GetWorld()->LineTraceSingleByChannel
 (TraceHit, GetActorLocation(), GetActorLocation() +
 (UKismetMathLibrary::GetForwardVector(FirstPersonCamera
 ->GetComponentRotation()) * PickUpArmLength),
 ECC_WorldDynamic, TraceParams);
DrawDebugLine(GetWorld(), GetActorLocation(),
  GetActorLocation() + (UKismetMathLibrary::GetForwardVector
  (FirstPersonCamera->GetComponentRotation()) *
  PickUpArmLength), FColor::White, true, 5, 0, 5);
// Check if the object hit has a tag of "CanBePickedUp"
if (TraceHit.GetActor()) {
if (TraceHit.GetActor()->ActorHasTag(FName("CanBePickedUp"))) {
        // If so, grab the actor via the ObjectPhysicsHandle
        // Calculate the object's component bounds to grab it in
        ↪  the center of the object
        ObjectPhysicsHandle->GrabComponentAtLocationWithRotation
     (TraceHit.GetComponent(), "", TraceHit.GetComponent()
     ->GetComponentLocation(), TraceHit.GetComponent()->
     GetComponentRotation());
```

This produced the results I was looking for - the object is held infront of the player at a fixed distance. However, when the player moved or rotated their camera, the box didn't follow as the PhysicsHandleComponent's target location did not dynamically update when the owning actor moved. This was fixed by updating it's target location whenever either an X input, Y input or Camera input was detected.

```
void ACharacter_Parent::MoveY(float AxisValue)...
if (ObjectPhysicsHandle->GetGrabbedComponent()) {
// Update the PhysicsHandles TargetLocation
ObjectPhysicsHandle->SetTargetLocationAndRotation(GetActorLocation()
    + (UKismetMathLibrary::GetForwardVector(FirstPersonCamera
    ->GetComponentRotation()) * PickUpArmLength),

    ↪  UKismetMathLibrary::FindLookAtRotation(ObjectPhysicsHandle->
    GetGrabbedComponent()->GetComponentLocation(),
    FirstPersonCamera->GetComponentLocation()));
```

The result was almost complete, except the box was never picked up at the center - rather one of the corners. I fixed this by calculating the box mesh's extents, then dividing them by 6 (as the mesh I used had a slight offset issue) and then using the final location when grabbed.

```
void ACharacter_Parent::Interact()...
// Calculate the object's component bounds to grab it in the
↪  center of the object
FVector BoxExtents;
BoxExtents =
↪  TraceHit.GetComponent()->GetLocalBounds().BoxExtent;
BoxExtents = BoxExtents / 6;
ObjectPhysicsHandle->GrabComponentAtLocationWithRotation
    (TraceHit.GetComponent(),
    "", TraceHit.GetComponent()->GetComponentLocation() +
    BoxExtents,
    TraceHit.GetComponent()->GetComponentRotation());
```

Finally, I tested the implementation with a collection of PROTO WorldObjects of varying mass and physics setting.

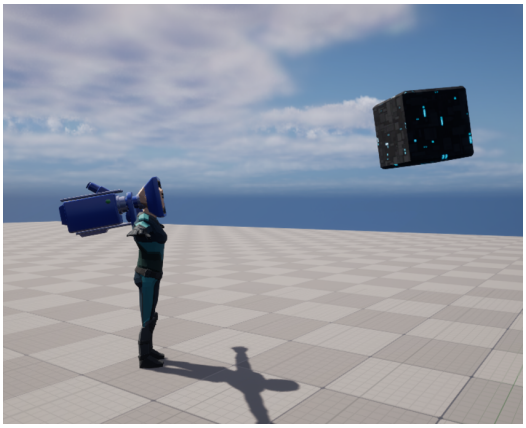

Figure 3. The PhysicsHandleComponent in action. The target location is only updated when the player moves, rather than on tick

3 Button and MechanicObject Parent

Commit

I started by updating how the player's forward movement is calculated. Previously, staring straight down would cause all forward movement to cease, as it was calculating from the FirstPersonCamera's ForwardVector (which was pointing at the floor). I fixed this by instead getting the scaled axis of the control rotation. The RotationMatrix' Z value is then set to 0, and a safe normal of this value is used in AddMovementInput.

```
void ACharacter_Parent::MoveX(float AxisValue)...
FVector Direction =
↪  FRotationMatrix(GetControlRotation()).GetScaledAxis(EAxis::X);
Direction = FVector(Direction.X, Direction.Y,
↪  0.0f).GetSafeNormal();
AddMovementInput(Direction, AxisValue, false);
```

Next, I started the implementation of Mechanic Objects. All mechanic objects are subclasses of the superclass MechanicObject Parent, where all similar features, components and properties are added (like usual inheritance). Firstly, I added both an active boolean - to denote if the MechanicObject is currently active - and an array to other MechanicObjects - allowing an input to link and trigger other objects.

Following this were the functions. The virtual functions StartTrigger and EndTrigger are called when the object's output is activated or deactivated and are overridden on each output to complete each objects induvidual task - open a door, turn a laser on etc. Next, the ModifyVisualElements function is implemented in Blueprints as it handles any visual element the object has. Finally, the test function BlueprintTestFunction is added to test if the trigger works with a PROTO object

```
UCLASS()
class STARMENDERS_API AMechanicObject_Parent : public AActor
{
        GENERATED_BODY()

public:
// Sets default values for this actor's properties
AMechanicObject_Parent();

// Called every frame
virtual void Tick(float DeltaTime) override;

/// -- Trigger Function --
// Called when this object recieves it's input
virtual void StartTrigger();

// Called when this object stops recieving it's input
virtual void EndTrigger();

// Called to modify the visual elements of this object
UFUNCTION(BlueprintImplementableEvent)
void ModifyVisualElements(bool bSetOn);

// Blueprint Test Function -- ONLY USE FOR TESTING
UFUNCTION(BlueprintImplementableEvent)
void BlueprintTestFunction();

public:
/// -- Global Components --
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
↪  "Components")
USceneComponent* Root = nullptr;

/// -- Mechanic Properties --
// FString denoting the objects name
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
↪  "Mechanic Properties")
FString ObjectName;

// Boolean denoting if the mechanic is active or not
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
↪  "Mechanic Properties")
bool bActive = false;

/// -- Outputs --
// TArray of pointers to other MechanicObjects
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
↪  "Outupts")
TArray<AMechanicObject_Parent*> OutputsObjects;
};
```

The button was my next target. It consists of two static meshes, one for the button base and the other for the button inner, and a BoxComponent trigger zone. When an object with the GameplayTag of "CanActivateButtons", it is added to a OverlapActors array. If the button has more than one object in this array, it is activated. The opposite happens when an actor leaves the zone - it is checked for the tag and removed from the array if it does.

```
void AMechanicObject_Button::OnButtonBeginOverlap
    (UPrimitiveComponent* OverlappedComp, AActor*
    OtherActor, UPrimitiveComponent* OtherComp,
    int32 OtherBodyIndex, bool bFromSweep,
    const FHitResult& SweepResult)
{
// Check if the overlapping actor has the tag
↪  "CanActivateButtons"
if (OtherActor->ActorHasTag("CanActivateButtons")) {
        OverlapActors.Add(OtherActor);

        // If OverlapActors now has exactly one index, activate
        ↪  the button
        if (OverlapActors.Num() == 1) {
                StartTrigger();
        }
}
}

void AMechanicObject_Button::OnButtonEndOverlap
    (UPrimitiveComponent* OverlappedComp,
    AActor* OtherActor, UPrimitiveComponent*
    OtherComp, int32 OtherBodyIndex)
{
// Check if the overlapping actor has the tag
↪  "CanActivateButtons"
if (OtherActor->ActorHasTag("CanActivateButtons")) {
        OverlapActors.Remove(OtherActor);

        // If OverlapActors now is empty, deactivate the button
        if (OverlapActors.Num() == 0) {
                EndTrigger();
        }
}
}
```

Finally, I created a prototype BluePrint object to check the button's output. When triggered, the light of the object changes from red to yellow and vice versa. I followed by adding the gameplay tag to the WorldObjects, and they successfully trigger the button alongside the player.

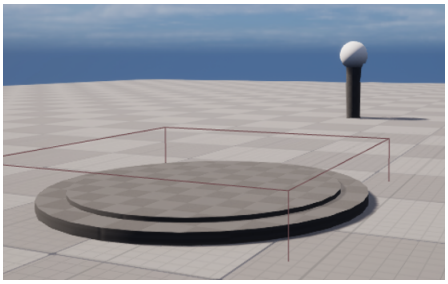

Figure 4. The Button and the prototype output. When triggered by standing on the button, the light changes colour

4 Doors and Advanced MechanicObjects

Commit

Improvements to the MechanicObjects were my next target. My first goal was to make it easier to denote what state an object is in - either Off (where it cannot produce an output), On (can produce an output but currently isn't) and Active (is currently producing an output). To achieve this, I created an enum named EObjectState and as only MechanicObjects and classes that use/interact with them need the enum, I defined it in the Parent class.

```
UENUM(BlueprintType, Category = "Mechanics")
enum EObjectState
{
        Off UMETA(DisplayName = "Off"),
        On UMETA(DisplayName = "On"),
        Active UMETA(DisplayName = "Active")
};
```

I also simplified how objects are triggered, with the TriggerRequirements and TriggerCount int properties. When TriggerCount equals or exceeds TriggerRequirement, the state of the object changes from Off to On. This allows a requirement of multiple inputs to be activated, such as two buttons, to use an object, such as a door. TriggerCount is visible but read only in BluePrints, while TriggerCount is read and write.

Next, I added the ActivateObject function, called when other objects activate an object. After incrementing or decrementing TriggerCount, it checks if the TriggerCount has now reached or exceeded the TriggerReqirements. If they have StartTrigger is called, while EndTrigger is called when TriggerCount is less than the requirement. It also updates the object state.

```
void AMechanicObject_Parent::ActivateObject(bool bPositive)
{
if (bPositive) {
        TriggerCount++;
}
else {
        TriggerCount--;
}

// Check if the TriggerCount has now reached the
↪  TriggerReqirements
if (TriggerCount >= TriggerRequirement) {
        ObjectState = EObjectState::On;
        StartTrigger();
}
else {
        ObjectState = EObjectState::Off;
        EndTrigger();
}
}
```

As each object could have diifferent requirements for activation, ActivateObject is a virtual function to allow overrides. For example, the MechanicObject Button checks to see if the OverlapActors array is checked to see if it has any indices. If it does, the button is activated (as it would be if an object was placed on it when it was on). On the other hand, when TriggerCount is lower than TriggerRequirement the OverlapActors array is checked again to see if it is not equal to 0 (aka something is triggering it). If something is, the button is cleared of being active.

```
void AMechanicObject_Button::ActivateObject(bool bPositive)
{
if (bPositive) {
        TriggerCount++;
}
else {
        TriggerCount--;
}

// Check if the TriggerCount has now reached the
↪  TriggerReqirements
if (TriggerCount >= TriggerRequirement) {
        // Check if the object is currently being triggered.  If
        ↪  so, StartTrigger
        if (OverlapActors.Num() == 1) {
                StartTrigger();
```

```
        }
        ObjectState = EObjectState::On;
}
else {
        if (OverlapActors.Num() != 0) {
                EndTrigger();
        }

        ObjectState = EObjectState::Off;
}
}
```

Finally, I created a simple door object which acts similar to the button. When TriggerCount equals or exceeds TriggerRequirment, the StaticMesh of the door is hidden.

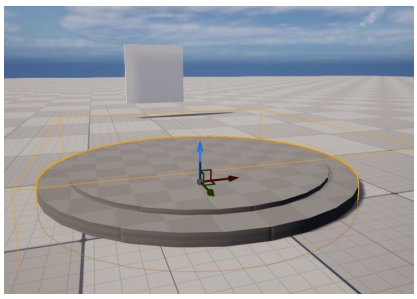

Figure 5. The MechanicObject Door object. When triggered by standing on the button, the door opens

5 UI Start and RecordPad

Commit

The main mechanic - recording movements and playing them back - was my next aim. However before saving the player's actions, I had to make a way to save them to an object so they could be played back later. I implemented the RecordPad class to fulfil this task. It consists of two static meshes - one inner and one outer mesh - one skeletal mesh - a t-posing player character model - and a BoxComponent.

Figure 6. The RecordPad object. The BluePrint has an additional component - a RotatingMovementComponent - to make the skeletal mesh rotate

Next, I added overlap events to the BoxComponent to check when the player stands on the pad. When they do, the player's CurrentRecordPad pointer is updated and the skeletal mesh is no longer rendered. Once the player steps off the pad, the pointer is set to nullptr and the mesh is rendered again.

```
void ARecordPad::OnRecorderBeginOverlap()
{
if (OtherActor->IsA(ACharacter_Parent::StaticClass())) {
        bPlayerOverlapping = true;
        Cast<ACharacter_Parent>(OtherActor)->
     SetCurrentRecordPad(this);
        ToggleHologramVisibility();
}}

void ARecordPad::OnRecorderEndOverlap()
{
if (OtherActor->IsA(ACharacter_Parent::StaticClass())) {
        bPlayerOverlapping = false;
        Cast<ACharacter_Parent>(OtherActor)->
     SetCurrentRecordPad(nullptr);
        ToggleHologramVisibility();
}}
```

Additionally, I added a TMap with keys of type int and values of LinearColor, to act as colours to differentiate each record pad currently in the world - instead of them all being white. The meta of the property is set to EditDefaultsOnly so they can only be set in the BluePrint class and not in the level itself, keeping the values identical between each pad. I also added a BluePrintImplementableEvent function SetupVisualElements, which updates the colour of the hologram based on the index.

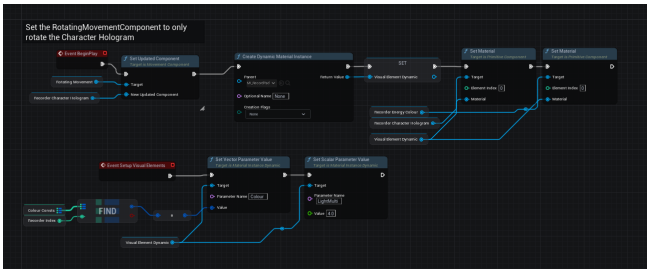

Figure 7. The SetupVisualElements BluePrint event. It simply finds the colour in the TMap that matches the index and updates the colour parameter in the dynamic material

Next, I worked on how the player would interact with the RecordPad. I decided on a WidgetComponent on the player class instead of having a UI added to the view port as I wanted the UI to be as diegetic as possible - the player would have a wrist pad that they could press buttons on in a later version. I had implemented a similar system in my GalaxyExplorer project previously, so I had some experience in the area.

First, I created the UserWidget class. Before, I would define each element in the C++ class, as well as any functionality but link the buttons to the functions in BluePrint. However, I have learned that this is not necessarily the best practice, as well as C++ has the ability to bind the events to the button functions via AddDynamic.

```
void UInGame_Master::NativeConstruct()
{
Super::NativeConstruct();

// Setup Button Binds for the Play Button
PlayButton->OnHovered.AddDynamic(this,
 &UInGame_Master::OnPlayButtonHovered);
PlayButton->OnReleased.AddDynamic(this,
 &UInGame_Master::OnPlayButtonReleased);

// Setup Button Binds for the Record Button
RecordButton->OnHovered.AddDynamic(this,
 &UInGame_Master::OnRecordButtonHovered);
RecordButton->OnReleased.AddDynamic(this,
 &UInGame_Master::OnRecordButtonReleased);

// Setup Button Binds for the Delete Button
DeleteButton->OnHovered.AddDynamic(this,
 &UInGame_Master::OnDeleteButtonHovered);
...
}
```

Next, I created the BluePrint class for the UI. It consists of three buttons - Play, Delete and Record - as well as a Text Box explaining what each button does. Each text state is stored in a map and is searched for when a button is overlaped.

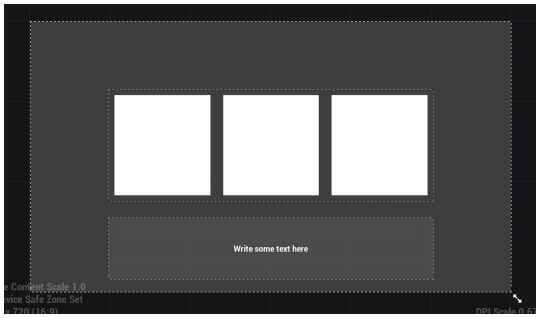

Figure 8. The The InGame Master BluePrint. When a button is hovered the text box is updated, or cleared when no button is hovered

Finally, before connecting the UI to the RecordPad, I added the UI to the player character via a WidgetComponent, while also adding a WidgetInteractionComponent which allows the player's mouse to click and use widgets in the world via PressPointerKey and ReleasePointerKey. I also added an input that allows the player to toggle the visibility of the menu on the widget, which simply changes the visibility of the widget.

```
void ACharacter_Parent::ToggleMenu(bool bInMenu)
{
        MenuWidgetComponent->SetVisibility(bInMenu);
        bMovementDisabled = bInMenu;
}
```

I also added a getter and setter function for CurrentRecordPad, which removes the need of other classes directly updating the pointer, as well as fixed a few errors I added - such as updating the FirstPersonCamera's Z axis instead of the X axis and the ObjectPhysicsHandle now uses the FirstPersonCamera's ComponentLocation rather than the ActorLocation of the character.

6 Recording

6a - Prototyping

Commit

Before implementing recording in the C++ class, I prototyped the recording and playback systems in

BluePrint, with my first aim recording X movement. Originally, I was going to simply use Tick with it's DetlaTime output to record the direction of the Recording character on each frame. However, if the FPS would fluctuate the character would be recorded wrong. For example, if the recording was captured at 60fps but played back at 30, the character would be much slower in the playback. Additionally, this wouldn't allow for animations to play easily as the playback character would just be teleported on each frame.

My first attempt was to utilize the EnhancedInput's Input Action Instance to get the elapsed time of the input being fired and store how long each input was held down for in an array. However, this didn't allow a value to be recorded easily as certain inputs - such as MoveX - handle multiple directions - such as forward and backwards.

Instead, I created a side tick system that uses two timers to capture any input active. RecordingTick would loop on a very short duration cycle, while RecordingTotal would manage the total time of the recording and clear both timers when the time was up. Both the recording and playback will use a timer with the same 'tick' rate to maintain the speed of the character, implemented in the Character Parent.

```
/// -- Recording --
// Timer handle for the recording tick
FTimerHandle RecordingTickHandle;

// Timer handle for the total recording time
FTimerHandle RecordingTotalHandle;
```

I implemented a child class of the Default character next, which is to be spawned and possessed when recording starts. I created the FRecordingData structure to hold the recording information, consisting of a float Value and float Tick which holds the tick when it was recorded, and added a TArray of the structs to the new Recording character alongside a boolean denoting if the input is active.

```
// Struct holding the data of an input recording
USTRUCT(BlueprintType, Category = "Recording")
struct STARMENDERS_API FRecordingData
{
public:
        GENERATED_BODY();

/// -- Recording Data --
// The value of the input at the tick
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
↪  "Recording Data")
float Value;

// The tick associated with this value
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
↪  "Recording Data")
float Tick;

public:
FRecordingData();
FRecordingData(float NewValue, float NewTick);
~FRecordingData();
};

class STARMENDERS_API ACharacter_Record...
/// -- Input Booleans --
// Bool for each input to denote if the input is currently being
↪  pressed
bool bMoveXActive = false;

// -- RecordedInputs --
// TArray storing all recorded MoveX inputs
UPROPERTY(VisibleAnywhere, BlueprintReadOnly);
TArray<FRecordingData> MoveXRecording;
```

In the constructor, Character Record finds and stores the input context data identically to the Controller Player class. This data is used to setup inputs specific to this character, which binds to the inputs Started and Complete state, where these are fired when the input is pressed and again when it it released. When an input is started, the respective boolean is set to true. When the input is complete, the boolean is set to false.

```
void ACharacter_Record::RecordMoveX(const FInputActionValue&
↪  Value)
{
// Check if this is either the start or end of the input
// If bMoveXActive is true at this stage, the input has been
↪  completed
if (bMoveXActive) {
        bMoveXActive = false;
        UE_LOG(LogTemp, Warning, TEXT("MoveX End"));
}
else {
        bMoveXActive = true;
        UE_LOG(LogTemp, Warning, TEXT("MoveX Start"));
}
}
```

While recording, each time RecordingTick is called each input bool is checked to see if they are true. If they are, a new index is added to the respective TArray with the input's value and the current tick time.

```
void ACharacter_Record::RecordingTick()
{
// Record each of the active inputs at this tick
// Record MoveX tick
if (bMoveXActive) {
        MoveXRecording.Add(FRecordingData(
        ↪  MoveXBind->GetValue().
```

```
    Get<float>(), CurrentTickTime));
}

// Finally, increment CurrentTickTime
CurrentTickTime += TimerTickRate;
}
```

Finally, I did some cleanup on my previous classes. I converted from a single UI class to a master class that is added to the player with different states, and moved all recording assets to InGame Default and added an InGame Default to the master class' widget switcher. Additionally, I fixed an error I implemented by accident and moved MenuUI setup from Tick to BeginPlay in the Parent Character and made the MoveX, MoveY and Interact input functions virtual, so they could be overriden by the Playback class when implemented.

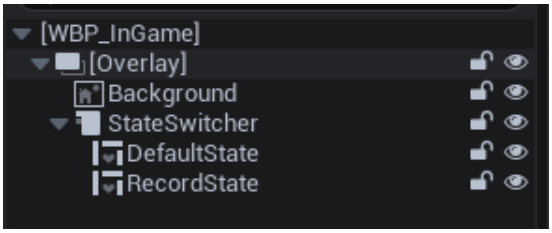

Figure 9. The Updated InGame Master Heirarchy. Now, each state of the UI is contained in it's own BluePrint class and added to the WidgetSwitcher

6b - Recording all inputs

Commit

Next up was adding the ability to record all inputs. This began by creating functions and booleans for each input currently in use and then updating the structs to hold additional data. First up was MoveY, which was implemented in the exact same way as MoveX was previously. Following this was both Interact and Jump. As these inputs don't require an associated value and only a time they were pressed, they are simply added to the array based on their tick time. They also only have one input bind, that being when they are Started

```
void ACharacter_Record::SetupPlayerInputComponent()...
EnhancedInputComponent->BindAction(InputConfig->JumpInput,
 ETriggerEvent::Started, this, &ACharacter_Record::RecordJump);
EnhancedInputComponent->BindAction(InputConfig->InteractInput,
 ETriggerEvent::Started, this,
 ↪  &ACharacter_Record::RecordInteract);
```

```
void ACharacter_Record::RecordJump()
{
JumpRecording.Add(CurrentTickTime);
}

void ACharacter_Record::RecordInteract()
{
InteractRecording.Add(CurrentTickTime);
}
```

Camera Recording was a bit more difficult to implement. Instead of using a float input it instead uses a FVector2D for both X and Y rotation. To handle this, I created a new struct FRecordingDataVector, which is identical to FRecordingData except has a FVector2D Value instead of a Float one.

```
// Struct holding the data of an input FVector recording
USTRUCT(BlueprintType, Category = "Recording")
struct STARMENDERS_API FRecordingDataVector
{
public:
GENERATED_BODY();

/// -- Recording Data --
// The value of the input at the tick
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
↪  "Recording Data")
FVector2D Value;

// The tick associated with this value
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
↪  "Recording Data")
float Tick;

public:
FRecordingDataVector();
FRecordingDataVector(FVector2D NewValue, float NewTick);
~FRecordingDataVector();
};
```

Finally, I updated RecordingTick to capture the values of both the MoveY and Camera Inputs.

```
ACharacter_Record::RecordingTick()...
if (bMoveYActive) {
        MoveYRecording.Add(FRecordingData(MoveYBind->GetValue().
     Get<float>(), CurrentTickTime));
}

if (bCameraActive) {
        CameraRecording.Add(FRecordingDataVector(MoveYBind->GetValue().
     Get<FVector2D>(), CurrentTickTime));
}
```

6c - RecordingData Class

Commit

I simply moved the Recording Structs from the Character Record to a separate class - RecordingData - and renamed FRecordingData to FRecordingDataFloat instead. Any class that requires the RecordingData structs can now include RecordingData rather than Character Record now.

6d - Start and End Record

Commit

To start, I created the struct FRecordingData that combines all of the input arrays into one structure that will be used to pass the recording between classes. I implemented an additional constructor that takes in an array for each input and sets the values up correctly.

```
// Struct holding the data of an recording
USTRUCT(BlueprintType, Category = "Recording")
struct STARMENDERS_API FRecordingData
{
public:
        GENERATED_BODY();

/// -- Recording Data --
// TArray storing all recorded MoveX inputs
UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category =
↪  "Recording Data");
TArray<FRecordingDataFloat> MoveXRecording;

UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category =
↪  "Recording Data");
TArray<FRecordingDataFloat> MoveYRecording;

UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category =
↪  "Recording Data");
TArray<float> JumpRecording;

UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category =
↪  "Recording Data");
TArray<float> InteractRecording;

UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category =
↪  "Recording Data");
TArray<FRecordingDataVector> CameraRecording;

public:
FRecordingData();
FRecordingData(TArray<FRecordingDataFloat> NewMoveXRecording,
 TArray<FRecordingDataFloat> NewMoveYRecording,
 TArray<float> NewJumpRecording, TArray<float>
 ↪  NewInteractRecording,
 TArray<FRecordingDataVector> NewCameraRecording);
~FRecordingData();
};
```

Next, I added a new input to the Controller Player which is used to interact with the WidgetComponent on the Characters. As stated earlier in Part 5, to interact with widgets in the world characters require a WidgetInteractionComponent, which uses PressPointerKey and ReleasePointerKey to press buttons.

```
void AController_Player::SetupInputComponent()...
EnhancedInputComponent->BindAction(InputConfig->UIInteractInput,
 ETriggerEvent::Triggered, this,
 ↪  &AController_Player::UIInteract);

void AController_Player::UIInteract(const FInputActionValue&
↪  Value)
{
if (Character) {
        GetActiveCharacter()->UIInteract(bInMenu);
}
}
```

Next I worked on starting the recording, which starts by spawning the Character Record. However, before I could possess the character I had to slightly rework the Controller Player. Currently, each input checks to see if the Player pointer is valid before calling a function with that pointer. I could easily update this pointer with the new Character Record - as they are both subclasses of Character Parent - but this would make it difficult to re-possess the Character Default on recording end.

To bypass this, I added a new pointer to the Character Record which will be set when the player starts recording. Following this, I added a new function inside of Controller Player that determines if the player is controlling the Character Record or the Character Default and updated each input to use the output of the function.

```
ACharacter_Parent* AController_Player::GetActiveCharacter()
{
if (RecordingCharacter) {
        return RecordingCharacter;
}
return Character;
}

void AController_Player::MoveX(const FInputActionInstance&
↪  Instance)
{
if (Character) {
        GetActiveCharacter()->MoveX(Instance.GetValue().Get<float>());
}}
```

Finally, I override the OnPossess function to check if the InPawn is either a Character Record or a Character Default. If the InPawn argument is a recording character, the pointer is updated and the Character Default is hidden. I also implemented a function which is called to re-possess the Character Default.

```
void AController_Player::OnPossess(APawn* InPawn)
{
Super::OnPossess(InPawn);

// Check if the pawn is a Character_Record.  If so, cast and set
↪  the pointer to it
if (InPawn->IsA(ACharacter_Record::StaticClass())) {
        RecordingCharacter = Cast<ACharacter_Record>(InPawn);
        bInMenu = false;
        bShowMouseCursor = false;

        // Also hide the main character
        Character->SetActorHiddenInGame(true);
        Character->SetActorEnableCollision(false);
}
else if (InPawn->IsA(ACharacter_Parent::StaticClass())){
        RecordingCharacter = nullptr;

        if (!Character) {
                // Cast to the character pawn and store it
                Character = Cast<ACharacter_Parent>(GetPawn());
        }

        Character->SetActorHiddenInGame(false);
        Character->SetActorEnableCollision(true);
}}

void AController_Player::RePossessCharacter()
{
this->Possess(Character);
}
```

Next was the UI inputs. When the Record button is pressed, StartRecording is called on the CurrentRecordPad - the record pad that the player is standing on. If CurrentRecordPad is nullptr, nothing happens.

```
void UInGame_Default::OnRecordButtonReleased()
{
// Call StartRecord on the assocciated pad
if (MasterUI->GetPlayerOwner()->GetCurrentRecordPad()) {
        MasterUI->GetPlayerOwner()->GetCurrentRecordPad()

    ↪  ->StartRecording(MasterUI->GetPlayerOwner()->GetController());
        }
}
```

When StartRecording is called, the RecordPad spawns a Character Record at it's location, which is then possessed by the Controller Player and StartRecording is called on the new character - which stores the RecordPad, resets the CurrentTickTime and starts the two timers.

Figure 10. The Character Record. It is spawned under the map, then moved to the RecordPad's location to make sure it always spawns

After RecordingTotal reaches it's full duration, both timers are cleared and the actor is destroyed. However, this leads to the player controlling a dead character and the recording lost. To combat this, I added the SetRecoring function in the RecordPad. It simply stores the record from the character, calls the RePossess function on the controller and moves the Character Default to the RecordPad's location.

```
void ACharacter_Record::EndRecording() {
        // Clear the timer handles
        GetWorld()->GetTimerManager().ClearTimer(RecordingTickHandle);
        GetWorld()->GetTimerManager().ClearTimer(RecordingTotalHandle);

        // Pass the recording to the recording pad associated
        ↪  with this recording character
        OwningRecordPad->SetRecording(FRecordingData(MoveXRecording,
        ↪  MoveYRecording, JumpRecording, InteractRecording,
        ↪  CameraRecording), GetController());

        // Then destory this actor
        Destroy();
}

void ARecordPad::SetRecording(FRecordingData NewRecord,
↪  AController* PlayerController)
{
Record = NewRecord;
RecordPresent = true;

// Cast to the controller and RePossess the original character
AController_Player* PC =
↪  Cast<AController_Player>(PlayerController);
PC->RePossessCharacter();
PC->GetPawn()->SetActorLocation(GetActorLocation() +
↪  FVector(0.0f, 0.0f, 95.0f));
}
```

6e - Recording UI and More

Commit

As the Character Record has different UI requirements to the Character Default, I created a new UI state that will be used when recording. It simply consists of a button - where the button simply calls EndRecording early to end the recording before all of the time is used - and a text block - which updates on timer tick to display the time remaining. I added it to the InGame Master widget as a new state, as well as implemented a function to change the state of the WidgetSwitcher.

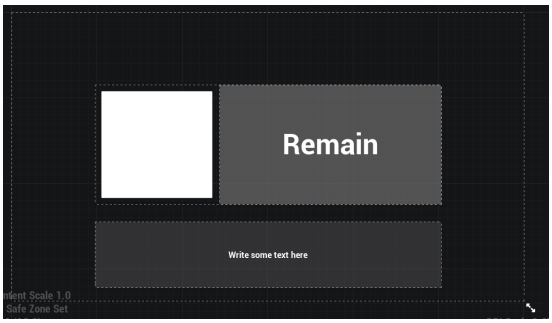

Figure 11. The InGame Recording UI State. As stated above, the text block updates to show the remaining time left to record in seconds

```
void UInGame_Recording::UpdateRemainingTime(float
↪  InTimeRemaining)
{
int a = InTimeRemaining;
a += 1;
RemainingTimeText->SetText(FText::FromString(FString::Printf(TEXT("%i")
↪  a)));
}
```

Next was a clean up of the code implemented in this version. First off, I added a function in RecordPad to clear any recording that exists, as well as updated the InGame Default to call this function when the correct button is pressed. Additionally, I updated the Record Character spawn to use a TSubclassOf pointer instead of a StaticClass, as this allows BluePrint classes to be spawned instead.

```
// Pointer to the recording character class blueprint
UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category =
↪  "Recorder Properties")
TSubclassOf<class ACharacter_Record> RecordingCharacterClass =
↪  nullptr;
```

Previously, when a new pawn was possessed, the controller still acted as it was in a menu. To counteract this, ToggleMenu is now called when possessing a new pawn. Additionally, as only Character Default used GetCurrentRecordPad and SetCurrentRecordPad I moved to the Default class from the Parent class.

```
ARecordPad* ACharacter_Default::GetCurrentRecordPad()
{
        return CurrentRecordPad;
}

void ACharacter_Default::SetCurrentRecordPad(ARecordPad*
↪  NewRecordPad)
{
        CurrentRecordPad = NewRecordPad;
}
```

Finally, the UI of the Record character is updated in it's BeginPlay, and the UI's UpdateRemainingTime is called on each recording tick.

```
void ACharacter_Record::BeginPlay()...
MenuUI->UpdateActiveState("Recording");

void ACharacter_Record::RecordingTick()...
MenuUI->RecordState->UpdateRemainingTime
 (MaximumRecordingTime - CurrentTickTime);
```

7 Playback

7a - Character

Commit

Next was the implementation of the Playback Character. The playback character is spawned on every recording pad that has a recording saved but is not in use by the player. The main aim for the character is to take the data stored on a record pad and convert it back into movement and inputs. But first I had to setup the playback tick timer and total timer, which required calculating the total time needed by finding the last tick value from all of the inputs.

My first attempt used a c++ array to store all of the last indexes from each input array, then using the max-element function to find the largest element. However, inserting the data into the array was proving difficult so this was scrapped.

```
// Make a temporary index of all the arrays final indecies
float a[] = {Playback.MoveXRecording.Last().Tick,
Playback.MoveYRecording.Last().Tick,
Playback.JumpRecording.Last(),
Playback.InteractRecording.Last(),
Playback.CameraRecording.Last().Tick};
```

My next attempt used a UE4 TArray instead, staring by inserting each of the last index tick from each input into the array. However, if the input array is empty, a default value is entered instead. Next, the array is iterated over to find the largest value, which is then used to set the PlaybackTotal timer's duration.

```
TArray<float> b;
if (Playback.MoveXRecording.IsEmpty()) {b.Insert(0, 0);}
 else {b.Insert(Playback.MoveXRecording.Last().Tick, 0); };
if (Playback.MoveYRecording.IsEmpty()) { b.Insert(0, 1); }
 else { b.Insert(Playback.MoveYRecording.Last().Tick, 1); };
if (Playback.JumpRecording.IsEmpty()) { b.Insert(0, 2); }
 else { b.Insert(Playback.JumpRecording.Last(), 2); }
if (Playback.InteractRecording.IsEmpty()) { b.Insert(0, 3); }
 else { b.Insert(Playback.InteractRecording.Last(), 3); };
if (Playback.CameraRecording.IsEmpty()) { b.Insert(0, 4); }
 else { b.Insert(Playback.CameraRecording.Last().Tick, 4); };

// Compare each index and store the largest value
for (float c : b){
 if (c > MaximumPlaybackTime) {
  MaximumPlaybackTime = c;
}}
```

Next, I implemented the PlaybackTick function. It simply works in reverse of the recording - on each tick each array's first index is checked to see if it is greater than or equal to the current tick time. If it is, the corresponding input is called and the index is removed from the array

```
// Check if MoveX[0].Tick == CurrentTick.  If so, call MoveX
↪  with the value and remove index 0
if (!Playback.MoveXRecording.IsEmpty()) {
 if (Playback.MoveXRecording[0].Tick <= CurrentTickTime) {
  MoveX(Playback.MoveXRecording[0].Value);
  Playback.MoveXRecording.RemoveAt(0);
}}

// Repeat for each input (MoveY, Interact, Camera and Jump)
```

However, this caused some issued for RotateCamera. As they use the control rotation of the class instead of the camera's rotation, the character would not rotate as their controller didn't exist. This required a small rework to how the camera input was recorded by the Character Record, now instead storing the control rotation instead of the input values. Character Playback then takes this value, updates their camera's pitch and yaw while clearing roll and finally sets the actor's yaw.

```
void ACharacter_Record::RecordingTick()...
if (bCameraActive) {
 CameraRecording.Add(FRecordingDataVector
(FVector2D(GetControlRotation().Pitch,
GetControlRotation().Yaw), CurrentTickTime));
}

void ACharacter_Playback::RotateCamera(FVector2D AxisValue){
if (!bMovementDisabled) {
// Update the FirstPersonCamera with the new pitch
 FirstPersonCamera->SetWorldRotation(FRotator(AxisValue.X,
 ↪  AxisValue.Y, 0.0f));
 SetActorRotation(FRotator(0.0f, AxisValue.Y, 0.0f));
}
```

The next aim was to spawn the character when the player starts recording with a different pad. This required setting up the LevelController and updating how the UI and the characters react with the record pads.

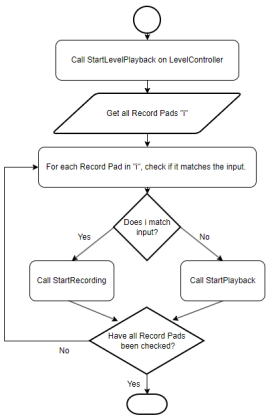

Figure 12. The new path to Playback. Instead of directly accessing the record pad, when the player starts recording they call StartLevelPlayback on the LevelController

First, I implemented the StartLevelPlayback and EndLevelPlayback functions. StartLevelPlayback takes in a RecordPad pointer argument and, when called, compares all pointers in the RecordPad array with the inputted parameter. If they match, then StartRecording is called on that RecordPad. Else, StartPlayback is called instead. EndLevelPlayback simply calls EndPlayback on all RecordPads in the array. Additionally UpdatePlayerUIElements was removed, as this is done via the UI class itself.

```
void ALevelController::StartLevelPlayback(ARecordPad*
↪  PadToRecordOn, ACharacter_Parent* Character)
{
// Check each stored record pad pointer matches
```

```
// the pointer inputted.  If so, start recording on that pad
// Else, start that pads playback
for (ARecordPad* i : RecordPads) {
 if (i == PadToRecordOn) {
  i->StartRecording(Character->GetController());
}
else {
 i->StartPlayback();
}}}

void ALevelController::EndLevelPlayback()
{
for (ARecordPad* i : RecordPads) {
 i->EndPlayback();
}}
```

Next, I updated the OnRecordButtonReleased function in the InGame Master UI state to now find the LevelController in the world and call StartLevelPlayback instead of calling StartRecording in the RecordPad directly.

```
void UInGame_Default::OnRecordButtonReleased()...
// Call StartRecord on the assocciated pad
if (DefaultCharacter->GetCurrentRecordPad()) {
  // Find the LevelController in the world
  Cast<ALevelController>(UGameplayStatics::GetActorOfClass
  (GetWorld(), ALevelController::StaticClass()))->
  StartLevelPlayback(DefaultCharacter->GetCurrentRecordPad(),
  DefaultCharacter);
        }
```

Finally, I did some general cleanup, including: changed MaximumRecordingTime from 5 seconds to 60 seconds in Character Record, changed TimerTickRate from 0.1 to 0.005 in Character Parent and added PlaybackCharacterClass property, used by the BluePrint class to set what ACharacterPlayback should be used for playback in the RecordPad.

7b - WorldObjects

Commit

My next aim was to update the current objects that can be picked up to their own separate class so characters can grab them from other characters. This was simple enough, only requiring a root component alongside a static mesh and a pointer to the object currently grabbing it.

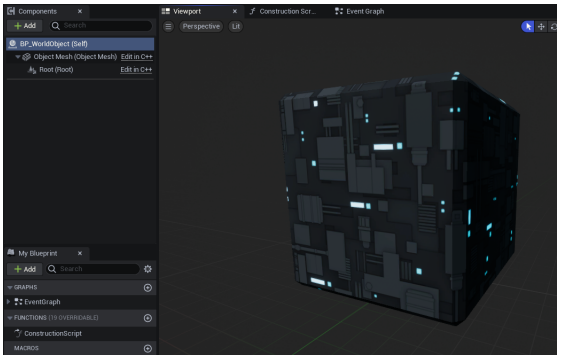

Figure 13. The new WorldObject class. The PROTO WorldObject BluePrint class was updated to be a subclass of WorldObject

Following this, I updated the Interact function in the Character Parent to now check if the trace has hit the WorldObject class. If so, checks if the object is currently being grabbed and if that is true, the object is ungrabbed and then regrabbed by the new character.

```
void ACharacter_Parent::Interact()...
// Check if the object hit has a tag of "CanBePickedUp"
if (TraceHit.GetActor()) {
 if (TraceHit.GetActor()->ActorHasTag(FName("CanBePickedUp")))
 ↪  {
  // Next, check if the object is currently being grabbed.
  // Cast to the WorldObject, call GetCurrentGrabber and release
  ↪  the object from the grabber if not nullptr
  AMechanic_WorldObject* CG =
  ↪  Cast<AMechanic_WorldObject>(TraceHit.GetActor());
  if (CG->GetCurrentGrabber()) {
   CG->GetCurrentGrabber()->
   ObjectPhysicsHandle->ReleaseComponent();
}
```

8 Levels

8a - Portal Doors

Commit

My next aim was to get level generation implemented.

Design wise, I wanted players to summon the level from the level select, which is then spawned in the world removing the need for loading screens and adding to the effect that the level is generated on the ship. Additionally, I wanted players to enter the level from door A to the start of the level at door B, then use the door C to exit the level teleporting them to door

A. This required a portal system to be implemented, which I tackled first.

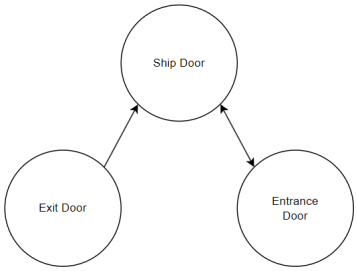

Figure 14. Level Entrance/Exit theory. The player is teleported between the level and the ship hub

First, I setup the components. The door is made up of two static meshes - one for the frame and one for the display plane - a BoxCollision - used to detect if the player is overlapping the portal - and a SceneCapture2D - to capture the view of the other portal door.

On my first attempt, I calculated the angle that the camera needed to rotate by calculating the dot product between the player's location and the portal's location, inverted the output rotation then updating the paired portal's spring arm. This provided a portal that updated the capture camera's location correctly. However, this didn't provide the full effect I was looking.

```
/// Original experimentation with DotProd, worked on rotation
↪  but didnt look 100% correct
// Calculate the angle
// If so, then get the player's forward vector and normalize it
FVector VecA = GetWorld()->GetFirstPlayerController()->
 GetPawn()->GetActorForwardVector();
VecA.Normalize();

// Next, get the door's forward vector and normalize it
FVector VecB = GetActorForwardVector();
VecB.Normalize();

// Calcualate the dot product between the two vectors and
↪  convert it back to degrees
float OutAngle =
↪  FMath::RadiansToDegrees(acosf(FVector::DotProduct(VecA,
↪  VecB)));

// Calculate the cross product of the angle
if (FVector::CrossProduct(VecA, VecB).Z > 0)
{
 OutAngle = -OutAngle;
}

// Finally, add the doors actual rotation and update the angle
↪  of the spring arm
OutAngle = OutAngle + GetActorRotation().Yaw;
PairedDoor->CaptureCamera->SetWorldRotation(FRotator
 (0.0f, OutAngle + GetActorRotation().Yaw, 0.0f));
```

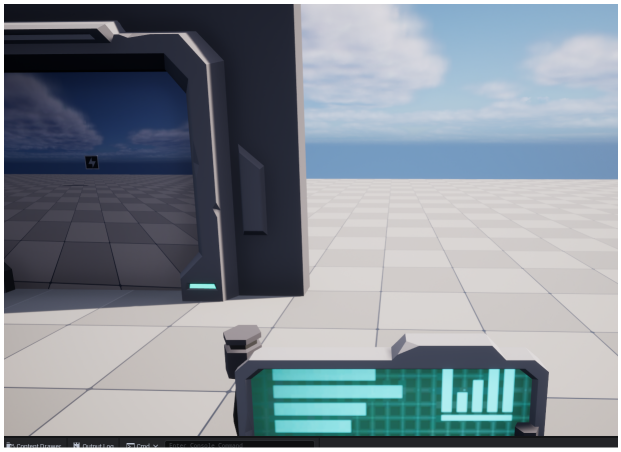

Figure 15. Portal Attempt One. The camera rotation updated correctly on the horizontal axis, but the location of the camera couldn't be calculated accurately

From this, I researched other developers creations to find the effect I was looking for and came across Dev Squared's series on "How to Create Portals in Unreal Engine". In their series, they utilized BluePrint methods but I converted them into C++. However, I wanted to learn and understand how the math works instead of just coping what they created, so I did further research into what each part does.

First, the location of the player's camera is converted into local space via InverseTransformPosition based on the portal's location, while also inverting the scale of the portal's transform so the output vector is inverted. This is then re-converted back to the paired portal's world space via TransformPosition, then used to update the CaptureCamera's world location.

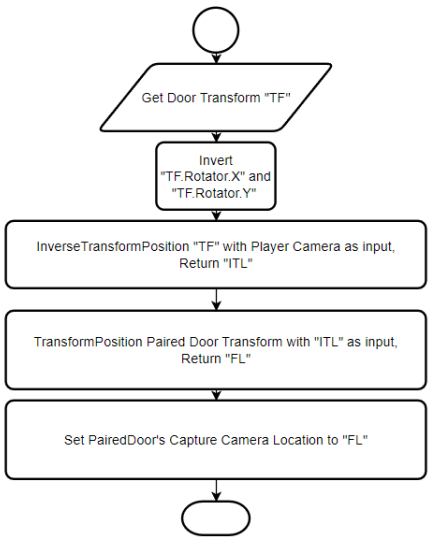

Figure 16. Updating the capture camera's location. The location of the camera is updated based on the location of the player to the paired portal

InverseTransformPosition takes a transform and converts it into a local space vector, while TransformPostion does the opposite and converts a transform to a world space vector.

```
// Caluclate the distance
 FTransform TF = GetActorTransform();
TF.SetScale3D(FVector(TF.GetScale3D().X * -1,
 TF.GetScale3D().Y * -1, TF.GetScale3D().Z));
FVector ITL = TF.InverseTransformPosition(
 UGameplayStatics::GetPlayerCameraManager(GetWorld(), 0)->
 GetTransformComponent()->GetSocketLocation(""));
FVector FL = PairedDoor->GetTransform().TransformPosition(ITL);
PairedDoor->CaptureCamera->SetWorldLocation(FL);
```

Next, the camera's rotation is calculated in a similar method. First, the player's camera rotation is broken into it's axes then a FRotator is made from each axis after mirroring it in local space, which returns the final rotation.

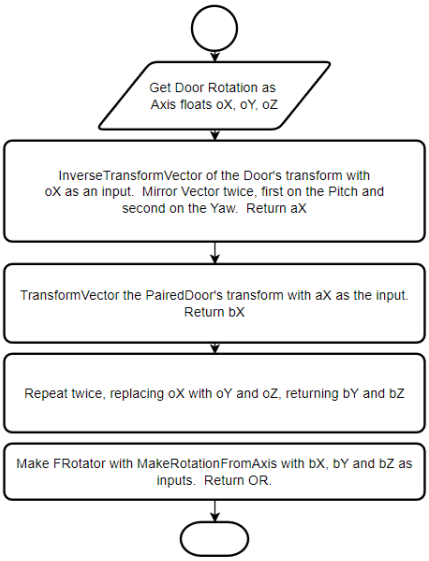

Figure 17. Updating the capture camera's rotation. The rotation of the camera is updated based on the rotation of the player to the paired portal

```
// First, setup some holding values for each axis
FVector OutX, OutY, OutZ;

// Then break the rotation of the input into axes,
// returning a vector for pitch, roll and yaw based on
 //the rotation's position on each axis
UKismetMathLibrary::BreakRotIntoAxes(InRotator, OutX, OutY,
↪  OutZ);

// Then make a rotator from the axes vectors after
 //mirroring each axis on the inverse tranform of the paired
 ↪  door
// This will flip the camera from being positive or
 //negative (aka it will point in the opposite direction
 //of the player on the paired door)
FRotator OR = UKismetMathLibrary::MakeRotationFromAxes(
 PairedDoor->GetTransform().TransformVector(GetTransform().
  InverseTransformVector(OutX).MirrorByVector
  (FVector(1.0f, 0.0f, 0.0f)).MirrorByVector
  (FVector(0.0f, 1.0f, 0.0f))),
 PairedDoor->GetTransform().TransformVector(GetTransform().
  InverseTransformVector(OutY).MirrorByVector
  (FVector(1.0f, 0.0f, 0.0f)).MirrorByVector
  (FVector(0.0f, 1.0f, 0.0f))),
 PairedDoor->GetTransform().TransformVector(GetTransform().
  InverseTransformVector(OutZ).MirrorByVector
  (FVector(1.0f, 0.0f, 0.0f)).MirrorByVector
  (FVector(0.0f, 1.0f, 0.0f)))
);
```

Finally, the door checks if there is an overlap currently being made. This is done by first checking if the TriggerZone is currently overlapping something. If

so, then GetShouldTeleport is called which completes a set of tests to make sure that it is a character that overlapping. A final test is conducted to check if the character is overlapping the portal mesh via the function GetIsOverlappingPortal.

```
// Check if the object colliding with the trigger zone is a
 //Character
if (!TriggerZoneOverlaps.IsEmpty()) {
 if (TriggerZoneOverlaps[0]) {
  if (TriggerZoneOverlaps[0]->IsA
   (ACharacter_Parent::StaticClass())) {
```

GetIsOverlappingPortal starts by checking if the distance of the portal's normal is greater than or equal too the distance between the point to check and the portal's location. Next, a plane is made from the portal location and the normal (the direction of the portal), which is used calculate if the point is intersecting the portal. Finally, it calculates if the character is crossing the portal, while also updating bLastInFront and store the PointToCheck for the next check.

```
bool ALevelDoor::GetIsCrossingPortal(FVector PointToCheck,
 FVector PortalLocation, FVector PortalNormal){
// Check if the distance of the portal's normal is greater than
 //or equal too the distance between the point to check and
//the portal's location
bool bIsInFront = FVector::DotProduct(PortalNormal,
↪  PointToCheck - PortalLocation) >= 0.0f;

// Next, make a plane from the portal location and the normal
↪  (the direction of the portal)
// And use the plane to calculate if the point is intersecting
↪  the portal
FPlane PortalPlane =
↪  UKismetMathLibrary::MakePlaneFromPointAndNormal
 (PortalLocation, PortalNormal);
float T; FVector IntersectionLocation;
bool bIsIntersecting = UKismetMathLibrary::
 LinePlaneIntersection(LastPosition, PointToCheck,
 PortalPlane, T, IntersectionLocation);

// Calculate if we are crossing the portal, update bLastInFront
 //and store the PointToCheck for the next check
bool bIsCrossingPlane = bIsIntersecting && !bIsInFront &&
↪  bLastInFront;
bLastInFront = bIsInFront; LastPosition = PointToCheck;

return bIsCrossingPlane;}
```

If GetIsOverlappingPortal returns true, the door finally teleports the character to it's pair, keeping the current velocity of the character while teleporting. It is implemented in a similar way to how the capture camera is moved and rotated and the velocity is kept the same by using the CharacterMovementComponent's Velocity property as the input for an InverseTransformVector. This is then mirrored on the Pitch and Yaw as before, then used as the input for a TransformVector of the PairedDoor's transform. This value is then used to calculate the characters final velocity by multiplying it and the character's Velocity length.

```
void ALevelDoor::TeleportCharacter(AActor* CharacterToTeleport)
{
ACharacter_Parent* Char =
↪  Cast<ACharacter_Parent>(CharacterToTeleport);
FTransform TF = GetActorTransform();
TF.SetScale3D(FVector(TF.GetScale3D().X * -1, TF.GetScale3D().Y
↪  * -1, TF.GetScale3D().Z));
FVector ITL =
↪  TF.InverseTransformPosition(CharacterToTeleport->GetActorLocation());
FVector FL = PairedDoor->GetTransform().TransformPosition(ITL);

CharacterToTeleport->SetActorLocation(FL);
CharacterToTeleport->SetActorRotation(GetInverseRotation
 (CharacterToTeleport->GetActorRotation()));
Char->GetController()->SetControlRotation(GetInverseRotation
 (Char->GetController()->GetControlRotation()));

// Now update Char's velocity
FVector Vel = Char->GetMovementComponent()->Velocity;
↪  Vel.Normalize(0.001);
FVector CIT =
↪  Char->GetActorTransform().InverseTransformVector(Vel);
FVector OutVel =
↪  PairedDoor->GetActorTransform().TransformVector
 (CIT.MirrorByVector(FVector(1.0f, 0.0f, 0.0f)).MirrorByVector
  (FVector(0.0f, 1.0f, 0.0f)));
Char->GetMovementComponent()->Velocity = OutVel *
 Char->GetMovementComponent()->Velocity.Length();
}
```

8b - Level Spawning Part One

Commit

As the way to enter and exit levels was now complete, I turned my focus to the level spawning and setup. I started by creating a ParentRoom class, which all rooms that I create will extend from. It consist of three components - a Root SceneComponent and two ChildActorComponents for the entrance and exit portal doors. Additionally, the class contains 2 functions: SetupRoomDoors, which sets up the scene capture between the doors and SetupMechanics, which sorts each child actor component into separate arrays.

```
void ARoom_Parent::SetupRoomDoors(ALevelDoor* ShipDoor,
↪  ALevelDoor* HiddenShipDoor)
{
// Check that the door classes have been set up correctly
if (EntranceDoorComponent->GetChildActor()) {
 // If so, store their pointers
 EntranceDoor =
 ↪  Cast<ALevelDoor>(EntranceDoorComponent->GetChildActor());
 ExitDoor =
 ↪  Cast<ALevelDoor>(ExitDoorComponent->GetChildActor());
```

```
 // Then setup the doors
 ShipDoor->SetupPairedDoor(EntranceDoor);
 EntranceDoor->SetupPairedDoor(ShipDoor);
 ExitDoor->SetupPairedDoor(HiddenShipDoor);
 HiddenShipDoor->SetupPairedDoor(ExitDoor, ShipDoor);
}}

void ARoom_Parent::SetupMechanics()
{
// Get all ChildActorComponents
TArray<AActor*> ChildActors;
GetAllChildActors(ChildActors, false);

// Then sort them into arrays based on what class they are
for (AActor* i : ChildActors) {
 if (i->IsA(ARecordPad::StaticClass())) {
  RecordPads.Add(Cast<ARecordPad>(i));
  RecordPads[RecordPads.Num() -
  ↪  1]->SetupVisualElements(NextRecordPadIndex);
  NextRecordPadIndex++;
 }
 else if (i->IsA(AMechanicObject_Parent::StaticClass())) {
  MechanicObject.Add(Cast<AMechanicObject_Parent>(i));
}}}
```

Next, I created the struct FLevelData which is used to store all the level data used to spawn levels in a data table. Currently, it only contains Name and Class property, but will be updated later with 'sectors' - or collections of levels.

```
struct STARMENDERS_API FLevelData : public FTableRowBase
{
public:
sGENERATED_BODY();

/// -- Level Data --
// The name of the level
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Level
↪  Data")
float Name;

// The class of the leve;
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Level
↪  Data")
TSubclassOf<class ARoom_Parent> Class;
}
```

Next, I fixed a small bug I created when the player teleports through a portal door. Before they would be jolted slightly to the right, as the calculations would not create the correct velocity vector and would instead return one rotated to the right. I fixed this by replacing InverseTransformVector with InverseTransformDirection from UKismetMathLibrary.

```
FVector Vel = Char->GetMovementComponent()->Velocity;
↪  Vel.Normalize(0.0001);
FVector CIT = UKismetMathLibrary::InverseTransformDirection
 (GetActorTransform(), Vel);
FVector OutVel = UKismetMathLibrary::
 TransformDirection(PairedDoor->GetActorTransform(),
 CIT.MirrorByVector(FVector(1.0f, 0.0f, 0.0f)).
 MirrorByVector(FVector(0.0f, 1.0f, 0.0f)));
```

I also implemented SetDoorState, which allows the door to be toggled open or closed. When closed the paired door's capture component is not updated and the player cannot teleport through it.

Next I worked on updating the WorldObjects and MechanicObjects. First, I created a new mechanic object of WorldObjectStart, which simply spawns their associated WorldObject when the recording is started. I also updated the WorldObject to use the ObjectMesh as it's root insteads of the current SceneComponent, which I plan on removing. Finally, I added a ResetToDefault virtual function in the parent MechanicObject class, which is used to reset the mechanic object to it's default state/location.

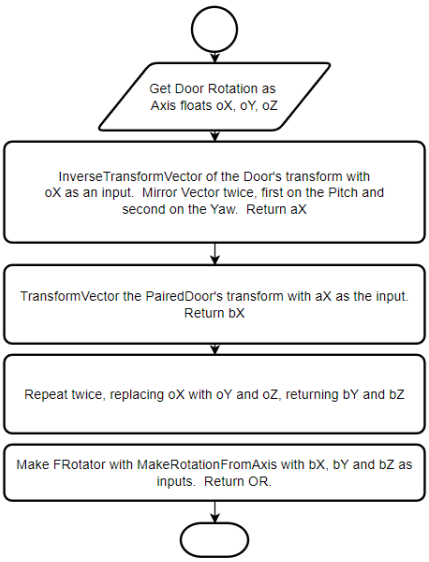

Figure 18. The WorldObjectStart object. When recording is started, the WorldObjectStart either moves it's spawned object back to it's location or spawns a new one

Next was the LevelController. First, I updated the constructor - which now finds the data table of all the levels and stores it and also disables the Tick function of the class. Following this, I implemented the SetupLevel function, which finds the data table row of a

supplied LevelID and spawns it if it is valid. Next, the level is setup by calling SetupRoomDoors and SetupMechanics in the new level class.

```
void ALevelController::SetupLevel(FName InLevelID)
{
// Find the level from the data table
FLevelData FoundRoom;
FoundRoom = *LevelDataTable->FindRow<FLevelData>(InLevelID, "",
↪  true);

if (FoundRoom.Class) {
 // Get the class from the found struct and spawn it in the
 ↪  world
 ActiveRoom =
 ↪  GetWorld()->SpawnActor<ARoom_Parent>(FoundRoom.Class,
  FVector(0.0f, 1000.0f, 2000.0f), FRotator(0.0f, 40.0f,
  ↪  0.0f));
 ActiveRoom->SetupRoomDoors(ShipDoor, HiddenShipDoor);
 ActiveRoom->SetupMechanics();
}}
```

I also updated StartLevelPlayback and EndLevelPlayback, which now use the RecordPad array of the spawned level and also open/close the entrance door.

This was my first attempt at the level spawning system, which turned out to work how I expected it too until I found out you cannot set child actor pointers in the class editor. My next attempt will be creating a tool which translates all of the child actor components into structs, then use the data in those structs to individually spawn each mechanic object and record pads in the world, rather than use one class.

8c - Level Spawning Part Two

Commit

After doing some external research, I decided to have a new attempt at level generation by instead not using a class to spawn a level, instead spawning each part of the level individually. I started by reworking the FLevelData struct to hold the new level information. First, I added two new structs - these being FRecordPadData, which stores the Transform and Index of RecordPads in a level, and FMechanicData, which stores the ID, Transform, Class, OutputIDs, TriggerRequirement and DefaultObjectState of each mechanic in a level. I also added LevelMesh, EntranceDoorTransform, ExitDoorTransform, RecordPad TArray and Mechanics TArray properties to LevelData.

```
USTRUCT(BlueprintType, Category = "Levels")
struct STARMENDERS_API FRecordPadData
// The index of this record pad
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Record
↪  Pad Data")
int Index = -1;

// The transform of the record pad
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Record
↪  Pad Data")
FTransform Transform;


USTRUCT(BlueprintType, Category = "Levels")
struct STARMENDERS_API FMechanicData
// The ID of this mechanic
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
↪  "Mechanic Data")
FName ID;

// The transform of the mechanic
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
↪  "Mechanic Data")
FTransform Transform;

// The class of the mechanic object
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
↪  "Mechanic Data")
TSubclassOf<class AMechanicObject_Parent> Class;

// The ID's of this mechanic's output object (if there is any)
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
↪  "Mechanic Data")
TArray<FName> OutputIDs;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
↪  "Mechanic Data")
int TriggerReqirements = 0;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
↪  "Mechanic Data")
TEnumAsByte<enum EObjectState> DefaultObjectState;

USTRUCT(BlueprintType, Category = "Levels")
struct STARMENDERS_API FLevelData : public FTableRowBase
/// -- Level Data --
// The name of the level
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Level
↪  Data")
float Name;
FName Name;

// The static mesh of the level
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Level
↪  Data")
TSubclassOf<class ARoom_Parent> Class;
UStaticMesh* LevelMesh = nullptr;

/// -- Portal Door Data --
// The transform of the entrance door
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Portal
↪  Door Data")
FTransform EntranceDoorTransform;

// The transform of the exit door
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Portal
↪  Door Data")
FTransform ExitDoorTransform;
```

```
// TArray of all record pads in the level
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Record
↪  Pad Data")
TArray<FRecordPadData> RecordPads;

// TArray of all record pads in the level
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
↪  "Mechanic Data")
TArray<FMechanicData> Mechanics;
```

Next, I updated the LevelDoor to use a enum property to denote each door - between Entrance, Exit and Ship. These are used to correctly pair the doors together and make sure only the Entrance and Exit doors are moved by the level controller. Additionally, I changed the property type of ObjectName in the MechanicObject Parent from a FString to a FName, to remove any unneeded conversions between the two types.

Now was updating the LevelController to use the new spawning method. I started off by adding new properties - these being: two pointers to the EntranceDoor and ExitDoor, a FVector used to offset the level, a TArray of MechanicObject Parent to all mechanics spawned for the level and a RecordClass pointer, used to set the class to spawn record pads - while removing the unneeded Room Parent pointer. Additionally, the door setup is moved from the RoomParent class to the LevelController.

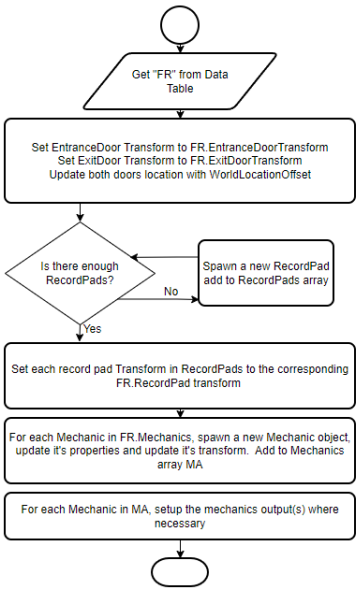

Figure 19. The theory behind level setup.

Next, I added a StaticMeshComponent that displays the mesh of the level - which is a mesh made from merging multiple other meshes together in Unreal. Following this, I updated the SetupLevel function. As RoomParent classes are no longer used, SetupLevel starts by moving the Entrance and Exit portal door transforms, followed by the spawning or moving of all required RecordPads.

```
if (FoundRoom.Name != "") {
// First, update the entrance and exit portal locations
EntranceDoor->SetActorTransform(FoundRoom.EntranceDoorTransform);
EntranceDoor->AddActorWorldOffset(WorldLocationOffset);
ExitDoor->SetActorTransform(FoundRoom.ExitDoorTransform);
ExitDoor->AddActorWorldOffset(WorldLocationOffset);

// Next, spawn the required RecordPads
for (FRecordPadData i : FoundRoom.RecordPads) {
 ARecordPad* NewPad =
 ↪  GetWorld()->SpawnActor<ARecordPad>(RecordPadClass,
 ↪  i.Transform);
 NewPad->AddActorWorldOffset(WorldLocationOffset);
 RecordPads.Add(NewPad);
 NewPad->SetupVisualElements(i.Index);
}
```

Next, all MechanicObjects required by the new level are spawned and their outputs are setup where necessary. Finally, the LevelMesh is set.

```
// Follow with the Mechanic Objects
for (FMechanicData j : FoundRoom.Mechanics) {
 AMechanicObject_Parent* NewMechanic = GetWorld()->SpawnActor
  <AMechanicObject_Parent>(j.Class, j.Transform);
 NewMechanic->AddActorWorldOffset(WorldLocationOffset);
 NewMechanic->ObjectName = j.ID;
 NewMechanic->ObjectState = j.DefaultObjectState;
 NewMechanic->TriggerRequirement = j.TriggerReqirements;
 Mechanics.Add(NewMechanic);
}

// And setup their outputs where nessassary
for (int k = 0; k < Mechanics.Num(); k++) {
 for (FName l : FoundRoom.Mechanics[k].OutputIDs) {
  for (AMechanicObject_Parent* m : Mechanics) {
   if (m->ObjectName == l) {
        Mechanics[k]->OutputsObjects.Add(m);
}}}}

// Set the level mesh
LevelMesh->SetStaticMesh(FoundRoom.LevelMesh);
```

This second attempt worked much better than the first attempt, but takes a slightly different path of storing the level data. Instead of using a single class per level, a row in the data table now stores all of the mechanic details, the record pad locations and the portal

door locations in a struct. However, levels became incredibly difficult to setup, so I created a editor tool to convert level data into a struct. This tool is made out of two classes - a EditorUtilityidget which handles the user input and a Actor object which converts the level data into a struct for the data table.

Starting with the EditorUtilityWidget, I added a Editor Utility Editable Text Box to take in the level name, as well as a Editor Utility Button to accept the name and start the level creation. Additionally, I added an output message Text Block to display either a success or fail message to the user.

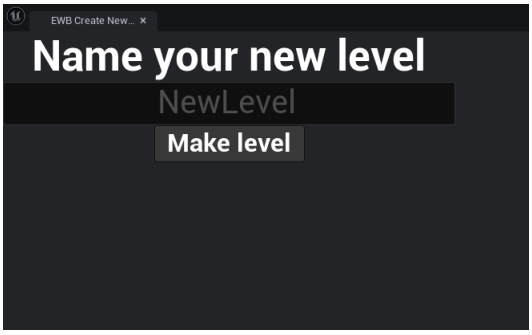

Figure 20. The EWB CreateNewLevel widget. When MakeLevel is pressed, the level creation is started

Next, I implemented the button's event. First, the widget checks if a GetLevelData object already exists in the world, and destroys it if it does exist. Next, a new GetLevelData object is spawned and the data in the Editable Text Box is checked. If the text box is empty, then a default value is used, else the inputted text is stored for later.

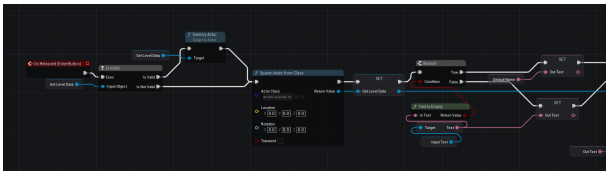

Figure 21. Part One of the button event.

Next the static meshes in the level are collected and merged into a new static mesh with the name inputted. These are outputted in Levels/OutLevelMeshes.

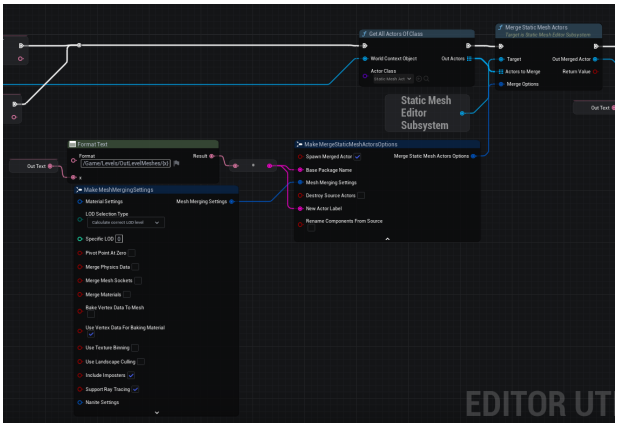

Figure 22. Part Two of the button event.

Following this, StartGeneratingLevelData is called on the GetLevelData actor, using the new merged mesh and the level name as inputs. StartGeneratingLevelData begins by finding all of the level doors that currently exist and storing the Transform of the EntranceDoor and ExitDoor using each of the door's enums.

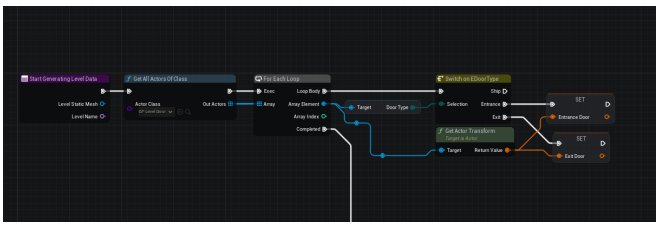

Figure 23. Part One of StartGeneratingLevelData.

Next, the same is done with all existing RecordPads - finding all of the objects and storing the transform - but also their index is collected.

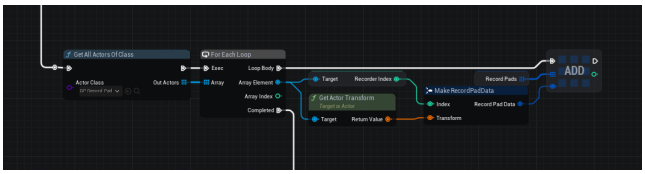

Figure 24. Part Two of StartGeneratingLevelData.

Repeat with all MechanicObjects and store all

necassary information required by MechanicData - ObjectName, transform, class, any LinkedObjects, TriggerRequirements and the DefaultObjectState.

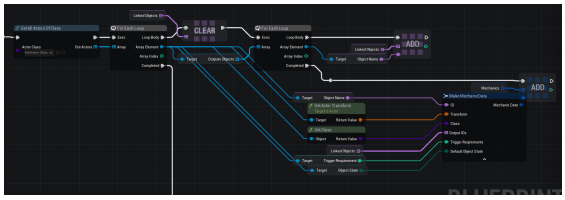

Figure 25. Part Three of StartGeneratingLevelData.

Finally, all of the information collected is added to a FLevelData struct and outputted back to the widget.

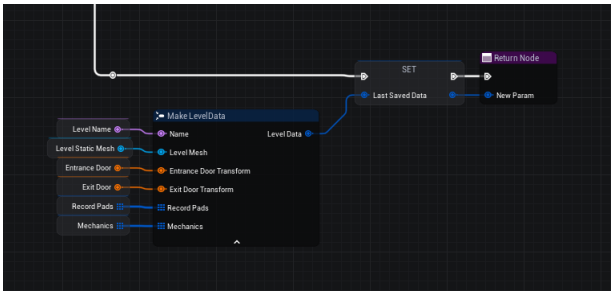

Figure 26. Part Four of StartGeneratingLevelData.

The widget uses the FLevelData to add a new data table row in the DT LevelData table and, after a short delay, destroys the GetLevelData actor from the world. However, the data table will require saving to keep the new row(s) as BluePrints don't have any functions to save assets.

Figure 27. Part Three of of the button event.

This was the second time I created a Editor Utility Blueprint and I feel like this works well to what it needs. It successfully transfers the data from the level I create into a single data struct which can then be used to re-spawn the level in game. It also allows me to bypass the issue that came up in attempt one - where pointers could not be set between child actors in a class.

Finally, I made a new Unreal Map LevelCreationMap, used to create the levels and convert them into the data structs. I also created some test levels, which were successfully created.

8d - Mechanic Simplification

Commit

I decided that the mechanic objects were currently to complex, with some scripting required by only inputs and not outputs and vice versa. Additionally, implementing new mechanics could be slightly confusing as both classes used the same superclass. To fix this, I created two new subclasses of MechanicObject Parent - Input and Output.

Input Objects now have the ToggleInputActive function, which simply inverts the current state of bInputActive, and calls IncreaseInputAmount on all output objects. OuptutObjects now have the IncreaseInputAmount function - which increases its InputCount, and calls ToggleOutput when it reaches its InputRequirements - and the ToggleOutput function - a virtual function used to update the object when called.

As the mechanics were now split, I also had to update the LevelData struct and LevelController to use the two new superclasses. LevelController now first spawns the OutputObjects, then the InputObjects while also setting any OutputObject pointers during Input spawning.

```
// Follow with the Output Mechanic Objects
for (FOutputMechanicData j : FoundRoom.OutputMechanics) {
AMechanicObject_Output* NewMechanic = GetWorld()->
 SpawnActor<AMechanicObject_Output>(j.Class, j.Transform);
NewMechanic->AddActorWorldOffset(WorldLocationOffset);
NewMechanic->ObjectName = j.ID;
NewMechanic->InputRequirement = j.InputRequirements;
NewMechanic->bOutputAlwaysActive = j.bOutputAlwaysActive;
OutputMechanics.Add(NewMechanic);
}

// And the Input Mechanic Objects
for (FInputMechanicData j : FoundRoom.InputMechanics) {
 AMechanicObject_Input* NewMechanic = GetWorld()->SpawnActor
  <AMechanicObject_Input>(j.Class, j.Transform);
 NewMechanic->AddActorWorldOffset(WorldLocationOffset);
 NewMechanic->ObjectName = j.ID;
  for (FName l : j.OutputIDs) {
   for (AMechanicObject_Output* m : OutputMechanics) {
        if (m->ObjectName == l) {
         NewMechanic->OutputsToObjects.Add(m);
```

```
}}
 InputMechanics.Add(NewMechanic);
}
```

8e - Opening the Exit Door

Commit

Currently, each level has no goal for the player to reach. To fix this, I updated the Portal Door to now extend from MechanicObject Output. This allows input objects to interact with the door and allow it to be unlocked while the level is being played. However, this required the removal of the Portal Door's Root component as it now inherits one from MechanicObject Output.

I also did some cleanup of other classes to utilize the new Portal Door. LevelData now stores the ExitDoorRequirements, MechanicData Output now has a SetAlwaysActive function which is called by the LevelController to pre toggle the output if necessary and updated the LevelCreationTool to use the new updated LevelData struct, while also adding a check to not add LevelDoors to the OutputMechanics array by comparing their class.

8f - Removing Recording Character

Commit

After completing some play testing in the previous update, I found that when a player finished a level they would need to re-possess the Default character, which then needed to teleport to the same location as the current Record character. This would require the LevelDoor class to find the Default character in the world, teleport it to an exact location and then call RePossess to keep the illusion of the portal. To work around this, I decided to remove the Recording character and only use Default and Playback characters.

I started by dividing each function and property that currently was in each class into three categories - Default, Playback and Both - based on which class uses which. I also worked out what functions needed to be overridden.

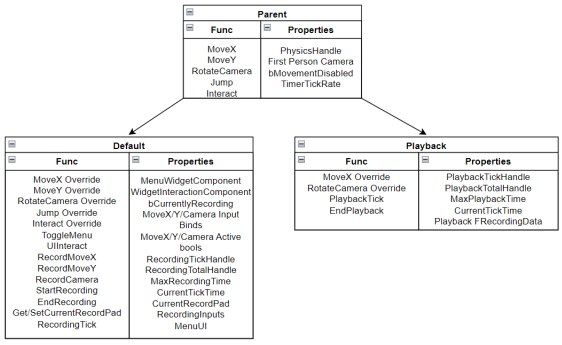

Figure 28. The refactor of the Character classes.

I stared by moving everything currently in the classes to a separate holding text document to consolidate everything. I then began moving everything required by both classes back into the Character Parent class, then the Playback class and finally the Default class. Items added to the Default class from the other classes include the functions: ToggleMenu, UIInteract, RecordMoveX, RecordMoveY, RecordCamera, StartRecording, EndRecording, RecordingTick and the properties: Input Value Binds, Input Active Bools, Recording Timer Handles, Max and Current Tick times, TArrays of Recorded Inputs and MenuUI pointer.

Next, I updated the classes that previously interacted with the Recording character to now interact with the Default class, starting with the RecordPad. Instead of spawning a Character Record, the RecordPad now calls StartRecording on the Default character when the player starts recording. Additionally, SetRecording no longer calls the controller to RePossess and instead moves the character to the pad's location.

```
void ARecordPad::StartRecording(ACharacter_Default*
↪  PlayerCharacter)
{
ClearRecording();

if (PlayerCharacter) {
 // Setup the character and start it's timer
 PlayerCharacter->StartRecording(this);
}}

void ARecordPad::SetRecording(FRecordingData NewRecord,
 ACharacter_Default* PlayerCharacter)
{
PlayerCharacter->SetActorLocation(GetActorLocation() +
 FVector(0.0f, 0.0f, 200.0f));
}
```

Next, I updated the RecordingUI to now cast to Character Default instead. Finally, I updated the Controller Player. First, the pointer type of Character was changed from Character Parent to Character Default as the player will now only ever control a Default character. Additionally, I updated OnPossess to no longer check if the class is of Character Record or Character Default for the same reason.

```
Controller_Player.h...
class ACharacter_Default* Character = nullptr;

AController_Player::OnPossess(APawn* InPawn)
{
Super::OnPossess(InPawn);

if (!Character) {
 // Cast to the character pawn and store it
 Character = Cast<ACharacter_Default>(GetPawn());
}

bInMenu = false;
Character->ToggleMenu(bInMenu);
bShowMouseCursor = bInMenu;

Character->SetActorHiddenInGame(false);
Character->SetActorEnableCollision(true);

// Get the EnhancedInputComponent
UEnhancedInputComponent* EnhancedInputComponent =
↪  Cast<UEnhancedInputComponent>(InputComponent);
if (EnhancedInputComponent) {
 Character->MoveXBind =
 ↪  &EnhancedInputComponent->BindActionValue(InputConfig->MoveXInput);
 Character->MoveYBind =
 ↪  &EnhancedInputComponent->BindActionValue(InputConfig->MoveYInput);
 Character->CameraBind =
 ↪  &EnhancedInputComponent->BindActionValue(InputConfig->CameraInput);
}}
```

8g - Standing Button and WorldObjectSpawner

Commit

My next goal was to add a button that the player can interact with to trigger other mechanics, like Portal 2's Pedistal Button (Valve, 2011). I started by creating a new MechanicObject subclass, added it's static mesh components and set a gameplay tag to mark it as interactable.

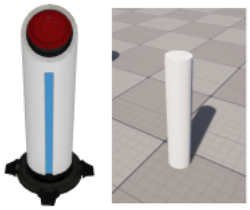

Figure 29. Portal 2's Pedistal button and my StandingButton.

Next, I implemented the UseButton function which is called by the player's Interact function. When used, the button is enabled and is disabled after a short delay.

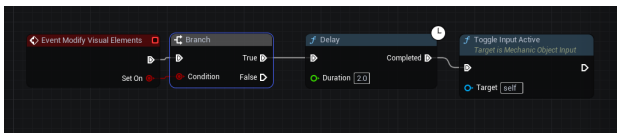

Figure 30. The UseButton function in StandingButton.

I then updated the Character's Interact function. When pressed, Interact will now check if the IsInteractable tag exists on the class. If it does, it currently casts to the StandingButton and calls UseButton. The use of a gameplay tag allows more inputs to be interacted with later on in development (if required).

```
void ACharacter_Parent::Interact()...
 // Finally, update the WorldObject's CurrentGrabber with this
 ↪  character
 CG->SetCurrentGrabber(this);
}
// If the object can't be grabbed, check if it has an
↪  interactable tag
// Currently only SButtons can be interacted with, but this
↪  could change in future versions
 else if
 ↪  (TraceHit.GetActor()->ActorHasTag(FName("IsInteractable")))
 ↪  {
 // Cast to the SButton class
 AMechanicObject_SButton* SButton =
 ↪  Cast<AMechanicObject_SButton>(TraceHit.GetActor());
 SButton->UseButton();
}}
```

Next I added a new output mechanic - the WorldObjectSpawner. As the name implies, when triggered it spawns a new WorldObject at it's location. This is also inspired by a Portal 2 object, the Vital Apparatus Vent. It is made up of three components - two static mesh components and a SceneComponent to control the object's spawn location. I also added a pointer to the spawned object and a class pointer to control what world object is spawned. On ToggleOutput, the input argument is checked to be true as we only want the box to move/spawn on activation. Next, the pointer is checked if it is set (an object is spawned). If one has, it

is moved to the ObjectSpawnLocation. If one hasn't, it is spawned now.

```
void AMechanicObject_WOSpawner::ToggleOutput(bool bNowActive)
{
// Check if bNowActive is true (we only want the box to move on
↪  activation)
if (bNowActive = true) {
 // Check if LiveObject is set
 if (LiveObject) {
  // Move it to the ObjectSpawnLocation
  LiveObject->SetActorLocation(ObjectSpawnLocation->
  GetComponentLocation());
 }
 // If it isn't, spawn an object now
 else {
  LiveObject = GetWorld()->SpawnActor<AActor>(ObjectToSpawn,
   ObjectSpawnLocation->GetComponentLocation(),
   ObjectSpawnLocation->GetComponentRotation());
 }}
ModifyVisualElements(bNowActive);
}
```

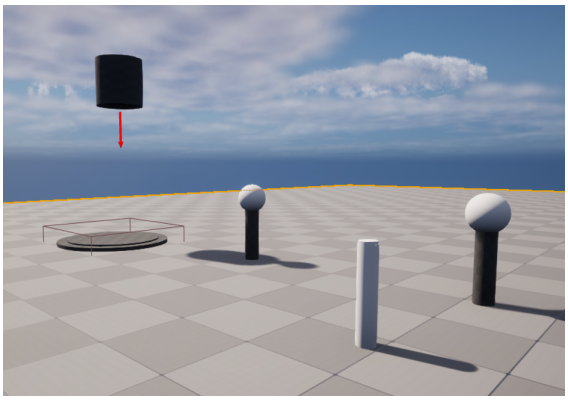

Figure 31. StandingButton and WorldObjectSpawner in the world.

8h - End Record on Room Complete

Commit

My next aim was to update the record pad to stop recording when the player completes and exits the level. First, I updated the RecordPad by adding a default property for the PlayerCharacter argument in SetRecording. Additonally, the function now checks if the PlayerCharacter parameter is valid. If it is, the player is teleported back to the record pad's location. This allows the record pad to stop recording while not teleporting the player to it.

```
void SetRecording(FRecordingData NewRecord,
 ACharacter_Default* PlayerCharacter = nullptr)...

if (PlayerCharacter) {
 PlayerCharacter->SetActorLocation(GetActorLocation()
  + FVector(0.0f, 0.0f, 200.0f));
}
```

Next, I updated the LevelDoor. It now checks if the character is a default character before teleporting. If it is, then it casts to the Character Default class and checks if they are recording, ending the recording if they are.

```
void ALevelDoor::TeleportCharacter(AActor*
↪  CharacterToTeleport)...
// Finally, if the door is the ExitDoor and the
//character is recording, save the recording on the record pad
if (Char->IsA(ACharacter_Default::StaticClass()) && DoorType ==
↪  Exit) {
 ACharacter_Default* Def = Cast<ACharacter_Default>(Char);
  if (Def->bCurrentlyRecording) {
   Def->EndRecording(false);
}}
```

After this, I updated the Default Character. Now, the menu is no longer toggled when the recording is ended, except if they are teleported back to the record pad in use.

```
if (bTeleportCharacter) {
 CurrentRecordPad->SetRecording(FRecordingData(MoveXRecording,
  MoveYRecording, JumpRecording, InteractRecording,
  ↪  CameraRecording), this);
 Cast<AController_Player>(GetController())->ToggleMenu(false);
}
else {
 CurrentRecordPad->SetRecording(FRecordingData(MoveXRecording,
  MoveYRecording, JumpRecording, InteractRecording,
  ↪  CameraRecording));
}
```

Finally, I fixed a bug that I erroneously introduced - where I forgot a second = in a comparison in the WorldObjectSpawner.

8i - Level Select

Commit

My final aim for this version was to get a level select in the game.

My first iteration of the level select was to use a panel with a large UI element displaying each level in the sector currently selected. However, this design didn't match the diegetic theme that I was going for

to the full extent. The character would be able to see the panel in the world, but my aim was to have a 3D element showing the sector as well. Additionally, having sectors of varying size could prove to be difficult to implement onto the UI canvas.

My next iteration was to have a 3D level select with levels rotation around a central star. Each level is displayed as a planet, with each planet having a different colour scheme, distance to star and rotation speed. I started by creating a data struct to hold this information - named SelectorData.

Each sector is stored in a single FSectorData, which is stored in a data table. Each sector currently has an ID and an array of FPlanetData, which stores all of the information about the planet rotation around the central star, as well as the level ID spawned by the LevelController when selected.

```
USTRUCT(BlueprintType, Category = "Levels")
struct STARMENDERS_API FPlanetData
{
public:
GENERATED_BODY();

// The ID of the planet data of this level
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Level
↪  Data")
FName PlanetID;

// The static mesh of the planet
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Planet
↪  Data")
UStaticMesh* Mesh = nullptr;

// The rotation rate of the planet on it's axis
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Planet
↪  Data")
float AxisRotationRate;

// The rotation rate of the planet around the central star
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Planet
↪  Data")
float CentralRotationRate;

// The distance of the planet from the central star (the spring
↪  arm's length)
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Planet
↪  Data")
float Distance;

// The relative starting rotation based on the current time
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Planet
↪  Data")
float RelativeStartingRotation;

// The scale of the planet
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Planet
↪  Data")
float Scale;

// The colour of the planet
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Planet
↪  Data")
FColor Colour;

public:
FPlanetData();
~FPlanetData();
};

USTRUCT(BlueprintType, Category = "Levels")
struct STARMENDERS_API FSectorData : public FTableRowBase
{
public:
GENERATED_BODY();

// The name of the sector
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Sector
↪  Data")
FName SectorID;

// The planets which are contained in this sector
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Sector
↪  Data")
TArray<FPlanetData> PlanetIDs;

public:
FSectorData();
~FSectorData();
};
```

Next, I created the LevelSelect object itself. It simply consists of a Root SceneComponent, a Camera used to blend the player's view target between and a StaticMeshComponent for the central star. I started by implementing the constructor - which finds the sector data table - and BeginPlay - which finds and stores a pointer in the LevelController.

I followed this by implementing the ability to spawn a new planet mesh. However, as I wanted the levels to rotate around the central star, I had to spawn in more than just a static mesh component. I did this by first implementing the FPlanetVisualData struct, which holds pointers to all of the components used for each planet.

```
USTRUCT(BlueprintType, Category = "Levels")
struct STARMENDERS_API FPlanetVisualData
{
public:
GENERATED_BODY();

// The planet mesh component
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Sector
↪  Data")
UStaticMeshComponent* Mesh;

// The planet spring arm component
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Sector
↪  Data")
USpringArmComponent* SpringArm;

// The planet rotation component
```

```
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Sector
↪  Data")
URotatingMovementComponent* Rotating;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Sector
↪  Data")
FName LevelID;

public:
FPlanetVisualData();
FPlanetVisualData(UStaticMeshComponent* NewMesh,
↪  USpringArmComponent*
 NewSpringArm, URotatingMovementComponent* NewRotating,
  FName NewLevelID = "");
void UpdateLevelID(FName NewLevelID);
        ~FPlanetVisualData();
};
```

With this new struct, I added the AddNewPlanetVisual function, used to spawn in a new set of components. It begins by adding a new spring arm, then a a new static mesh component, attaching the mesh to the spring arm, adding a new rotating movement component and setting it up to target the spring arm. A new struct is created containing these components, which is then added to an array for later.

```
// Add a new spring arm
USpringArmComponent* NewSpringArm = NewObject
 <USpringArmComponent>(this, FName(*FString::Printf
  (TEXT("Planet Spring Arm %i"), NewNumber)));
NewSpringArm->RegisterComponent();
NewSpringArm->bDoCollisionTest = false;
NewSpringArm->AttachToComponent(Root,
↪  FAttachmentTransformRules::
 SnapToTargetNotIncludingScale, "");

// Then a new static mesh component
UStaticMeshComponent* NewMeshComp = NewObject
 <UStaticMeshComponent>(this, FName(*FString::Printf
 (TEXT("Planet Mesh %i"), NewNumber)));
NewMeshComp->RegisterComponent();
NewMeshComp->AttachToComponent(NewSpringArm,
↪  FAttachmentTransformRules::
 SnapToTargetNotIncludingScale, "SpringEndpoint");

// Finally, add a new rotational movement component
URotatingMovementComponent* NewRotComp = NewObject
 <URotatingMovementComponent>(this, FName(*FString::Printf
 (TEXT("Planet Rot Comp %i"), NewNumber)));
NewRotComp->RegisterComponent();
NewRotComp->SetUpdatedComponent(NewSpringArm);

// Add it to the array
PlanetVisuals.Add(FPlanetVisualData(NewMeshComp, NewSpringArm,
↪  NewRotComp));
```

To simplify spawning in multiple new planet visuals, I made the function recursive by adding a Target input and comparing if the array's length is equal to the target.

```
// Check if there is now enough component sets added.  If not,
↪  recurse
if (PlanetVisuals.Num() != Target) {
 AddNewPlanetVisual(Target);
```

Next, I added the SetActiveSector function, used to update the LevelSelector with a new sector of planets. The function starts by searching for the sector inputted in the data table, then calling AddNewPlanetVisual to add enough component sets until the required amount is needed.

```
// Find the sector in the data table
FSectorData FoundSector;
FoundSector =
↪  *WorldMapDataTable->FindRow<FSectorData>(SectorID, "",
↪  true);

// Add enough to match the amount of planets needed
if (FoundSector.PlanetIDs.Num() != PlanetVisuals.Num()) {
 AddNewPlanetVisual(FoundSector.PlanetIDs.Num());
}
```

Following this, each planet visual is updated to match the data in the table.

```
// Next, setup each planet visual
for (int i = 0; i < PlanetVisuals.Num(); i++) {
 // First, set the matching planet name
 PlanetVisuals[i].LevelID = FoundSector.PlanetIDs[i].PlanetID;

 // Next, setup the spring arm
 PlanetVisuals[i].SpringArm->TargetArmLength =
 ↪  FoundSector.PlanetIDs[i].Distance;

 // Follow with the static mesh
 PlanetVisuals[i].Mesh->SetStaticMesh(FoundSector.
  PlanetIDs[i].Mesh);
 PlanetVisuals[i].Mesh->SetWorldScale3D(FVector
  (FoundSector.PlanetIDs[i].Scale));

 // Finally, setup the rotating movement component
 PlanetVisuals[i].Rotating->RotationRate = FRotator
  (0.0f, FoundSector.PlanetIDs[i].CentralRotationRate, 0.0f);
}
```

Next, I added the SetActive and EndInteraction functions. These simply update the CharacterUsing function in the LevelSelector, allowing the object to call ExitInteraction on the player.

```
void ALevelSelector::SetActive(ACharacter_Default*
↪  NewInteractee)
{
CharacterUsing = NewInteractee;
}

void ALevelSelector::EndInteraction()
{
CharacterUsing->ExitInteraction();
CharacterUsing = nullptr;
}
```

Finally, I added the PrimaryInteract function. This simply fires a trace from the player's mouse position in the mouse direction. which is then checked to see if it hits a component. If it does, it queries the PlanetVisuals array to find a matching component, then calls SetupLevel based on the component found.

I then created a test sector for the selector to use.

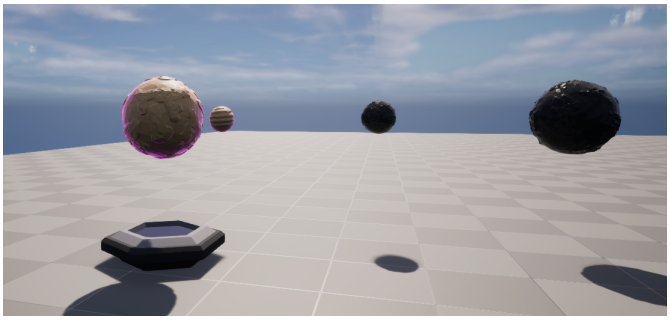

Figure 32. LevelSelector with the TestSector data. Each planet is a different level in the sector

Next I updated the LevelController by first no longer calling SetupLevel, as that is now handled by the LevelSelector. Next, I addad the ClearLevel function, which simply destroys all objects in each mechanic array (excluding the ExitDoor) and then empties the arrays. I plan on updating this by converting the spawned mechanics into either different mechanics or into item pools under the level.

```
void ALevelController::ClearLevel()
{
// Check if a level is spawned.  If so, remove it
if (LevelMesh->GetStaticMesh()) {
 LevelMesh->SetStaticMesh(nullptr);

 for (ARecordPad* RecordPad : RecordPads) {
 RecordPad->Destroy();
}
RecordPads.Empty();

for (AMechanicObject_Input* input : InputMechanics) {
 input->Destroy();
}
InputMechanics.Empty();

for (AMechanicObject_Output* output : OutputMechanics) {
 if (output->ObjectName != "OutputDoor") {
 output->Destroy();
}}
OutputMechanics.Empty();
}}
```

Finally, SetupLevel now clears the level before searching the data table, as well as updating the clip plane of the Entrance and Exit doors and setting ObjectToSpawn on the OutputMechanics.

Next, I worked on making the character interact properly with the LevelSelector. First, I added a pointer to the Controller Player which is set in BeginPlay to avoid multiple casts. Next, I added an empty pointer to a LevelSelector, used when interacting with one.

Following this, I overrided Interact to now query the trace against the LevelSelector. If the trace does hit a selector, it is casted to, the player's view target is blended with the selectors camera and the CurrentLevelSelector is set to active. Additionally, if the player is currently using a level selector, interacting again will exit

```
void ACharacter_Default::Interact()...
// If the object has neither tag, check if it is a
 LevelSelector after checking if this character has the
 tag of CanAccessLevelSelect
else if (TraceHit.GetActor()->IsA(ALevelSelector::
 StaticClass()) && ActorHasTag("CanAccessLevelSelect")) {
 // Cast to the LevelSelector, blend between the two cameras and
 ↪  call
 CurrentLevelSelector =
 ↪  Cast<ALevelSelector>(TraceHit.GetActor());
 PC->bShowMouseCursor = true;
 PC->SetViewTargetWithBlend(CurrentLevelSelector, 2.0f);
 bMovementDisabled = true;
 CurrentLevelSelector->SetActive(this);
}
```

I also added the ExitInteraction, which resets the player's view to their FirstPersonCamera when called.

```
void ACharacter_Default::ExitInteraction()
{
CurrentLevelSelector = nullptr;
bMovementDisabled = false;
PC->bShowMouseCursor = false;
PC->SetViewTargetWithBlend(this, 2.0f);
}
```

Next was a small amount of cleanup in other classes, including: fixing a playback bug in RecordPad by moving the player to the record pad on recording start, swapped the parent class of WorldObjectStart from MechanicObject Parent to MechanicObject Output and moving UpdateClipPlane from protected to public in LevelDoor.

Finally, I removed any unused test merged level meshes and created three new tutorial levels to showcase a small amount of the game. However, some issues arose with the LevelCreationTool, so I introduced an output message to the widget alongside some error messages to desribe what went wrong.

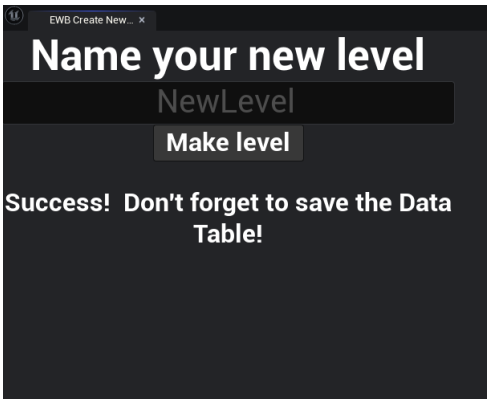

Figure 33. The new output message in the widget. Error messages also appear here if necessary.

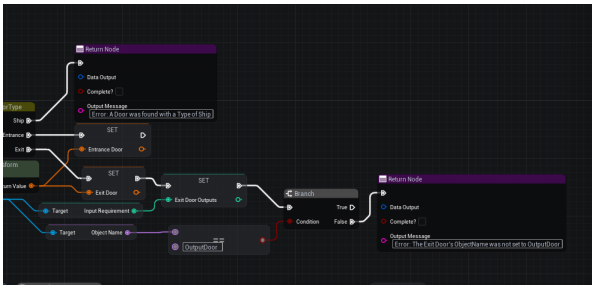

Figure 34. Example of an output message.

Bibliography

References

Epic Games. (2014) Unreal Engine 4/5 [Game Engine]

Epic Games. 'Enhanced Input', Available at: https://dev.epicgames.com/documentation/en-us/unreal-engine/enhanced-input-in-unreal-engine (Accessed: 19 April 2024)

Epic Games. 'UPhysicsHandleComponent', Available at: https://docs.unrealengine.com/4.26/en-US/API/Runtime/Engine/PhysicsEngine/UPhysicsHandleComponent/ (Accessed: 19 April 2024)

Epic Games. 'WidgetInteractionComponent', Available at: https://docs.unrealengine.com/4.27/en-US/InteractiveExperiences/UMG/UserGuide/WidgetInteraction/ (Accessed: 21 April 2024)

Dev Squared. 'How to Create Portals in Unreal Engine', Available at :https://www.youtube.com/watch?v=6BT-Ux56KBs&list=PLlYisyZ--cm_wsMGpy2C9Ewwm--jsv5Cm&index=1 (Accessed: 1 May 2024)