001

Dore's Pub
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 Player Movement

Commit

I began the project by working on the player movement, including the player's camera controls and introducing a basic UI setup.

a - Player Controller

I began with the Player Controller. As with my previous projects, I wanted to setup Unreal's Enhanced-Input in the controller, which would then call functions on the player character when possessed. I started the same way as before, with an UDataAsset to hold pointers to the individual UInputAction assets by the engine to map player inputs.

```
// An example from the Input_ConfigData class
UCLASS()
class DORESPUB_API UInput_ConfigData : public UDataAsset
{
        GENERATED_BODY()

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

EnhancedInput also requires a InputMappingContext asset to function alongside multiple UInputActions, so I created one for the player and added all the current InputActions to it. Both of these assets are found and stored as pointers in the Player-Controller class for later use via ConstructorHelpers.

```
APlayer_Controller::APlayer_Controller()
{
        // Find the input objects
        // First, find the Input Context Data
        ConstructorHelpers::FObjectFinder<UInput_ConfigData>
    ICDObject(TEXT("/Game/Inputs/ICD_DoresPub"));
        if (ICDObject.Succeeded()) { InputConfig =
        ↪  ICDObject.Object; }

        // Last, find the Input Mapping Context
        ConstructorHelpers::FObjectFinder<UInputMappingContext>
    IMCObject(TEXT("/Game/Inputs/IMC_DoresPub"));
        if (IMCObject.Succeeded()) { InputMapping =
        ↪  IMCObject.Object; }
}
```

Finally, the inputs are bound to functions in the controller via the SetupInputComponent function, starting by checking if the assets were stored correctly. If so, the local player subsystem is collected, with any mappings already in use cleared. Finally, the inputs are bound in the EnhancedInputComponent via BindAction delegates.

```
void APlayer_Controller::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
    <UEnhancedInputLocalPlayerSubsystem>(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,
        ↪  &APlayer_Controller::MoveX);
```

I decided to bind the inputs in the controller rather than the character as all of these inputs should be available to the player at any time. If any input is only to be used by one character, such as if I was to create a spectator character, I would instead add inputs to that character class instead.

b - Player Character

Next, I implemented the Player Character. Currently, it only requires two components - A spring arm component and a camera component. The spring arm is attached to the capsule component of the character, while the camera is attached to the spring arm. I also created functions to handle each of the inputs from the controller - placed in the public section so they can be called outside the character.

```
// An example from the Player_Character class
/// -- Inputs --
// Called to make the player move on the X axis
↪  (forward/backward)
void MoveX(float AxisValue);
```

These functions were then scripted to handle the events required by the function - the MoveX and MoveY input function first, moving the player on the X or Y axis. RotateCameraByStep was next, where the current rotation amount is stored separately as an int and increased or decreased based on the input - this value is then multiplied by the RotationStepAmount and set as the control's Yaw value.

```
void APlayer_Character::RotateCameraByStep(bool
↪  bRotateClockwise)
{
// If the camera is to rotate counter clockwise, decrement the
↪  current step
if (!bRotateClockwise) {
        CurrentStep--;
}
// Else, increment the step if the camera is to rotate
↪  clockwise
else {
        CurrentStep++;
}
// Then set the rotation based on the current step * the
↪  rotation step
PC->SetControlRotation(FRotator(0.0f, CurrentStep *
↪  RotationStep, 0.0f));
}
```

ZoomCameraByStep simply increases/decreased the spring arm's target arm length by a different step amount, checking if the value exceeds an upper or lower bound. If they do, the arm length is then set to the relevant bound.

```
void APlayer_Character::ZoomCameraByStep(bool bZoomIn)
{
// Check if the player wants to zoom in or out
if (bZoomIn) {
        // If they want to zoom in, then remove steps from the
        ↪  CameraSpringArm not going below the ZoomMin
        CameraSpringArm->TargetArmLength -= ZoomStep;
        if (CameraSpringArm->TargetArmLength < ZoomMin) {
                CameraSpringArm->TargetArmLength = ZoomMin;
        }
}
else {
        // If they want to zoom out, then add steps to the
        ↪  CameraSpringArm not going above the ZoomMax
        CameraSpringArm->TargetArmLength += ZoomStep;
        if (CameraSpringArm->TargetArmLength < ZoomMax) {
                CameraSpringArm->TargetArmLength = ZoomMax;
        }
}
}
```

Next was connecting the controller to the character - via casting in the OnPossessed function and storing a pointer to the character for use in each of the input functions. Each function checks that the pointer is set before calling a function on the character to avoid calling on an invalid pointer.

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

// Cast to the character pawn and store it
Character = Cast<APlayer_Character>(GetPawn());
}
```

Finally, I added a utility function FireTraceToActor, which simply fires a trace from the camera to the location and direction of the player's mouse, returning the FHitResult of the trace. This function will primary be used by any primary inputs.

```
FHitResult APlayer_Character::FireTraceToActor()
{
// Fire a line trace infront of this camera
FHitResult TraceHit;
FCollisionQueryParams TraceParams;
TraceParams.AddIgnoredActor(this);

// Mouse Location/Direction Vectors
FVector MouseDir, MouseLoc;

// Get the player's mouse location in world space
PC->DeprojectMousePositionToWorld(MouseLoc, MouseDir);

bool InteractTrace =
↪  GetWorld()->LineTraceSingleByChannel(TraceHit, MouseLoc,
↪  MouseLoc + (MouseDir * 4000), ECC_WorldDynamic,
↪  TraceParams);
//DrawDebugLine(GetWorld(), MouseLoc, MouseLoc + (MouseDir *
↪  2000), FColor::Red, false, 5, 2, 5);

return TraceHit;
}

// Example of Controller Input calls
void APlayer_Controller::MoveX(const FInputActionValue& Value)
{
// Check if the Character Pointer has been created successfully.
↪  If so...
if (Character) {
        // Call the MoveX command on the character
        Character->MoveX(Value.Get<float>());
}
}
```

c - Simple UI and Misc Assets

Finally, I setup the UI to be modified at a later date. The UI is collected by ContrustorHelpers and

added to the player viewport in the contructor, then casted to and stored as a pointer in BeginPlay. This UI has no functionality at the moment, but will be used in a later version.

I also created a gamemode to use the new controller and character - named GMB-DoresPub.

d - Reflection

I think this player movement works very well. The player can navigate over the environment easily, as well as perform all the basic tasks they would want from a top-down camera. Most of the basic movement features were inspired from real-time strategy games such as League of Legends (Riot Games, 2009) and Halo Wars (Ensemble Studios, 2009), however without the feature to move the camera by moving the mouse to the edges of the screen. I have implemented a feature like this in a previous project of mine, so the basic code is there.

Figure 1. League of Legends top down camera. The player cannot move the camera with WASD, unlike my project

2 Build Tool MK1

Commit

The next feature I worked on implementing was the basic Build Tool. This version of the build tool was designed to get the basic feature in - displaying what the player was trying to build in the world when they dragged over an area - then building the wall when they released the primary mouse button.

a - Inputs and Controller

I started by adding a few inputs to the InputConfigData to allow the player to change from no tool (dubbed "Default Mode" for the time) and the Build Tool. I also added inputs to allow the player to increase/decrease their grid snapping size and also updated the PrimaryAction input to have two triggers - one for when the input is first pressed and one for when it is released.

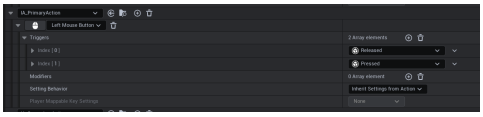

Figure 2. The PrimaryAction input in the InputMapping. As stated above, it has two triggers - Pressed and Released.

Next, I implemented the new inputs in the PlayerController by adding new functions and binding them in SetupInputComponent as before. I also added a utility function to check if an input value from a bind was 1 (a positive input) or -1 (a negative input), returning true or false based on if the value was positive. I also enabled the mouse cursor to be visible on the screen.

```
bool APlayer_Controller::GetInputValueIsPositive(float
↪  InInputValue)
{
if (InInputValue == 1) {
        return false;
}
return true;
}

// GetInputValueIsPositive is used in ZoomCamera to denote if
↪  the input should increase or decrease the value

void APlayer_Controller::ZoomCamera(const FInputActionValue&
↪  Value)
{
// Check if the Character Pointer has been created successfully.
↪  If so...
if (Character) {
Character->ZoomCameraByStep(GetInputValueIsPositive(Value.Get<float>()));
}}
```

b - Character Improvements

Next, I implemented the new inputs on the character. As before, each input is given it's own function which is called by the controller when input is

detected. However, swapping tools was given one function SwapCurrentActiveTool which takes in an int argument that is passed into a switch. The switch then calls a function on the Player-Tools class.

```
void APlayer_Character::SwapCurrentActiveTool(int NewTool)
{
// Swap to the new tool based on the input
switch (NewTool) {
case 0: // Default Tool
        PT->SwapTool(EToolType::Default);
        break;

case 1: // Build Tool
        PT->SwapTool(EToolType::Building);
        break;

default:
        break;
}}
```

I also implemented a world bounds system, limiting the area the player can edit in the future. The bounds are simply a FBox2D which is stored in the player (with a small 1uu (Unreal Unit) buffer so the player can still edit on the extent outside). To visualize the world bounds, I added a indefinite DrawDebugBox which draws the extents in the world. The box can also be updated in game by calling UpdateWorldBounds.

```
void APlayer_Character::UpdateWorldBounds(float X, float Y)
{
// Clear the world bounds if they exist already
if (WorldBounds.bIsValid) {
        WorldBounds.Init();
}

// Then create a FBox2D from the new points, with Min being the
↪  world origin at (0, 0)
// Add a little buffer so the player can still build on the
↪  extent outside
WorldBounds = FBox2D(FVector2D(-1.0f, -1.0f), FVector2D(X + 1,
↪  Y + 1));

// Draw it too
DrawDebugBox(GetWorld(), FVector(WorldBounds.GetCenter(),
↪  1.0f),
 FVector(WorldBounds.GetExtent(), 1.0f), FQuat(),
 FColor::Blue, true, -1.0f, 0U, 3.0f);
}
```

Figure 3. The WorldBounds debug box. This will be improved to be a texture or widget component in future, but the box fills the current needs

Next, I added a grid snapping system. Simply, the system takes in the mouse's position and rounds it to the closest multiple of the chosen snap distance. The available snapping values are stored in the TMap GridSnapValues, with the selected value stored as the int CurrentGridSnapValue as the index value. The snapping system is implemented in the Player-Tools instead of the Player-Character as it will be used there instead.

I also added some utility functions for the Player-Tools to use to get certian properties from the Player-Character class. These being GetPC - returning the Player-Character, GetCurrentGridSnapValue - returning the selected grid snap value, GetIsPointInsideBounds - which returns true if a point is inside the set world bounds, GetCurrentMoney - which returns the player's current money total, and UpdateMoney - called to update the player's money.

```
bool APlayer_Character::GetIsPointInsideBound(FVector Point)
{
        return WorldBounds.IsInside(FVector2D(Point));
}
```

Finally, I performed some cleanup of the previous elements added last time, by slightly modifying the camera angle and removed the spring arm's collision test to better suit it to the style of game play.

c - Player Tools class

The Player Tools was next, which is a separate class which stores all of the data related to the player's tools, such as the Build Tool, Default Tool and Object Tool. It consits of two components - a root SceneComponent and a WidgetComponent used to display the BuildToolArrow (which for now is simply a white box denoting where the player is pointing).

Figure 4. The BuildToolArrow. Currently has no functionality other than a simple white box.

First, I implemented the UpdateToolRotation function - used to update the rotation of the class to face the player's camera, instead of it facing (0, 0, 0) at all times. Next was adding the ability to swap tools. I began by creating the EToolType enum which contains values of the available tools to the player. I also implemented CurrentTool - which stores the current selected tool - and the function SwapTool, used to change the CurrentTool.

```
void APlayer_Tools::SwapTool(TEnumAsByte<EToolType> NewTool)
{
	CurrentTool = NewTool;

}
```

I also moved the FireTraceToActor function from the Player-Character class to the Player-Tools class, as it will not be used by the Player-Character. I also added the utility function GetNearestMultiple, which takes in two arguments, a Input and a Multiple to round too. The function simply returns the nearest multiple to the input, or 0 if the Multiple is 0. This is primary used by the grid snapping system.

```
float APlayer_Tools::GetNearestMultiple(float Input, int
↪  Multiple)
{
	if (Multiple == 0) {
		return Input;
	}
	return round(Input / Multiple) * Multiple;
}
```

Finally, I created a Blueprint class and added a ClassFinder that finds that Blueprint class in the Player-Character, which is then spawning during the BeginPlay function. Pointers between the Tools and Character class are also created, allowing the two classes to communicate between each other.

d - UI

The UI was then next area I targeted, as it was to be simple in this version. I added two buttons and a text box to allow the player to see the current selected grid snap value and change it among the available snapping values. I also added a text box that displays the current selected tool and implemented UpdateUIToCurrentTool, to update this text box when the tool was changed.

```
void UUI_Player_Master::UpdateUIToCurrentTool(int NewTool)
{
// Swap to the new tool based on the input
switch (NewTool) {
case 0: // Default Tool
	CurrentToolText->SetText(FText::FromString
  (FString::Printf(TEXT("Default Tool"))));
	break;

case 1: // Build Tool
	CurrentToolText->SetText(FText::FromString
  (FString::Printf(TEXT("Building Tool"))));
	break;

default:
	break;
}}
```

I also added a call to UpdateUIToCurrentTool when the SwapTool function is called in Player-Tools.

e - The BuildToolDisplay

Now was actually drawing with the build tool, displaying it to the player and finally spawning it in the world. This is where the BuildToolDisplay comes in.

I started by adding a default root SceneComponent, so I could modify the component easily in the world, as well as disabling the Tick function as it isn't used by this class. Next, I added a FBuildingData struct used to output the information from the BuildToolDisplay. It contains the mesh of the wall, it's location and it's rotation.

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

FBuildingData();
FBuildingData(UStaticMesh* NewMesh, FVector NewLocation,
↪  FRotator NewRotation);

~FBuildingData();

public:
// The mesh of the SMC
UPROPERTY(EditAnywhere, BlueprintReadWrite)
UStaticMesh* Mesh;

// The location of the SMC
UPROPERTY(EditAnywhere, BlueprintReadWrite)
FVector Location;

// The rotation of the SMC
UPROPERTY(EditAnywhere, BlueprintReadWrite)
FRotator Rotation;
};
```

Next, I added some utility functions that will be used by the BuildToolDisplay. These are: UpdateDisplayValidity - changes the colour of the material instance based on if the display is valid or invalid, GetDisplayWallsInUse - returns the amount of static mesh components are in use based on their set mesh (2 for wall, 1 for half wall, 0 for empty/pillar), GetDisplayData - returns the data for all static mesh components in use via struct FBuildingData, ClearBuildDIsplay - simply sets all static mesh components

meshes to nullptr, InitializeMaterial - a BlueprintImplementableEvent that initalizes the material instance for the static mesh components and UpdateMaterial - a BlueprintImplementableEvent that updates the material instance.

```
int APlayer_BuildToolDisplay::GetDisplayWallsInUse()
{
int amn = 0;
for (int i = 0; i < Total; i++) {
        if (SMCPool[i]->GetStaticMesh() == WallMesh) {
                amn += 2;
        }
        else if (SMCPool[i]->GetStaticMesh() == HalfWallMesh) {
                amn++;
        }
}
return amn;
}

TArray<FBuildingData>
↪  APlayer_BuildToolDisplay::GetDisplayData()
{
TArray<FBuildingData> out; FBuildingData curr;

for (UStaticMeshComponent* i : SMCPool) {
        if (i->GetStaticMesh() != nullptr) {
                curr.Mesh = i->GetStaticMesh();
                curr.Location = i->GetComponentLocation();
                curr.Rotation = i->GetComponentRotation();
                out.Add(curr);
        }
}
return out;
}
```

Next was the drawing function GenerateNewBuildDisplay. It starts by moving the BuildToolDisplay to the StartPosition, then calculates how many static mesh components are needed by calculating the difference between the start and end X positions, making the output absolute (always pos) then dividing that by the wall length (250 uu) and rounding up with ceil. It then repeats with the Y axis.

```
void APlayer_BuildToolDisplay::GenerateNewBuildDisplay(FVector
↪  StartPosition, FVector EndPosition)
{
// Move the BuildToolDisplay to the StartPosition
SetActorLocation(StartPosition);

// Start by calculating how many static mesh components are
↪  needed
int XRequires = 0;        int YRequires = 0;

// Calculate how many are needed for a single x edge (will
↪  double for a full box)
// By calculating the difference between the start and end
↪  position, making the output absolute (always pos)
// Then dividing that by the wall length (250 uu) and rounding
↪  up with ceil
XRequires = ceil(fabs(EndPosition.X - StartPosition.X) /
↪  WallSize);

// Do the same to the Y axis
YRequires = ceil(fabs(EndPosition.Y - StartPosition.Y) /
↪  WallSize);
```

Next, it calculates the total needed + 4 for the pillar corner blocks. If both X and Y require no walls, then don't add any to the required total, as no pillar blocks are needed.

```
// Then calculate the total needed + 4 for the pillar corner
↪  blocks
// If both X and Y require no walls, then don't add any to the
↪  required total
Total = (XRequires * ((YRequires != 0) ? 2 : 1)) + (YRequires *
↪  ((XRequires != 0) ? 2 : 1));
Total = ((Total != 0) ? Total + 4 : 0);
```

Next it checks how many StaticMeshComponents are required to complete the display, calling the AddNewStaticMeshComponent function to add new ones if required. All of the StaticMeshComponents spawned by this class are stored in a item pool named SMCPool, which then are modified when needed.

AddNewStaticMeshComponent starts by calculating the int that the new SMC will be named by adding one to the total amount in the SMCPool. It then creates a new SMC, registers it and updates the meshes material to the dynamic mateiral instance, finishing by adding it to the SMCPool. Finally, checks if enough SMC's have been created for the requirement. If not, the function recurses until the target is reached.

The BuildToolDisplay utilizes an item pool of StaticMeshComponents to display the current output of the build tool. Any unused SMC's are cleared and left at their last location until they are needed. I think this is a good way of utilizing a non-widget display, but could show some limitations if the player decides to try and make a huge box in the world.

```
// Add enough static mesh components to match the amount needed
if (Total > SMCPool.Num()) {
        AddNewStaticMeshComponent(Total);
}

void APlayer_BuildToolDisplay::AddNewStaticMeshComponent(int
↪  Target)
{
int NewNumber = SMCPool.Num() + 1;

UStaticMeshComponent* NewMeshComp =
↪  NewObject<UStaticMeshComponent>(this,
↪  FName(*FString::Printf(TEXT("Wall Mesh %i"), NewNumber)));
NewMeshComp->RegisterComponent();
NewMeshComp->SetMaterial(0, BuildToolMaterial);
NewMeshComp->SetMaterial(1, BuildToolMaterial);
```

```
NewMeshComp->SetCollisionEnabled(ECollisionEnabled::NoCollision);
SMCPool.Add(NewMeshComp);

// Check if there is now enough component sets added.  If not,
↪  recurse
if (SMCPool.Num() < Target) {
        AddNewStaticMeshComponent(Target);
}}
```

After enough SMCs have been created and added to the item pool, both the X axis and Y axis are checked to see if they require some SMCs. If only one axis requires them, then a wall is only created on that axis. If both do, then two of each wall is created. The functions CreateXWall and CreateYWall handle these requirements.

```
// Check if the X axis requires 0 SMC while Y requires some
↪  (build a wall on the Y axis only)
if (XRequires == 0 && YRequires != 0) {
        CreateYWall(StartPosition.Y, EndPosition.Y,
        ↪  StartPosition.X, YRequires, 0);
        CreatePillars(StartPosition, EndPosition);
}

// Check if the Y axis requires 0 SMC while X requires some
↪  (build a wall on the X axis only)
else if (XRequires != 0 && YRequires == 0) {
        CreateXWall(StartPosition.X, EndPosition.X,
        ↪  StartPosition.Y, XRequires, 0);
        CreatePillars(StartPosition, EndPosition);
}

// Else, check that both are not 0
        else if (XRequires != 0 && YRequires != 0) {
        // Start with the X walls, then the Y walls
        CreateXWall(StartPosition.X, EndPosition.X,
        ↪  StartPosition.Y, XRequires, 0);
        CreateXWall(StartPosition.X, EndPosition.X,
        ↪  EndPosition.Y, XRequires, XRequires);

        CreateYWall(StartPosition.Y, EndPosition.Y,
        ↪  StartPosition.X, YRequires, (XRequires * 2));
        CreateYWall(StartPosition.Y, EndPosition.Y,
        ↪  EndPosition.X, YRequires, (XRequires * 2) +
        ↪  YRequires);

        CreatePillars(StartPosition, EndPosition);
}
```

CreateXWall calculates which point is the start between the supplied Start and End points, followed by placing a SMC at each point every 250uu for the walls size. Finally, the final wall is checked to see if it needs to be changed to a half wall. CreateYWall performs the same tasks but on the Y axis instead.

```
void APlayer_BuildToolDisplay::CreateXWall(float Start, float
↪  End, float Y, int WallSegments, int StartWallSegment)
{
// Calculate which point is the start
int WallStart = (End - Start < -1) ? End : Start;
int WallEnd = (End - Start < -1) ? Start : End;

// For the amount X requires, place a SMC every 250 uu
for (int i = 0; i < WallSegments; i++) {
        SMCPool[StartWallSegment +
        ↪  i]->SetRelativeLocation(FVector(WallStart +
        ↪  (WallSize * i), Y, 1.0f));
        SMCPool[StartWallSegment +
        ↪  i]->SetRelativeRotation(FRotator(0.0f, 0.0f,
        ↪  0.0f));
                SMCPool[StartWallSegment +
                ↪  i]->SetStaticMesh(WallMesh);
}

// For the final wall, check if it needs to be a half wall
if (fabs((WallEnd - WallStart) - ((WallSegments - 1) *
↪  (WallSize))) == WallSize / 2) {
        SMCPool[StartWallSegment + WallSegments -
        ↪  1]->SetStaticMesh(HalfWallMesh);
}}
```

Originally, CreateX/YWall could create walls in either direction, by first calculating a DirectionMultiplier based on if the end point minus the start point was less than -1 (int DirectionMulti = (End.Y - Start.Y ¡ -1) ? -1 : 1;) This directional multiplier was then used in the placement calculations to modify the direction the SMC's were placed in.

Finally, any unused SMCs in the SMCPool are hidden by setting their mesh to nullptr;

```
// Finally, clear the excees SMC
for (int j = Total; j < SMCPool.Num(); j++) {
        SMCPool[j]->SetStaticMesh(nullptr);
}
```

As with the Player-Tools class before, a Blueprint class of BuildToolDisplay is created then found via a ConstructorHelper and stored in the Player-Tools class. It is then spawned and setup in on BeginPlay.

f - Connecting Tools to Display

Next, I connected the BuildToolsDisplay to the Player-Tools class. On Tick in Player-Tools, If the current tool is the Build tool, then fire a trace to the mouse's position, updating the tools location where the trace hits if it is inside of the WorldBounds. Also snap it to the building snapping distance (half of a wall size) and update the LastPosition and call any related functions. If the ClickPosition is valid (not (-1, -1, -1), then update the BTD and check if they can afford the current data. If they can, then check if MouseLocation is inside the WorldBounds, updating the validity of the display based on the outcome.

```
// Check which tool the player is in
// If the current tool is the Build tool, then...
if (CurrentTool == EToolType::Building) {
// Fire a trace to the mouse's position, updating the tools
↪  location where the trace hits if it is inside of the
↪  WorldBounds
// Also snap it to the building snapping distance (half of a
↪  wall size)
FVector MouseLocation = FireTraceToActor().Location;

MouseLocation.X = GetNearestMultiple(MouseLocation.X, 125);
MouseLocation.Y = GetNearestMultiple(MouseLocation.Y, 125);
MouseLocation.Z = 1.0f;
SetActorLocation(MouseLocation);

// Update the LastPosition and call any related functions
if (MouseLocation != LastPosition) {
        LastPosition = MouseLocation;
        UpdateToolRotation();

        // If the ClickPosition is valid (not (-1, -1, -1), then
        ↪  update the BTD
        if (ClickPosition != FVector(-1, -1, -1)) {
                BTD->GenerateNewBuildDisplay(ClickPosition,
                ↪  LastPosition);

        // Check if it is valid (money)
        if (BTD->GetDisplayWallsInUse() * HalfWallCost <=
        ↪  PC->GetCurrentMoney()) {
                // Next, check if the MouseLocation is outside
                ↪  the zone.
                if (!PC->GetIsPointInsideBound(MouseLocation))
                ↪  {
                BTD->UpdateDisplayValidity(false);
                }
                else {
                BTD->UpdateDisplayValidity(true);
                }
        }
        else {
        BTD->UpdateDisplayValidity(false);
}}}}
```

I also implemneted SelectedToolPrimaryPressed and SelectedToolPrimaryReleased. When the primary tool input is pressed, it checks the current selected tool. If it is the BuildTool, the position the player has clicked at is inside the world bounds, rounding it to the BuildingBounds and starting drawing with the BuildToolDisplay if it is valid. This also enables the ability for the tick function to continue.

```
void APlayer_Tools::SelectedToolPrimaryPressed()
{
// If the current tool is the BuildTool, then...
if (CurrentTool == EToolType::Building) {
        // Check where the player has clicked is inside of the
        ↪  WorldBounds
        FVector testClickPos = FireTraceToActor().Location;
        if (PC->GetIsPointInsideBound(testClickPos)) {
                // If it is, then round it to the Building
                ↪  Bounds and start "drawing" with the
                ↪  BuildToolDisplay
                testClickPos.X =
                ↪  GetNearestMultiple(testClickPos.X, 125);
                testClickPos.Y =
                ↪  GetNearestMultiple(testClickPos.Y, 125);
                testClickPos.Z = 1.0f;
                ClickPosition = testClickPos;
                BTD->GenerateNewBuildDisplay(ClickPosition,
                ↪  ClickPosition);
}}}
```

When it is released, the BuildToolDisplay is checked to see if it actually contains any data. If it does, it is converted into FBuildingData structs via GetDisplayData and passed to the BuildingLevel class.

```
void APlayer_Tools::SelectedToolPrimaryReleased()
{
// If the current tool is the BuildTool, then...
if (CurrentTool == EToolType::Building) {
        // Check where the player has clicked is inside of the
        ↪  WorldBounds
        FVector testClickPos = FireTraceToActor().Location;
        if (PC->GetIsPointInsideBound(testClickPos)) {
                // Check if they have enough money for the
                ↪  building
                int cost = BTD->GetDisplayWallsInUse() *
                ↪  HalfWallCost;
                if (cost <= PC->GetCurrentMoney()) {
                        // If they do, then
                        GroundFloor->AddBuildingObjects(BTD->GetDisplayData());
                        PC->UpdateMoney(-cost);
                }
        }
        ClickPosition = FVector(-1, -1, -1);
        // Also clear the BTD
        BTD->ClearBuildDisplay();
}}
```

Figure 5. BuildToolMk1 in use. The green walls show that the current drawing is valid.

g - Spawning in the World

Finally, I added the World-BuildingLevel class. Each of these objects spawned in the world handles the objects, walls and data of a selected floor in a building, so if the player wants to add to the ground floor of their building they call functions on the ground floor object. However, only one BuildingLevel is used currently.

BuildingLevels utilize a SMCPool as the BuildToolDisplay, but also have the AddBuildingObjects function - which converts the FBuildingData structs from the BuildToolDisplay into static meshes that create the building.

```
void AWorld_BuildingLevel::AddBuildingObjects(TArray<struct
↪  FBuildingData> DataToBuild)
{
```

```
// Start by calculating the amount of SMC's required for the
↪  current build + the new build data
int Required; int Current = 0;
for (UStaticMeshComponent* i : SMCPool) {
        if (i->GetStaticMesh() != nullptr) {
                Current++;
        }
}
Required = Current + DataToBuild.Num();

// Spawn in any required SMC's
if (Required > SMCPool.Num()) {
        AddNewStaticMeshComponent(Required);
}

// Then add the data to the building
for (int i = 0; i < DataToBuild.Num(); i++) {
        SMCPool[Current +
        ↪  i]->SetStaticMesh(DataToBuild[i].Mesh);
        SMCPool[Current +
        ↪  i]->SetRelativeLocation(DataToBuild[i].Location);
        SMCPool[Current +
        ↪  i]->SetRelativeRotation(DataToBuild[i].Rotation);
        }
}
```

h - Reflection

I think the BuildToolDisplay works very well to the projects needs. It utilizes some methods to reduce load, such as an item pool instead of deleting and adding new object each time it is used, but does seem a little bit messy. I could convert the StaticMeshComponents to "thatstaticmeshclassthaticantremeberthenameof", but I have a limited knowlegde on how useful they could be in this situation.

I don't fully like the direction I have taken the Player-Tools class, as it could become very hard to read if each tool is added to the class in the same way. I could have separate classes for each tool and a parent tool with common functions, but this would require more objects to be spawned in the beginning of the game.

Overall, the tool works well as it does what it says on the tin - build walls designed by the player.

3 Build Tool Mk2

Commit

MK2's upgrade was to remove the display and spawning of new walls when they already exist in the world already. Imagine drawing a initial box, then drawing another room connected to it, you wouldn't need to spawn the walls where the boxes are connected.

a - Attempt One

I took two attempts at creating a system to fix this problem. Attempt one consisted of a check if a collision was detected between the components of the BuildToolDisplay and the BuildingLevel. This proved successful in most attempts, but would require extra calcuations as walls would collide with the corners and walls. Unfortunately, GetOverlappingComponents returns UPrimitiveComponent, requiring a cast to each overlapped object to compare the mesh and get the one being removed, massivly increasing memory usage.

```
// -- Attempt 1 - Fail --
bool
↪  APlayer_BuildToolDisplay::GetOverlappingPlacedMesh(UStaticMeshComponent*
↪  BuildToolComponent)
{
TArray<UPrimitiveComponent*> OverlapComps;

// Check if this is now colliding with a already placed mesh.
BuildToolComponent->GetOverlappingComponents(OverlapComps);
if (!OverlapComps.IsEmpty()) {
for (UPrimitiveComponent* i : OverlapComps) {
 if (i->IsA<UStaticMeshComponent>()) {
 // If it is a StaticMeshcComponent, cast it to SMC
  UStaticMeshComponent* OverlapMesh =
  ↪  Cast<UStaticMeshComponent>(i);

  // Compare if the two meshes are the same.  If they are, then
  ↪  clear this mesh
  if (BuildToolComponent->GetStaticMesh() ==
  ↪  OverlapMesh->GetStaticMesh() &&
  ↪  BuildToolComponent->GetOwner() !=
  ↪  OverlapMesh->GetOwner()) {
  UE_LOG(LogTemp, Warning, TEXT("The two meshes are the
  ↪  same"));
  BuildToolComponent->SetStaticMesh(nullptr);
  return true;
}}}}
return false;
}
```

My second attempt was alot more successful at fixing the issue, mainly through the use of GetWallObjectMeshAtPosition This sends the data of the SMC to compare from the BuildToolDisplay to the current BuildingLevel to compare, with multiple checks before any large math calculations between the two. This could cause more overall calculations which is one of the cosiderations that I thought about before implementing it.

b - Updating BuildToolDisplay

I started by adding a temporary pointer to the GroundFloor BuildingLevel object in the world, which is found and casted to during BeginPlay. I also simplified GenerateToolDisplay by converting the pillar creating code from being on each path to now having a separate check after a path has been selected. Basically, if any wall was created then pillars are created too.

Collision is also no longer removed when a new StaticMeshcComponent is created, rather instead a custom BuildingTool collision profile is set instead.

c - GetWallObjectMeshAtPosition

Next was creating the function that is used to compare the data in the BuildToolDisplay to the data in the BuildingLevel already created - GetWallObjectMeshAtPosition - implemented in World-BuildingLevel. It starts by iterating over all of the meshes in it's mesh pool, checking if any mesh has been set. Once one has been found with a set mesh, it's location is compared against the Location argument passed in from the BuildToolDisplay. If the locations match, then the meshes are checked to see if they are the same.

```
UStaticMeshComponent*
↪  AWorld_BuildingLevel::GetWallObjectMeshAtPosition
 (FVector Location, FVector ForwardVector, UStaticMesh* Mesh)
{
for (UStaticMeshComponent* i : SMCPool) {
 if (i->GetStaticMesh() != nullptr) {
  if (i->GetComponentLocation() == Location) {
   if (Mesh == i->GetStaticMesh()) {
```

Finally, the rotations are compared, via calculating the dot product between the forward vectors of the StaticMeshcComponents from both the BuildToolDisplay and BuildingLevel. If the calculations return 0, then the rotations are the same and the function returns the StaticMeshcComponent of the BuildingLevel.

```
// Find the dot prod between the two rotators
FVector VecA = i->GetForwardVector();
VecA.Normalize();

FVector VecB = ForwardVector;
VecB.Normalize();

float OutAngle =
↪  FMath::RadiansToDegrees(acosf(FVector::DotProduct(VecA,
↪  VecB)));
if (OutAngle == 0.0) {
 return i;
}}}}}
return nullptr;
```

GetWallObjectMeshAtPosition is used in both the CreateXWall and CreateYWall functions when placing a new static mesh. If one is found at the same location, the mesh set is instead cleared.

```
void CreateYWall()...
if (fabs((WallEnd - WallStart) - ((WallSegments - 1) *
↪  (WallSize))) == WallSize / 2) {
 SMCPool[StartWallSegment + WallSegments -
 ↪  1]->SetStaticMesh(HalfWallMesh);
 }
}

UStaticMeshComponent* overlapSMC =
↪  GroundFloor->GetWallObjectMeshAtPosition(FVector
 (WallStart + (WallSize * i), Y, 1.0f),
 ↪  SMCPool[StartWallSegment + i]->GetForwardVector(),
 SMCPool[StartWallSegment + i]->GetStaticMesh());

if (overlapSMC) {
 if (overlapSMC->GetStaticMesh() == WallMesh) {
  SMCPool[StartWallSegment + i]->SetStaticMesh(nullptr);
  }
 }
}

used--;
```

d - Reflection

I think this is implementation of data comparisons works well, if not a little complicated and memory intensive. When a new wall is placed with CreateX/YWall, this dot product calculation is created for every mesh that is in the same location and with the same mesh. Imagine if the player intends to draw a massive room over a few walls, the caluclations could get intense.

I plan to replace the system with a custom data system to handle the data of each wall at each point. This system could then be expaned to include floors and ceiling if I want. However, I am aiming to get a object placement in the game for the next version, so this data system will be put on hold for a short time

Figure 6. Mk2 in action. As the wall is already placed in the world, the system doesn't place an additional wall at the same location

4 Object Tool Mk1

Commit

For this version, I wanted to implement a object placement system from a previous prototype of the game that I have created (DoresPub-Proto). That version had a sortable list view UI element which allowed the player to sort through sections and subsections of selectable items and place them in the world.

a - Inputs and Rotation Snapping

I started in the same was as Build Tool Mk1, by setting the inputs to change to the Object Tool. I created a new InputAction, added it to the InputConfigData asset and updated the InputMappingContext with the new input. Following this, I also updated the controller with a new function to swap to the Object Tool.

I also decided to add a rotation snap value, to allow the player to snap object's rotation to a set value. It was implemented in the same was as the grid snapping system, with a unique TMap of snapping values that the player can change between and a separate int value controlling their current choice. I also added buttons and a text block to the UI to allow the player to change the value, while the text is updated when the player changes the value.

```
void APlayer_Character::ChangeRotationSnapSize(bool bIncrement)
{
if (bIncrement) {
 if (CurrentRotationSnapValue + 1 >= RotationSnapValues.Num() -
 ↪  1) {
  // Set to max grid snap value
  CurrentRotationSnapValue = RotationSnapValues.Num() - 1;
 }
 else {
  CurrentRotationSnapValue++;
  }
}
 else {
  if (CurrentRotationSnapValue - 1 <= 0) {
   // Set to max grid snap value
   CurrentRotationSnapValue = 0;
  }
  else {
   CurrentRotationSnapValue--;
 }
}

// Finally, update the GridSnapText on the UI
UI->UpdateRotationSnapText(GetCurrentRotationSnapValue());
```

Figure 7. The Rotation Snap buttons. It is placed below the GridSnap buttons

I also added two new uitility functions - one in Player-Character and one in UI-Master, both returning a pointer to the Player-Tools class when called.

b - Object Data

Before working on the actual tool itself, I implemented the data used by the system to allow the player to actually select a asset to place. First was the enum EObjectType, denoting the objects main category, with the FObjectData struct holding all of the information about an item alongside the object. In my original prototype, subcategories also utilized a custom enum, but I decided against this and instead used a FString instead as some categories didn't make sence for certain objects - such as "MultiSeat" for a fridge for example.

```
        UENUM(BlueprintType)
enum EObjectType
{
        DefaultType UMETA(DisplayName = "Default"),
        Seating UMETA(DisplayName = "Seating"),
        Tables UMETA(DisplayName = "Tables"),
        Count UMETA(Hidden),
};

ENUM_RANGE_BY_COUNT(EObjectType, EObjectType::Count)

USTRUCT(BlueprintType)
struct FObjectData : public FTableRowBase
{
public:
        GENERATED_BODY();

        FObjectData();
        ~FObjectData();


public:
        UPROPERTY(EditAnywhere, BlueprintReadWrite)
        FString Name;

        UPROPERTY(EditAnywhere, BlueprintReadWrite)
        FString Description;
...
```

EObjectType also utilizes the ENUM RANGE BY COUNT UE5 macro, which allows each of the values of the enum to be iterated over, for use in for loops and more. I also had to implement the struct FSubCats, as one of the limitations of TMaps and TArrays is that

they cannot be 2D (aka an array inside an array). Using a struct with an array is a workaround.

Finally, I created a data table using the FObjectData struct by extending from FTableRowBase. I added two test objects which allows me to make sure the system works properly, these will be replaced in future versions of the tool.

Figure 8. The Data Table object, with the two test objects added - and

c - UI Setup

The basic setup of the UI was my next target. Getting in the most important elements - such as multiple buttons, list views and more - was required before actually implementing interactivity.

I started by creating a new widget state for the object tool, where all of the object components would be located. This was then placed inside the widget switcher in the UI-Main class, allowing it to be swapped to when needed.

I added three TileView components and a button - one TileView displays available categories the player can select, another displays applicable sub categories and the last displays all items that match the categories selected. The button will allow the player to toggle rotation mode.

```
public:
/// -- Components --
// Tile View Object for displaying the category buttons
UPROPERTY(BlueprintReadOnly, meta = (BindWidget))
UTileView* CategoryTileView = nullptr;

// Tile View Object for displaying the sub category buttons
UPROPERTY(BlueprintReadOnly, meta = (BindWidget))
UTileView* SubCategoryTileView = nullptr;

// Tile View Object for displaying the inventory
UPROPERTY(BlueprintReadOnly, meta = (BindWidget))
UTileView* ObjectTileView = nullptr;

// Button to swap to/from rotation mode
UPROPERTY(BlueprintReadOnly, meta = (BindWidget), Category =
↪  "Grid Snapping")
UButton* RotationModeButton = nullptr;
```

For the item subcategories, I added a TMap of EObjectType/FSubCats which allows the user to denote the subcategories per main category. If a subcategory is required between two categories, it is simply placed in multiple map values.

Figure 9. The subcategory TMap. It is set to EditDefaultsOnly, meaning the player cannot modify it during gameplay

I also added some extra properties, including a pointer back to the master UI class and properties for the selected category and subcategory.

Next was the data objects and the buttons used by the ListViews. The data objects were placed into their own UI-ObjectData class which are then used by the list view when adding new data. Each object was given the BlueprintType identifier so it could be used in Blueprints.

```
UCLASS(BlueprintType, Blueprintable)
class DORESPUB_API UCategoryButtonData : public UObject
{
        GENERATED_BODY()

public:
        UCategoryButtonData();
        ~UCategoryButtonData();

        void SetupData(class UUI_Player_Object* UI,
        ↪  TEnumAsByte<EObjectType> NewCategory);

public:
        // Pointer back to the Object UI class
        UPROPERTY(EditAnywhere, BlueprintReadWrite)
        class UUI_Player_Object* ObjectUIState = nullptr;

        UPROPERTY(EditAnywhere, BlueprintReadWrite)
        TEnumAsByte<EObjectType> Category;

        UPROPERTY(EditAnywhere, BlueprintReadWrite)
        UTexture2D* Icon;
};
```

Each button was given it's own UUserWidget class which was then used as a parent for three Blueprint classes. Unlike UI elements previously added, all of the functionality needed to be added to the BP class as the ObjectListEntry interface is unavailable in C++.

When data is added to a list view, a set EntryWidgetClass is spawned inside the list which can then be interacted with. The event OnListItemObjectSet is also called in the object, which includes the data added to the list view, which can be casted to use the data.

As both the category and subcategory buttons are identical except with what variable they changed when pressed, I only created one button class to be used by

two lists. The class simply checks if it can cast to the CategoryData object first, which then casts to the SubCategoryData if the cast fails.

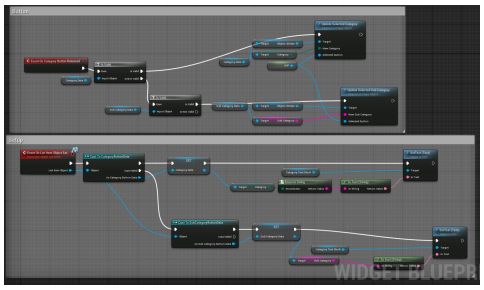

Figure 10. Casting to data. As stated above, both the category and subcategory lists use the same button

After casting, the buttons simply display the data contained - category buttons display the name of the category/subcategory and the item button displays a icon of the stored item.

Figure 11. The ItemButton displays the icon of the item contained in the button

d - Icon Creation Tool

Next I wanted to create a tool that would allow me to take a picture of an asset in-editor and create a texture asset for an icon.

My icon creation tool consists of two classes - an Editor Utility Widget and an Actor. Once opened, the Editor Utility Widget spawns the actor in the world. Selecting a StaticMesh from the Content Drawer displays the mesh on the actor, which then allows the user to rotate and zoom in/around the model selected with the buttons on the widget.

The user can enter a custom name for the widget in the TextBox at the top. Then pressing the MakeIcon button will the create a Texture2D asset via RenderTargetCreateStaticTextureEditorOnly, with an output message also being displayed if the icon was created successfully or not.

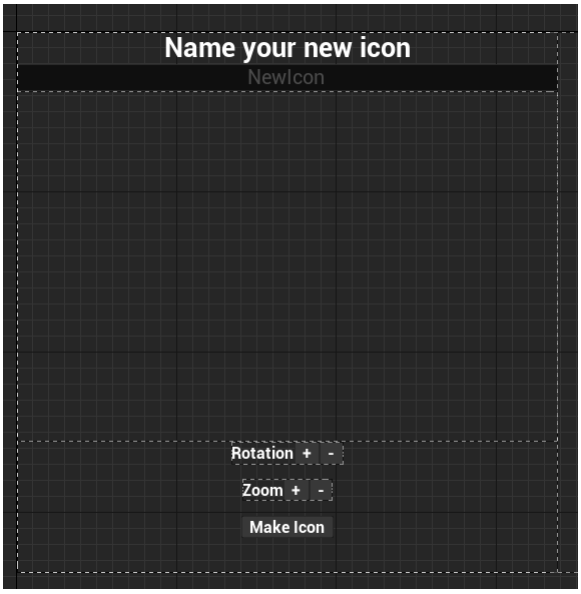

Figure 12. The Editor Utility Widget. As stated, the player can insert an icon name in the text box

It works by first detecting if the player has selected an asset in the content browser vai GetSelectedAssetsOfClass. If the array is not empty, the first one in the selecte array is casted to a StaticMesh, setting the SelectedMesh if it is valid.

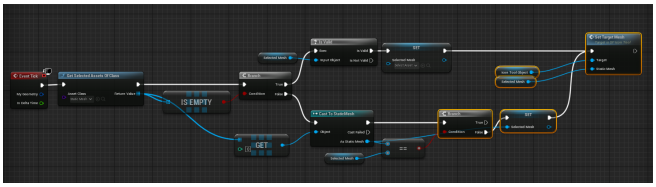

Figure 13. The detection code. This could be improved later if SkeletalMeshes also need icons

When the EnterButton is pressed, the text box is first checked if it is empty, returning a default name if so. Next, it checks if a mesh is actually selected, to stop any invalid icons being created. If no mesh is set an invalid ouptut message is returned to the player.

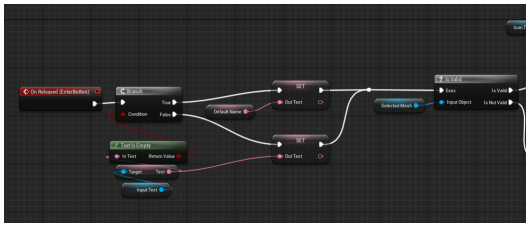

Figure 14. The default text check. If the text box is empty, then the default name of "NewIcon" is used

If a valid mesh is set, then an icon is created with the name inserted by the player (with a T- prefix) via the RenderTargetCreateStaticTextureEditorOnly. Finally, it gets the full path name of the new asset and returns it to the player as an output message.

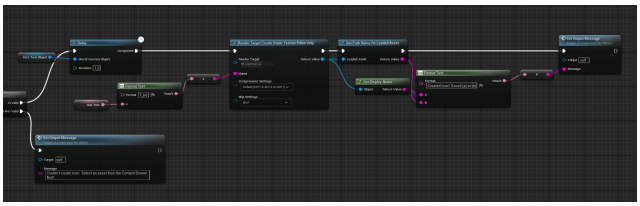

Figure 15. Image Creation. The RenderTargetCreateStaticTextureEditorOnly basically converts a frame from a render target to a texture.

I then used the tool to create two icons for my test assets.

e - UI functionality

Now that the UI was setup, functionality was my next goal. I started by working on the system that displays available items based on a selected category and subcategory. UpdateObjectTileView starts by clearing any items in the ItemTileView, then collects and stores all the row names of the ObjectDataTable in a TArray. Next, it figures out what items should be shown, based on SelectedCategory and SelectedSubCategory:

- If no category is selected, show all items.
- If a category is selected, but a sub-category is not selected, show all of the category items only.
- Else both a Category and SubCategory are selected, so only show items of the sub-category.

```
void UUI_Player_Object::UpdateObjectTileView()
{
// Start by clearing the TileView
ObjectTileView->ClearListItems();

// Store an array for all row names of the data table
TArray<FName> RowNames = ObjectDataTable->GetRowNames();

// Next, figure out what items should be shown.
// If no category is selected, show all items
if (CurrentSelectedObjectType == EObjectType::DefaultType) {
 for (FName i : RowNames) {

  ↪  AddItemToObjectTileView(*ObjectDataTable->FindRow<FObjectData>(i,
  ↪  ""), i);
 }
}

// If a category is selected, but a sub-category is not
↪  selected, show all of the category items only
else if (CurrentSelectedSubCategory == "") {
 for (FName i : RowNames) {
  FObjectData* Data = ObjectDataTable->FindRow<FObjectData>(i,
  ↪  "");
  if (Data->Category == CurrentSelectedObjectType) {

   ↪  AddItemToObjectTileView(*ObjectDataTable->FindRow<FObjectData>(i,
   ↪  ""), i);
  }
 }
}

// If both a Category and SubCategory are selected, then only
↪  show items of the sub-category
else {
 for (FName i : RowNames) {
  FObjectData* Data = ObjectDataTable->FindRow<FObjectData>(i,
  ↪  "");
  if (Data->Category == CurrentSelectedObjectType) {
   if (Data->SubCategory == CurrentSelectedSubCategory) {

    ↪  AddItemToObjectTileView(*ObjectDataTable->FindRow<FObjectData>(i,
    ↪  ""), i);
   }
  }
 }
}
}
```

Next was updating the available subcategory buttons with UpdateSubCategoryButtons. It starts by clearing the SubCategoryTileView, then finds the matching sub category index in the map. For each sub category create a USubCategoryButtonData and add it to the list view.

```
void UUI_Player_Object::UpdateSubCategoryButtons()
{
// Start by clearing the SubCategoryTileView
SubCategoryTileView->ClearListItems();

// Next, find the matching sub category index in the map
```

```
TArray<FString> FoundSubCats =
↪  SubCategories.FindRef(CurrentSelectedObjectType).X;

// For each sub category (plus the blank all category), create a
↪  USubCategoryButtonData and add it to the list view
for (FString i : FoundSubCats) {
 USubCategoryButtonData* NewSubCatObj =
 ↪  NewObject<USubCategoryButtonData>();
 NewSubCatObj->SetupData(this, i);
 SubCategoryTileView->AddItem(NewSubCatObj);
}
}
```

I added both UpdateSelectedCategory and UpdateSelectedSubCategory, which update SelectedCategory and SelectedSubCategory respectivly when a category button is pressed. If a button is re-selected again after being selected, the respective property is returned to the default. To actually add items to the ObjectListView I added AddItemToObjectTileView. It simply takes an ID argument, finds the row matching from the ObjectDataTable and adds the data to the list view as an UItemButtonData object.

```
void
↪  UUI_Player_Object::UpdateSelectedCategory(TEnumAsByte<EObjectType>
↪  NewCategory, class UUI_Player_Object_CatButton*
↪  SelectedButton)
{
// Update CurrentSelectedObjectType and upate the SubCatButtons
if (CurrentSelectedObjectType == NewCategory) {
 CurrentSelectedObjectType = EObjectType::DefaultType;
 CurrentSelectedSubCategory = "";
}
else {
 CurrentSelectedObjectType = NewCategory;
}

UpdateSubCategoryButtons();

// Then update the ObjectTileView
UpdateObjectTileView();
}
```

f - Selected Object

Finally, I implemented the ObjectTool to the Player-Tools. It simply consists of a StaticMeshcComponent which displays the selected object from the ItemListView. Functionality was added in the same was as the BuildTool, as a new path in the Tick function. If the player is in Object mode, it checks if the player is in Normal or Rotation mode. If they are in Normal, moves the ObjectToolMeshComponent to a suitable snap location based on the selected grid snap size, else it snaps the ObjectToolMeshComponent rotation to the nearest RotatingSnap based on the selected rotation snap size and the mouse's location as they are in Rotation Mode.

```
// If the current tool is the Object tool, then...
else if (CurrentTool == EToolType::Object) {
 // Check what mode the Object tool is in
 // If they are in Normal Mode
 if (!bInRotationMode) {
  // Fire a trace to the mouse's position, updating the tools
  ↪  location where the trace hits if it is inside of the
  ↪  WorldBounds
  // Also snap it to the building snapping distance (half of a
  ↪  wall size)
  FVector MouseLocation = FireTraceToActor().Location;

  MouseLocation.X = GetNearestMultiple(MouseLocation.X,
  ↪  PC->GetCurrentGridSnapValue());
  MouseLocation.Y = GetNearestMultiple(MouseLocation.Y,
  ↪  PC->GetCurrentGridSnapValue());
  MouseLocation.Z = 1.0f;
  SetActorLocation(MouseLocation);

  // Update the LastPosition and call any related functions
  if (MouseLocation != LastPosition) {
   LastPosition = MouseLocation;

        }
}
 // Else, if they are in Rotation Mode
 else {
  FRotator ObjectRotation =
  ↪  UKismetMathLibrary::FindLookAtRotation
   (ObjectToolMeshComponent->GetComponentLocation(),
   ↪  FireTraceToActor().Location);

  // Snap the rotation's yaw to the bounds, while also
  ↪  nullifying the roll and pitch
  ObjectRotation.Yaw = GetNearestMultiple(ObjectRotation.Yaw,
  ↪  PC->GetCurrentRotationSnapValue());
  ObjectRotation.Pitch = 0.0f;
  ObjectRotation.Roll = 0.0f;

  ObjectToolMeshComponent->SetWorldRotation(ObjectRotation);
 }
}
}
```

When the player released the primary action input, the object is placed in the world as a new StaticMeshcComponent.

Figure 16. An object in the world. Before placing, it follows the RotationSnap and GridSnap the player has set

g - Reflection

I think this version of the Object Tool is a good baseline that I can build upon in following versions. Compared to the prototype version I had created previously, it works much better as the objects available are not tied down to a set amount of subcategories across each catgeory as it doesn't use a subcategory enum. It will need a Mk2, as currently the objects are simple static meshes and not actual classes. Additionally, they don't interact with the money the player owns.

The icon editor tool is my second attempt at a Editor Utility class, which I think came out extremely well. The functionality provided by the widget allows me to successfully place selected objects in the frame of the icon and it update live based on where the asset is. It can definetly be improved upon however, such as new options to modify the camera's location, if an asset is offset.

5 Build Tool Mk3

Mark 3 of the Build Tool focused on rebuilding how the walls are stored on the World-BuildingLevel, as well as add new features such as a wall selection and the ability to erase walls placed in the world.

a - BuildData

Commit

Currently, the World-BuildingLevel had no idea what type of walls are placed on it, just where they are and their rotation. Additionally, a wall had no idea if it had neighbouring walls connecting to it, which I decided to fix.

The FBuildData was designed to handle all of the information about a point in the building. As a point can have a wall on both the X and Y axis, due to how the wall assets were created, FBuildData contains two pointers to a X and Y StaticMeshcComponent. It also has an origin vector so it can easily be located, booleans denoting if the walls are half walls and pointers to connecting FBuildData. These pointers, however, cannot be displayed in the Editor.

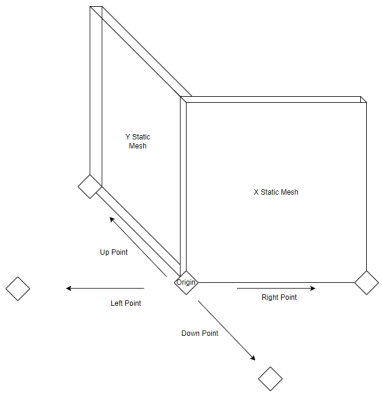

Figure 17. Visualization of FBuildData. Each point is connected via the FBuildData pointers

I then implemented it into the World-BuildingLevel. I started by adding the ability to find a FBuildData in the BuildData array via GetBuild-DataAtLocation. As each BuildData has an unique Origin vector, the array is iterated over comparing the Origin against the Location argument. If it matches, the index of the BuildData is returned, while if one isn't found an invalid int (-1) is returned instead.

```
int AWorld_BuildingLevel::GetBuildDataAtLocation(FVector
↪  Location)
{
 for (int i = 0; i < BuildData.Num(); i++) {
  if (BuildData[i].Origin == Location) {
   return i;
  }
 }
 return -1;
}
```

When all of the data from the BuildToolDisplay has been added to the BuildingLevel, the function Re-GenerateBuildData is called. It starts by clearing any BuildData currently in the array. Then, for every Stat-icMeshcComponent in the SMCPool that has a set static mesh, their location is checked with GetBuild-DataAtLocation to see if a FBuildData struct already exists for it. If one does UpdateBuildData is called, while AddNewBuildData is called instead if one isn't found.

```
void AWorld_BuildingLevel::ReGenerateBuildData()
{
 // Clear the BuildData array
```

```
 BuildData.Empty();

 // Now repeat for the remaining SMC's
 for (int i = 0; i < SMCPool.Num(); i++) {
  if (SMCPool[i]->GetStaticMesh()) {
   int j =
   ↪  GetBuildDataAtLocation(SMCPool[i]->GetComponentLocation());
   if (j != -1) {
        UpdateBuildData(SMCPool[i], j);
   }
   else {
        AddNewBuildData(SMCPool[i]);
   }
  }
 }
}
```

Both UpdateBuildData and AddNewBuildData complete similar tasks - UpdateBuildData updates a FBuildData already existing in the array while AddNewBuildData adds a new struct to the array. Once the update/addition has been comlpeted, the connecting point of the wall is also updated or added in the same way. If the wall is a X wall, then the UpPoint is modified. Else, the wall is Y so the RightPoint is modified instead.

```
void
↪  AWorld_BuildingLevel::AddNewBuildData(UStaticMeshComponent*
↪  SMC)
{
 // Initialize the variables for adding a new build data
 FBuildData NextBuildData; bool bIsHalfWall = false;
 NextBuildData.Origin = SMC->GetComponentLocation();

 FVector f = NextBuildData.Origin;
 bool bY = false;

 // Figure out if the wall is a full or half wall
 if (round(SMC->GetStaticMesh()->GetBoundingBox().GetSize().X)
 ↪  == 125) {
  bIsHalfWall = true;
 }

 // Next, figure out if the wall is an X or Y wall (x has a
 ↪  default rotator, while y has a 0, 90, 0 rotator)
 if (SMC->GetComponentRotation() == FRotator(0, 0, 0)) {
  NextBuildData.XStaticMeshComponent = SMC;
  f.X += bIsHalfWall ? 125 : 250;

 }
 else {
  NextBuildData.YStaticMeshComponent = SMC;
  f.Y += bIsHalfWall ? 125 : 250;
  bY = true;
 }
 BuildData.Add(NextBuildData);

 // Then figure out if we need to create a new FBuildData for
 ↪  the other connecting point
 int j = GetBuildDataAtLocation(f);
 if (j != -1) {
  if (!bY) {
   // If it does exist, update it
   BuildData[j].DownPoint = &BuildData[BuildData.Num() - 1];
   BuildData[BuildData.Num() - 1].UpPoint = &BuildData[j];
  }
  else {
   // If it does exist, update it
   BuildData[j].RightPoint = &BuildData[BuildData.Num() - 1];
   BuildData[BuildData.Num() - 1].LeftPoint = &BuildData[j];
  }
 }
 else {
  if (!bY) {
   // Else, create a new one
   NextBuildData = FBuildData();
   NextBuildData.Origin = f;
   NextBuildData.RightPoint = &BuildData[BuildData.Num() - 1];
   BuildData.Add(NextBuildData);
   BuildData[BuildData.Num() - 2].LeftPoint =
   ↪  &BuildData[BuildData.Num() - 1];
  }
  else {
   // Else, create a new one
   NextBuildData = FBuildData();
   NextBuildData.Origin = f;
   NextBuildData.DownPoint = &BuildData[BuildData.Num() - 1];
   BuildData.Add(NextBuildData);
   BuildData[BuildData.Num() - 2].UpPoint =
   ↪  &BuildData[BuildData.Num() - 1];
  }
 }
}
```

Finally, I disabled corners for now, as they would confuse the system as they also have the same location as the X and Y walls. This could be fixed by adding a corner StaticMeshcComponent pointer, but to reduce complexity I have decided against it for the time being.

I think the current implementation of the BuildData storage system works very well, as origins are not duplicated in the array like I wanted. Data is updated when needed and new arrays are successfully added. However, I need to add the ability to remove from the array with an erase mode. Additionally, I may want to add a better way of visualizing the BuildData array rather than using the Editor Properties tag.

b - Updating GetBuildDataAtLocation

Commit

Next, I updated the BuildToolDisplay to work with the new system. Before this, however, I decided to remove the usage of full size walls as they clash with how the BuildData array works. Each point is placed a full wall's length away (250uu) rather than half wall length, which would cause errors when trying to find a half wall's connecting point. This comes with the upside of simplifing the build system though, which will

make it easier to update and maintain over following versions.

The main update I completed was to the function GetWallObjectMeshAtPosition. Instead of using a dot product calculation to find the correct StaticMeshComponent in the pool, it instead uses GetBuildDataAtLocation to find the correct FBuildData and returns either the X or Y StaticMeshcComponent based on the rotation argument. I also updated CreateXWall and CreateYWall to use the updated function.

```
UStaticMeshComponent*
↪  AWorld_BuildingLevel::GetWallObjectMeshAtPosition(FVector
↪  Location, bool bOnXAxis)
{
 int i = GetBuildDataAtLocation(Location);
 if (i != -1) {
  if (bOnXAxis) {
   if (BuildData[i].XStaticMeshComponent) {
        return BuildData[i].XStaticMeshComponent;
   }
  }
  else {
   if (BuildData[i].YStaticMeshComponent) {
        return BuildData[i].YStaticMeshComponent;
   }
  }
 }
}
```

c - Erase Mode start

Commit

To start Erase Mode's implementation, I added a new function that toggles bInEraseMode true or false when called. I also added a check in SelectedToolPrimaryReleased checking if bInEraseMode is true, which gets the building objects like normal but calls RemoveBuildingObjects instead in the World-BuildingLevel.

CreateXWall and CreateYWall also required a small update, also adding an erase mode check. If the player is, they instead check to see if there is not a mesh at the new position, clearing the mesh if one isn't found.

```
if (!bInEraseMode) {
 if (GroundFloor->GetWallObjectMeshAtPosition(FVector(WallStart
 ↪  + (WallSize * i), Y, 1.0f), true)) {
  // If there is, clear the mesh
  SMCPool[StartWallSegment + i]->SetStaticMesh(nullptr);
 }
}
// Else, check if there isn't a mesh at the position
else {
 if
 ↪  (!GroundFloor->GetWallObjectMeshAtPosition(FVector(WallStart
 ↪  + (WallSize * i), Y, 1.0f), true)) {
  // If there isn't, clear the mesh
  SMCPool[StartWallSegment + i]->SetStaticMesh(nullptr);
 }
}
```

Finally, I added the ability to remove walls from the World-BuildingLevel via RemoveBuildingObjects. It collects the data from the BuildToolDisplay and for each element in the output array checks that the data to be removes actually exists in the BuildData. If it does, checks if a mesh exists on the rotation specified (X for a rotation of (0, 0, 0), Y for (0, 90, 0)), with the SMC is cleareda and the pointer removed if a mesh is found. Additionally, the pointer link to the corresponding FBuildData is removed, and if the FBuildData now has no pointers to other points, it is removed from the BuildData array,

```
void AWorld_BuildingLevel::RemoveBuildingObjects(TArray<struct
↪  FBuildingData> DataToRemove)
{
 int BuildDataIndex;

 // For each data to be removed...
 for (FBuildingData i : DataToRemove) {
  // First, check that the data to be removes actually exists in
  ↪  the BuildData
  BuildDataIndex = GetBuildDataAtLocation(i.Location);
  if (BuildDataIndex != -1) {
   // Next, check that a mesh exists on the rotation specified
   if (i.Rotation == FRotator(0, 0, 0)) {
        if (BuildData[BuildDataIndex].XStaticMeshComponent) {
         // If a mesh does exist on the X axis, clear the SMC,
         ↪  remove the pointer and update the BuildData on the
         ↪  up point

         ↪  BuildData[BuildDataIndex].XStaticMeshComponent->SetStaticMesh(null
         BuildData[BuildDataIndex].XStaticMeshComponent =
         ↪  nullptr;
         BuildData[BuildDataIndex].UpPoint->DownPoint =
         ↪  nullptr;
         if (!BuildData[BuildDataIndex].UpPoint &&
         ↪  !BuildData[BuildDataIndex].LeftPoint &&
          !BuildData[BuildDataIndex].DownPoint &&
          ↪  !BuildData[BuildDataIndex].RightPoint) {
          BuildData.RemoveAt(BuildDataIndex);
         }
        }
   }
```

d - Erase Button

Commit

I replaced the temporary method of swapping to erase mode after every primary build tool release to

a separate button on the UI. I also created an unique UI state for the BuildTool, and moved any BuildTool assets to it.

e - Selectable Walls

Commit

I decided a small amount of cleanup was required to keep the quality of my project going, so before I started updating the Build Tool more I cleaned up the commenting of the project. I also renamed the BuildingData files to BuildToolData, as well as moving FBuildToolData out of BuildToolDisplay and too BuildToolData. Finally, I removed the deprecated TestBox class.

Next, I updated the checks in GetDisplayWallsInUse and GenerateNewBuildDisplay to now check if the corresponding mesh pointers are valid, as well as updating both CreateXWall and CreateYWall to now set the StaticMeshcComponents to the SelectedMesh rather than the HalfWallMesh. Additionally, if the player is in erase mode and a mesh is already placed at a location, the mesh of the SMC is matched to the mesh found.

```
for (int i = 0; i < WallSegments; i++) {
 SMCPool[StartWallSegment +
 ↪  i]->SetRelativeLocation(FVector(WallStart + (WallSize *
 ↪  i), Y, 1.0f));
 SMCPool[StartWallSegment +
 ↪  i]->SetRelativeRotation(FRotator(0.0f, 0.0f, 0.0f));
 SMCPool[StartWallSegment + i]->SetStaticMesh(SelectedMesh);

 // If in default mode, check if there is a mesh already at the
 ↪  position
 if (!bInEraseMode) {
        ...
```

Following this, I worked on adding the feature to select a wall mesh from a suitable selection via the UI, starting by creating a new struct and data table to hold all the wall information. On BeginPlay, the UI gets all rows in the data table and adds a button that contains the wall data for each data table row.

```
        UCLASS(BlueprintType, Blueprintable)
class DORESPUB_API UWallButtonData : public UObject
{
        GENERATED_BODY()

public:
        UWallButtonData();
        ~UWallButtonData();

        void SetupData(class UUI_Player_Build* UI, FName
        ↪  NewName, UTexture2D* NewWallIcon);

public:
        // Pointer back to the Object UI class
        UPROPERTY(EditAnywhere, BlueprintReadWrite)
        class UUI_Player_Build* BuildUIState = nullptr;

        UPROPERTY(EditAnywhere, BlueprintReadWrite)
        FName RowName;

        UPROPERTY(EditAnywhere, BlueprintReadWrite)
        UTexture2D* WallIcon;
};
```

Finally, I implemented UpdateSelectedWall in Player-Tools which is used to update the selected mesh in the BuildToolDisplay. The buttons in the new SelectableWallList component call this function when pressed.

When using the IconTool to create images for the new walls, I found that the walls were placed too high in comparison to the camera. Additionally, they were misaligned with the center of the frame due to how they were imported, so I updated the IconTool with the ability to move the object on either the Y or Z axis.

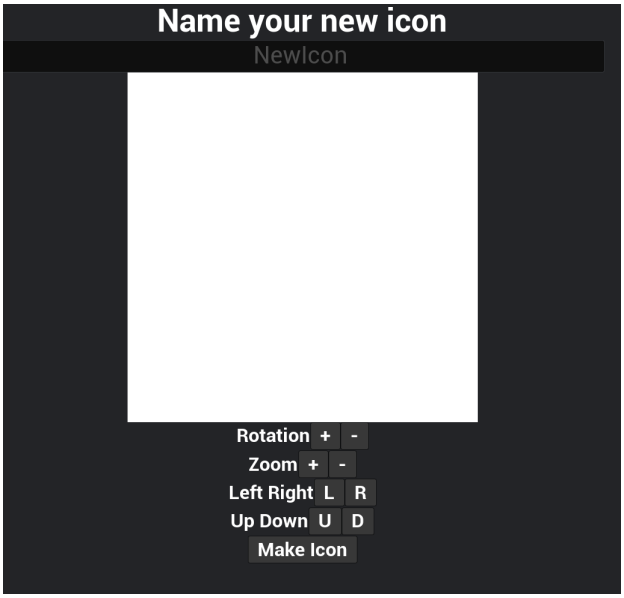

Figure 18. The updated IconTool. The player can now move the item in steps of 10uu with the new buttons

I think the improvements to the IconTool are much needed, as before it required every static mesh asset

to have the same relative origin to create similar images. The updates allow the user to modify the outcome when needed, but there are areas to improve, such as a 'last position' option as an example.

f - Refunding the player

Commit

Next was refunding the player when they remove walls they have placed already. In theory, when removing each wall from the World-BuildingLevel each wall ID needs to be placed in an array and returned to the Player-Tools, which can then be used to find a matching row in the WallDataTable. The price can then be collected from the row and used to refund the player.

I started by replacing the int Cost property in the FBuildToolData with a FName ID, which will allow the storage and eventual use of the correct ID in FBuildData. I also added a SelectedWallID property in the BuildToolDisplay, updated whenever the SelectedMesh is changed. This ID is then used in GetDisplayData alongside the original properties added to the array.

```
curr.Mesh = i->GetStaticMesh();
curr.Location = i->GetComponentLocation();
curr.Rotation = i->GetComponentRotation();
curr.ID = SelectedWallID;

// And add it to the total array
out.Add(curr);
```

I also moved the pointer to the WallDataTable from the UI-Player-Build class to the Player-Tools for better all around access. Now, the Player-Tools will no longer need to access that data from a different class when refunding and can instead directly search for matching ID's. This also means that the HalfWallCost is no longer needed as well. However, this causes the WallListView to no longer be filled on BeginPlay and remains empty, so I added the AddSelectableWallToList to the UI which is called from Player-Tools. On BeginPlay, every row of data from the WallDataTable is added as before.

```
void UUI_Player_Build::AddSelectableWallToList(FName ID,
↪  FSelectableWallData WallData)
{
        UWallButtonData* NewWallObj =
        ↪  NewObject<UWallButtonData>();
        NewWallObj->SetupData(this, ID, WallData.Icon);
        WallSelectionTileView->AddItem(NewWallObj);
}
```

The RemoveBuildingObjects function in World-BuildingLevel now returns an array of wall ID's when called, which is used in Player-Tools to find the matching ID from the data table, add it's cost to a local int, then multiply that cost by a RefundMultiplier which is then added to the player's money value.

```
TArray<FName>
↪  AWorld_BuildingLevel::RemoveBuildingObjects(TArray<struct
↪  FBuildToolData> DataToRemove)...
// Then add the ID to the output array and clear the ID in the
↪  XID
out.Add(BuildData[BuildDataIndex].XID);
BuildData[BuildDataIndex].XID = "";

// Next, remove the pointer in the BuildDataIndex and update the
↪  BuildData on the up point
PTB = BuildData[BuildDataIndex].XStaticMeshComponent;

void APlayer_Tools::SelectedToolPrimaryReleased()...
if (BTD->GetDisplayWallsInUse() > 0) {
 // Check what mode they are in
 if (bInEraseMode) {
  // Remove the Walls from the WorldBuildingLevel
  TArray<FName> wallsToRefund =
  ↪  GroundFloor->RemoveBuildingObjects(BTD->GetDisplayData());
  int total = 0;
  for (FName w : wallsToRefund) {
   total += WallDataTable->FindRow<FSelectableWallData>(w,
   ↪  "")->Price * RefundMultiplier;
  }
  PC->UpdateMoney(total);
}
```

This method of refunding the player works well, as it does not require the cost to be stored inside of the BuildData and only the ID of the matching wall. However, this could cause some issues down the line based on how many times the data table is accessed. I don't know if FindRow is quite memory intensive but I will research any optimization methods that I could implement.

g - Adding Floors

Commit

With erase mode in and refunding the player properly, I wanted to focus on one last addition in this version - adding and removing floors tiles. As floors have the same location as walls, I simply need to rework drawing walls to accept floor tiles too, add FloorData to BuildData and finally add them to the world as walls are.

I started by mini-reworking the data used by the BuildTool. Firstly, I added a sub-tool enum EBuildToolSubType, which will allow the player to swap between a floor and wall mode without creating a whole new tool. Addtionally, I added two new structs to store the data of the walls and floors in BuildData - FWallData and FFloorData. FWallData replaces the current StaticMeshComponent pointer and ID for the X and Y axis induvidually.

```
        // Enum denoting the different types of tools currently
        ↪  available to use
UENUM(BlueprintType, Category = "Tools")
enum EBuildToolSubType
{
        Wall UMETA(DisplayName = "Wall Sub-Tool"),
        Window UMETA(DisplayName = "Window Sub-Tool"),
        Door UMETA(DisplayName = "Door Sub-Tool"),
        Floor UMETA(DisplayName = "Floor Sub-Tool"),
        BuildToolSubTypeMax UMETA(Hidden),
};

// Allows the macro to be used with the EToolType Enum
ENUM_RANGE_BY_COUNT(EBuildToolSubType,
↪  EBuildToolSubType::BuildToolSubTypeMax);

USTRUCT(BlueprintType, Category = "Build Tool")
struct FWallData
{
public:
        GENERATED_BODY();

        // Default Constructor/Deconstructor
        FWallData();
        ~FWallData();

public:
        // Pointer to the StaticMeshComponent of this wall
        UPROPERTY(VisibleAnywhere, BlueprintReadOnly)
        UStaticMeshComponent* StaticMeshComponent = nullptr;

        // FName of the wall's ID
        UPROPERTY(VisibleAnywhere, BlueprintReadOnly)
        FName ID = "";

        // TArray of all materials on the wall
        UPROPERTY(VisibleAnywhere, BlueprintReadOnly)
        TArray<UMaterial*> Materials;
};
```

Next was introducing the ability to swap between sub-tools. I added two buttons to the BuildTool's UI - one for Wall and one for Floor - and added OnReleased binds which change the new CurrentSubTool when called. I also updated GetDisplayData to return the ID as 'floor' if the Floor Sub-Tool is selected.

```
void APlayer_BuildToolDisplay::UpdateSubTool
 (TEnumAsByte<EBuildToolSubType> NewSubTool)
{
        SelectedSubTool = NewSubTool;
}

TEnumAsByte<EBuildToolSubType>
↪  APlayer_BuildToolDisplay::GetSubTool()
{
        return SelectedSubTool;
}
```

To better distinguish what the function does per sub-tool, I replaced GenerateNewBuildDisplay with GenerateBuildDisplay, which instead calls the correct generate function based on the selected sub-tool - GenerateWallDisplay for Wall and GenerateFloorDisplay for floor. GenerateFloorDisplay works in a similar way to GenerateWallDisplay, but instead for the amount of Y's size calls CreateLineOnX instead of CreateLineOnX then CreateLineOnY

```
void APlayer_BuildToolDisplay::GenerateBuildDisplay(FVector
↪  StartPos, FVector EndPos)
{
switch (SelectedSubTool) {
case Wall:
 GenerateWallDisplay(StartPos, EndPos);
 break;

case Floor:
 GenerateFloorDisplay(StartPos, EndPos);
 break;

default:
 break;
 }
}
```

Finally, I renamed CreateXWall and CreateYWall to CreateLineOnX and CreateLineOnY and added a MeshOverride argument, which is used by the floor sub-tool to draw floors instead of a selected wall.

Following this, I updated Player-Tools to check for the selected sub-tool on SelectedToolPrimaryReleased - this currently ignores the price calcuation to avoid issues searching the data table for an invalid ID.

```
void APlayer_Tools::SelectedToolPrimaryReleased()...
else if (BTD->GetSubTool() == Floor) {
 // Check if there is some data in the display mode
 if (data.Num() > 0) {
  // Check what mode they are in
  if (!bInEraseMode) {
   // If they are, then add the floors to the world
        GroundFloor->AddBuildingObjects(data);
   }
   else {
        GroundFloor->RemoveBuildingObjects(data);
   }
  }
 }
```

I also updated World-BuildingLevel to use the newly updated BuildData struct, now filling the new XWall and YWall structs respectivly. It also checks

if the object being added is a floor object via the ID of 'floor', which in-turn modifies the new FloorData struct instead of a wall one.

```
void AWorld_BuildingLevel::AddNewBuildData
 (UStaticMeshComponent* SMC, FName ID)
// Check if the data we are trying to insert is a floor data
if (ID == "floor") {
 NextBuildData.FloorData.StaticMeshComponent = SMC;

 BuildData.Add(NextBuildData);
}
```

h - Reflection

I think Mk3 of the Build Tool is very strong. The player can better interact with the world than before, removing walls they have placed already easily by swapping sub-tool. The BuildToolDisplay also quickly updates to the new sub-tool with minimal issues.

Erasing is currently a little bare-bones, as it uses the same method of drawing that the normal wall tool uses. This could be improved with a collision based method, drawing a box and removing all walls that overlap it. Floors are implemented well with a large area of improvements, such as selectable display patters and materials.

The BuildTool is not complete at this stage however, as I still want the player to be able to select and add windows and doors to their building. This will be done by replacing an existing wall in the world and change it's static mesh to a different one.

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)

Riot Games. (2009) League of Legends [Game]

Ensemble Studios. (2009) Halo Wars [Game]