001

Galaxy Explorer
Development and Design Document

Daniel George Mark Dore

Brief

Contents

List of Figures

Design

1 Overview

Development

1 Base Character

Commit

I started the project by adding my usual simple base character, who features - X/Y movement, third and first person camera control and swapping between third and first person.

From this base, I added the ability to enter an interact mode, where the camera slightly zooms in and the mouse cursor appears on the screen. This stops camera rotation while still allowing player movement, which will be amended later in development where the player can rotate the camera slowly by moving the cursor to the edge of the screen.

```
void ABaseCharacter::InteractModePress()
{
bInInteractMode = true;

// Modify first person camera's FoV to the interact mode value
FirstPersonCamera->SetFieldOfView(InteractModeFocus);

// Show mouse cursor on screen
PC->bShowMouseCursor = true;
}

void ABaseCharacter::InteractModeRelease()
{
bInInteractMode = false;

// Reset focusing (if required)
FirstPersonCamera->SetFieldOfView(DefaultFocus);

// Hide mouse cursor
PC->bShowMouseCursor = false;
}
```

I also added the focus feature - by scrolling while in interact mode, the camera zooms in which is reset when exiting interact mode.

```
ABaseCharacter.ccp
FocusCamera(float AxisValue)
{
if (AxisValue != 0 && bInInteractMode)
{
if (AxisValue == 1)
{
        FocusLerp -= FocusMultiplier;
        if (FocusLerp <= 0)
        {
                FocusLerp = 0.0f;
        }

}
else if (AxisValue == -1)
{
        FocusLerp += FocusMultiplier;
        if (FocusLerp >= 1)
        {
                FocusLerp = 1.0f;
        }
}
FirstPersonCamera->SetFieldOfView(FMath::Lerp(MinFocus,
↪  InteractModeFocus, FocusLerp));
}
}
```

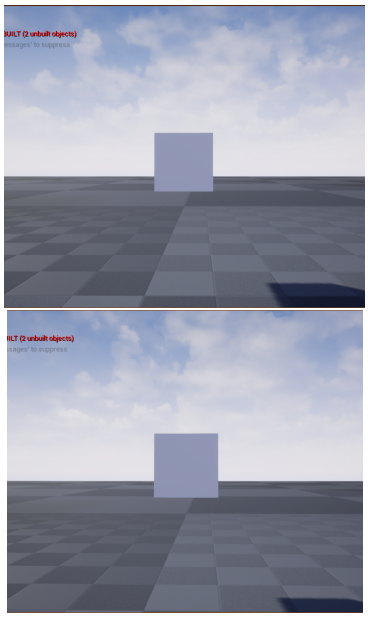

Figure 1. Example of the InteractMode - the camera zooms in and mouse controls are enabled

Finally, a quick interact feature was added. By simply tapping the interact mode instead of holding it, any object hit by the interact trace will execute its first interact feature. The time needed between the press and release can be modified by changing the 'QuickInteractTime' variable.

```
ABaseCharacter.h
// How long after pressing the player has to exit interact mode
↪  to quick interact
float QuickInteractTime = 0.2f;
```

```
ABaseCharacter.ccp
InteractModePress()
{
        GetWorld()->GetTimerManager().ClearTimer(QuickInteractHandle);

        // Start Timer for Quick Interact
        bQuickInteract = true;
        GetWorld()->GetTimerManager().SetTimer(QuickInteractHandle,
↪   FTimerDelegate::CreateUObject(this,
↪   &ABaseCharacter::QuickInteract), QuickInteractTime, false,
↪   QuickInteractTime);
}

InteractModeRelease()
{
        // Reset bQuickInteract
        bQuickInteract = false;
}

QuickInteract()
{
        // Fire a trace.  If hit an interactable, complete first
↪   event
        if (bQuickInteract) {

        }

        // Reset bQuickInteract
        bQuickInteract = false;
}
```

2 Base Interactable

Commit

In this commit, I started by slightly changed the focus variables to better suit the focus mechanic. This smooth the focus's zoom and allows the player to zoom in slighlty further.

```
//ABaseCharacter.h
float FocusMultiplier = 0.05 -> 0.08;
float MinFocus = 60.0 -> 50.0;
```

Next, I implemented the BaseInteractable class. Interactables are object that the player can interact with in the world, such as using, sitting in, powering on and more. I also added a child interactable in BasePanel. Currently, they only have a switch case on their interact function, allowing programmers to add multiple features to a single interactable which the player can choose between. These features can be easily added in the editor and bound to an event in the switch, via the map InteractionPoints.

```
//ABaseInteractable.cpp
Interact(int InteractionValue)
{
        switch (InteractionValue) {
        case 0:
                Interact_Use();
                break;

        default:
                break;
        }
```

To allow the player to interact with an interactable, the InteractWidget is added. This simply contains a ListView component, which contains a list of InteractItem buttons. The list is updated when the mouse hovers over an interactable.

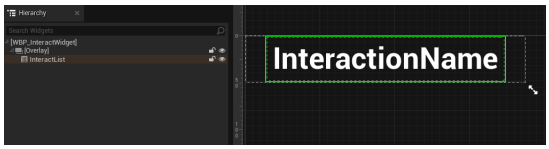

Figure 1. The InteractWidget. The list view component is updated from any interactable hit by the player's trace

This widget is added to the player as a WidgetComponent, allowing it to be placed physically in the world. I also added a SceneComponent to BaseInteractable called InteractWidgetPos, which allows the user to change where the widget will appear on a hovered interactable.

```
//ABaseInteractable.h
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
↪   "Components")
        USceneComponent* InteractionWidgetPos;

//ABaseCharacter.h
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category =
↪   "Components")
        UWidgetComponent* InteractWidgetComponent;
```

Finally, I added the Player to Widget script. I implemented a trace in the BaseCharacter's tick event, which simply fires at trace from the location of the mouse in world space, hitting any objects underneath. It then checks if a hit object is a interactable. If it is, checks to see if a soft object pointer is set already (aka an interactable has been hit before). If one is set, check if the new object hit is the same as the pointer object, and if they are different, replace the pointer to a cast of the new object. If the soft object pointer is not set,

cast to the interactable. Finally, update the visibility and location of the InteractWidgetComponent to the location of the interactable and hide component on InteractMode exit or if pointer is nullptr.

```
//ABaseCharacter::Tick()
if (bInInteractMode) {
// Get cursor location in world space
PC->DeprojectMousePositionToWorld(MouseWorldLocation,
↪  MouseWorldDirection);

// Trace for interactables under the mouse
GetWorld()->LineTraceSingleByChannel(TraceHit,
↪  MouseWorldLocation, MouseWorldLocation +
(MouseWorldDirection * PlacementDistance),
ECC_WorldStatic, FCollisionQueryParams(FName("DistTrace"),
↪  true));

// First, check if the trace hit anything
if (TraceHit.Actor != nullptr) {
if (TraceHit.Actor->IsA(ABaseInteractable::StaticClass())) {
        if (LastInteractedObject == nullptr) {
                // Cast to the object
                LastInteractedObject =
↪  Cast<ABaseInteractable>(TraceHit.Actor);
        }
        else {
                // Check if the LastInteractedObject and
↪  TraceHit.Actor are the same object
                if (LastInteractedObject->GetName() ==
↪  TraceHit.Actor->GetName()) {
                        // Don't cast again
                }
                else {
                        // Replace the pointer in
↪  LastInteractedObject with the new TraceHit->Actor
                        LastInteractedObject =
↪  Cast<ABaseInteractable>(TraceHit.Actor);
                }
        }
        // Update the location of the interactable widget and
↪  make it visible
        InteractWidgetComponent->SetWorldLocation(LastInteracted
        Object->InteractionWidgetPos->GetComponentLocation());
        InteractWidgetComponent->SetWorldRotation(LastInteracted
       Object->InteractionWidgetPos->GetComponentRotation());
        InteractWidgetComponent->SetVisibility(true, true);

}
else {
        // Set last hit object to nullptr and hide the
↪  interactable widget
        LastInteractedObject = nullptr;
        InteractWidgetComponent->SetVisibility(false, false);
}
}
```

3 Panel Interactable

Commit

In this commit, I started by adding the ability to interact with widgets in the world via a WidgetInteractionComponent, and also implemented two functions - InteractModePrimaryPress and InteractModePrimaryRelease.

```
//ABaseCharacter.cpp
InteractModePrimaryPress()
{
        WidgetInteractionComponent->PressPointerKey(EKeys::LeftMouseButton);
}

InteractModePrimaryRelease()
{
        WidgetInteractionComponent->ReleasePointerKey(EKeys::LeftMouseButton);
}
```

I also added a simple jump to the player.

```
//ABaseCharacter.cpp
JumpPress
{
        ACharacter::Jump();
}

JumpRelease
{
        ACharacter::StopJumping();
}
```

Next, I slightly modified the core of the interactables. Firstly, the WidgetInteractionPoint is now attached to the root of the object. I also added a InteractionPoints map for when an item is turned off, if item is, UpdateInteractionList returns this map instead. Finally, the boolean bPowerOn is added, which denotes if an object is turned on or off.

```
//ABaseInteractable.h
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
↪  "Interaction")
        TMap<int, FString> InterationPointsPowerOff;

// Bool to denote if the interactable is currently in an on or
↪  off state
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Power")
        bool bPowerOn = true;

// Bool to denote if the interactable's power can be toggled
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Power")
        bool bPowerToggle = false;
```

This was tested and implemented in the ShipManagerPanel and PanelButton interactables. Interacting with the button turns the panel to either an on state or off state.

Figure 3. Example of the on and off states - using the button toggles the panel on and off

Finally, I added the ability to attach the player to an interactable, through the function AttachToInteractable. While attached, the player's camera will rotate to face the interactable on tick until they either exit interation mode or move to far away from the attached object.

```
//ABaseCharacter.cpp
//Tick()
// Check if locked to an object
if (InteractableLockedTo) {
        // If so, update the controllers rotation to look at
↪  said interactable
        PC->SetControlRotation((GetActorLocation() -
    InteractableLockedTo->GetActorLocation()).Rotation() +
    FRotator(0.0f, 180.0f, 0.0f));

        // Then check if the player has moved out of range.  If
↪  so, dettach.
        if (FVector::Dist(GetActorLocation(),
↪  InteractableLockedTo->GetActorLocation()) >
↪  DettachDistance) {
                InteractModeRelease();
        }
}
else...
```

Some modifications were added to the interact mode, such as bIgnoreNextRelease, as the player is already in interact mode when attaching. This allows the player to release the interact mode key after they attach, rather than holding it down while using the object.

4 Ship Manager UI

Commit

In this commit, I started by creating the simple blockout of the ShipManagerUI. Currently, it only contains a widget switcher - which is used to swap between the UI's different states (off, idle, in use) - and a list view component - that displays all of the ships in the players inventory. Currently, the list isn't implemented but will be implemented next.

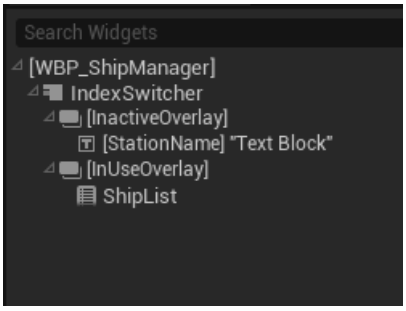

Figure 4. The ShipManagerUI hierarchy

Next, I added a ShipInventoryComponent ActorComponent. This will handle the storing and collecting of the data of the player's owned ships.

To implement the ability to return the UI to it's idle state after unlocking, I modified the players unlock function to call the Interact-OnUnlock function in the attached interactable pointer if attached to one.

```
//ABaseInteractable.cpp
Interact_OnUnlock(ABaseCharacter* Interactee)
{
}

//ABaseCharacter.cpp
//InteractModeReleased()
if (InteractableLockedTo) {
        Cast<ABaseInteractable>(InteractableLockedTo)->
    Interact_OnUnlock(this);
}
```

Finally, I added the Ship Manager object, which consists of two child actors - the Panel Button and Ship Manager Panel interactables. From this, I added the ShipManagerUI that I created earlier to the Ship Manager Panel, along with a function to update the object.

```
AShipManagerPanel
BeginPlay() {
// Get reference to the interact widget class
ManagerWidget =
↪  Cast<UShipManagerUI>(ManagerWidgetComponent->GetWidget());
}

UpdateManagerWidget(int Index)
{
ManagerWidget->UpdateManagerSwitcher(Index);
}
```

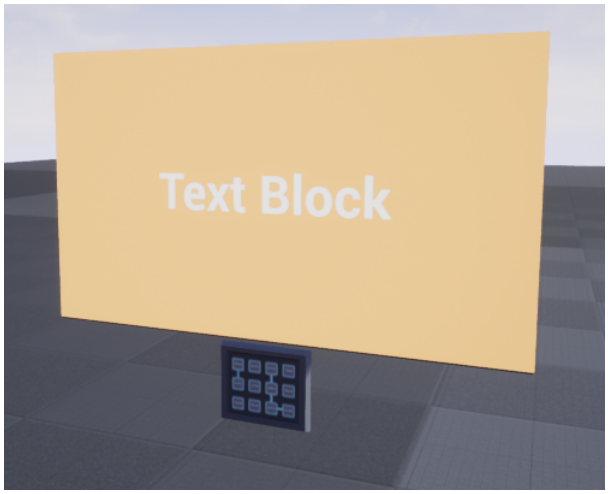

Figure 5. The ship manager object - the button below can be used to turn the panel on and off

5 Fleet Manager

Commit

In this commit I created the design and scripted the fleet manager, starting by creating a few data objects to handle some of the ship stats. Usually I create a data library to store all of the structs and enums, but this can cause more memory to be used than necessary. For example, if the data library holds 3 structs but a class only need to use one, that class still needs to load all three using more memory. Instead, I created the Data folder and placed all of the structs and enums in separate classes.

```
USTRUCT(BlueprintType)
struct GALAXYEXPLORER_API FShipData : public FTableRowBase
{
public:
GENERATED_BODY();

UPROPERTY(EditDefaultsOnly, BlueprintReadOnly)
        FString Name;

UPROPERTY(EditDefaultsOnly, BlueprintReadOnly)
        TEnumAsByte<EShipManufacturer> Manufacturer;;

UPROPERTY(EditDefaultsOnly, BlueprintReadOnly)
        TSubclassOf<class ABaseShip> Class;

UPROPERTY(EditAnywhere, BlueprintReadOnly)
        FString Status; // Replace with ENUM

UPROPERTY(EditAnywhere, BlueprintReadOnly)
        FString Location;

UPROPERTY(EditAnywhere, BlueprintReadOnly)
        int Cargo;

UPROPERTY(EditDefaultsOnly, BlueprintReadOnly)
        int CargoMax;

UPROPERTY(EditDefaultsOnly, BlueprintReadOnly)
        int Crew;

FShipData();
FShipData(FString InName, TEnumAsByte<EShipManufacturer>
↪  InManufacturer,
 TSubclassOf<class ABaseShip> InClass, FString InStatus,
 FString InLocation, int InCargo, int InCargoMax, int InCrew);
~FShipData();
};
```

The FShipData struct is used to hold the ship information inside the players ShipInventoryComponent, as well as displaying in the FleetManagerUI. The EManufacturer enum is used to designate the ship' manufacturer. Next, I implemented two functions into the ShipInventoryComponent - GetPlayerShipList and AddShipToList. One returns the ship list array while the other adds to the array.

```
//UShipInventoryComponent.cpp
TArray<FShipData> GetPlayerShipList()
{
        return PlayerShipList;
}

AddShipToList(FShipData NewShip)
{
        PlayerShipList.Add(NewShip);
}
```

UpdateFleetList is next, which I implemented through a BlueprintImplementableEvent in the UI class. It is called when a player attaches to the panel, which gets the player's ship list via GetPlayerShipList. For each entry in the array, a new FleetManagerObjectData class is constructed which is then added to the List View component. Finally, the data is displayed as individual FleetManagerObjects in the list.

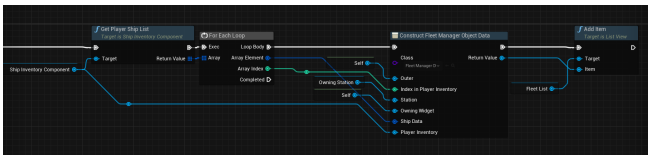

Figure 6. The UpdateFleetList function. I created this in BP as the interface used by the list object cannot be used in c++

Figure 7. Example of the default FleetManagerObject

Finally, I updated the design of the FleetManager widget to look more like the one from Star Citizen.

Figure 8. Comparison between the current widget's design and Star Citizen's version

6 Station Managers

Commit

In this commit, I added the ability to create stations.

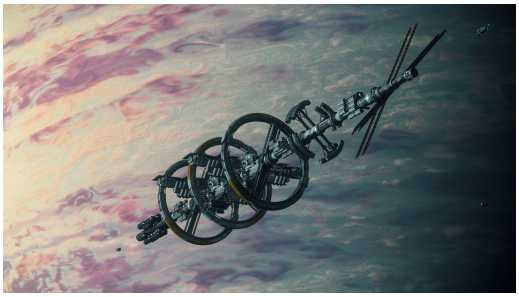

Figure 9. Port Olisar is a station in Star Citizen which orbits Crusader. Players can land, store and retrieve their ships from the Fleet Managers available.

In Star Citizen, stations are large social areas where players can navigate to, manager their ships, collect cargo and more. In my project they are controlled by a single StationManager class that currently stores pointers to all FleetManagers and SpawnLocations of a single station, which can then be used to spawn ships from the players inventory.

Firstly, I had to convert the existing FleetManager from a Blueprint class to it's own C++ class, to allow the StationManager class to access it. I also created a new class for the ShipSpawnLocations. Both of theses classes have a pointer to a StationManager, linking the two objects together.

Next, I updated the FleetManagerObjectData and FleetManagerObject classes with a pointer to the StationManager that is in use, allowing for the FleetManagerObject to call functions on button press. Finally, on BeginPlay StationManagers now directly update a FleetManagerUI pointer, rather than using a function on the FleetManager that then updates the UI.

```
//AStationManager::BeginPlay()
// Update all SpawnLocations station pointers to this
for (int i = 0; i < SpawnLocations.Num(); i++) {
        SpawnLocations[i]->Station = this;
}

// Update all FleetManagers station pointers to this
for (int i = 0; i < FleetManagers.Num(); i++) {
        FleetManagers[i]->Station = this;
        FleetManagers[i]->Panel->ManagerWidget->UpdateStationDetails(StationNam
↪  this);

}
```

7 Interact Mode Scroll

Commit

In this commit, I attempted to implement the ability to scroll through interaction points in Interact mode, but this was not successful. Scroll inputs are taken successfully, but Unreal Engine's WidgetInteractionComponent doesn't seem to translate these inputs into scrolling on ListView components.

I will rework this later in time with a simpler system, where scrolling up or down will clear and remake the list with three entries - the current choice and choices in both directions (when available).

Figure 10. How the rework of the InteractWidget will look like.

8 Small Interactable Rework

Commit

In this commit, I reworked the interactables to use two booleans to determine their state - Enabled and RecievingPower. This is mainly to fix the problem which would be introduced with ship components - turning the ship on will provide power to all components, but will only activate the enabled ones.

9 Spawning Ships

Commit

In this commit, I added the ability to spawn ships from the players inventory. However, this required a little modification to the ship's data struct in the addition of the ShipClassification enum. Spawn Locations also use this enum, to only allow ships on or below its classification to spawn on the pad, and ignore any above.

```
UENUM(BlueprintType, Category = "Ship")
enum EShipClassification
{
	Small,
	Medium,
	Large,
	Capital
};
```

Additionally, I added bIgnorePower to BaseInteractable, as some interactables such as buttons should be treated as always receiving power.

To make sure ships don't spawn at spawn locations that already have a ship assigned to them or in their collision, I added a overlap box component. Every time an overlap occurs - be it player or ship - an int is increment in StationSpawnLocation, which is decremented once the overlap ends. If this int is larger than 0, this spawn location is ignored when spawning a new ship.

```
void
↪  AStationSpawnLocation::OnSCBeginOverlap(UPrimitiveComponent*
OverlappedComp, AActor* OtherActor, UPrimitiveComponent*
↪  OtherComp,
int32 OtherBodyIndex, bool bFromSweep, const FHitResult&
↪  SweepResult)
{
	Overlaps++;
}

void AStationSpawnLocation::OnSCEndOverlap(UPrimitiveComponent*
↪  OverlappedComp,
AActor* OtherActor, UPrimitiveComponent* OtherComp, int32
↪  OtherBodyIndex)
{
	Overlaps--;
}
```

I also added the bool bInUse to StationSpawnLocation, to state when a pad should be ignored even if no collisions are currently occurring - such as on ship take off or land. Both of these conditions are check through the function PadNotInUse

```
bool AStationSpawnLocation::PadNotInUse()
{
	if (Overlaps == 0 && !bInUse) {
		return true;
	}
	return false;
}
```

Finally, I implemented the SpawnShipAtLocation function, which spawns the ship in the middle of the collision box.

```
bool AStationSpawnLocation::SpawnShipAtLocation(FShipData
↪  InShipData)
{
// Spawn ship
FVector wL = SpawnCollision->GetComponentLocation();
ABaseShip* NewShip =
↪  GetWorld()->SpawnActor<ABaseShip>(InShipData.Class,
FVector(wL.X, wL.Y, wL.Z -
↪  (SpawnCollision->GetScaledBoxExtent().Z)),
FRotator());
bInUse = true;
return true;
}
```

Before spawning however, the FleetManager searches through all StationSpawnLocations of a station to find one which is big enough and not in use, through the GetSuitableSpawnLocation function.

```
AStationSpawnLocation*
↪  AStationManager::GetSuitableSpawnLocation(TEnumAsByte<EShipClassi
-fication> InClass)
{
for (int i = 0; i < SpawnLocations.Num(); i++) {
        if (SpawnLocations[i]->ReturnPadData().GetValue() <=
    InClass.GetValue() && SpawnLocations[i]->PadNotInUse()) {
                return SpawnLocations[i];
        }
}
return nullptr;
}
```

The last thing I added in this commit were the start of the ship-specific classes, which includes - BaseShip-Interactables, which are interactables specifically used by ships and have one function linking a pointer between it and the ship, the VTOLGimbal SceneComponent and the LandingGear SceneComponent. These will be the next target to get implemented.

10 Ship Components

Commit

In this commit, I implemented the VTOL Gimbals, added ship control functions and improved the base ship class. Firstly, I started by modifying the SpawnShipAtLocation (StationSpawnLocation) and SpawnShip (StationManager) functions to include a pointer to a ShipInventory class. This is due to requiring it later on in development when linking the spawned ship to the player and vice-versa.

```
bool AStationSpawnLocation::SpawnShipAtLocation(FShipData
↪  InShipData,
int Index, UShipInventoryComponent* OwningInvent)

FString AStationManager::SpawnShip(FShipData InShipData,
int Index, UShipInventoryComponent* OwningInvent)
```

Next, I implemented the VTOLGimbal which was added last commit. These scene components have one role - rotating attached components between an active and deactive state via a timeline. The designer can also modify the speed of the rotation by changing the VTOLRotationSpeed float.

```
UVTOLGimbal::BeginPlay()
if (TimelineCurve) {
        FOnTimelineFloat TimelineProgress;
        TimelineProgress.BindUFunction(this,
↪  FName("RotationTimelineProgress"));
        RotationTimeline.AddInterpFloat(TimelineCurve,
↪  TimelineProgress);
        RotationTimeline.SetLooping(false);
        RotationTimeline.SetPlayRate(1 / VTOL_RotationSpeed);
}
```

Timelines in C++ are setup in a different way to Blueprint ones, as they require the implementation of the TickTimeline function on the owning classes Tick. Additionally, they require a FCurveFloat object rather than setting up one internally.

```
void UVTOLGimbal::TickComponent(float DeltaTime
, ELevelTick TickType, FActorComponentTickFunction*
↪  ThisTickFunction)
{
Super::TickComponent(DeltaTime, TickType, ThisTickFunction);

RotationTimeline.TickTimeline(DeltaTime);
}

UVTOLGimbal::UVTOLGimbal()
ConstructorHelpers::FObjectFinder<UCurveFloat>CurveObj
(TEXT("/Game/Ship/Data/VTOLRotationCurve"));
        if (CurveObj.Succeeded()) { TimelineCurve =
↪  CurveObj.Object; }
```

Next, I improved the storing of all the ship's components by sorting them into different arrays, along side the addition of the InteriorShipLights and ExteriorShipLights arrays - which store pointers to the lights inside and outside the ship respectively. I also implemented some control functions and active booleans, these being:

- ToggleExteriorLights function, used to toggle the exterior lights
- ToggleInteriorLights function, like above but interior
- ToggleVTOLMode function, to change the state of VTOL gimbals
- bLandingGearDown, bExLightsOn, bInLightsOn, bInVTOLMode bools, to denote the state of the ship

```
void ABaseShip::ToggleExteriorLights()
{
bExLightsOn = !bExLightsOn;
for (int i = 0; i < ExteriorLights.Num(); i++) {
        ExteriorLights[i]->SetVisibility(bExLightsOn);
}
}

void ABaseShip::ToggleInteriorLights()
{
bInLightsOn = !bInLightsOn;
```

```
for (int i = 0; i < InteriorLights.Num(); i++) {
        InteriorLights[i]->SetVisibility(bInLightsOn);
}
}

void ABaseShip::ToggleVTOLMode()
{
bInVTOLMode = !bInVTOLMode;
for (int i = 0; i < VTOLGimbals.Num(); i++) {
        VTOLGimbals[i]->ToggleVTOLMode(bInVTOLMode);
}
}
```

Finally, I override the Interact function in the BaseShipInteractable class to call these new ship functions. From this class, I created a child BP class of BPShipButton, and added this button to the ship.

```
void ABaseShipInteractable::Interact(int InteractionValue,
↪  ABaseCharacter* Interactee)
{
switch (InteractionValue) {
case 0:
        OwningShip->ToggleExteriorLights();

        // Then update the Interactees interaction widget
        Interactee->UpdateInteractWidget();
        break;

case 1:
        // Toggle state of ship landing gear
        OwningShip->ToggleInteriorLights();

        // Then update the Interactees interaction widget
        Interactee->UpdateInteractWidget();
        break;

case 2:
        // Toggle state of ship VTOL mode
        OwningShip->ToggleVTOLMode();

        // Then update the Interactees interaction widget
        Interactee->UpdateInteractWidget();
        break;

default:
        break;
}
```

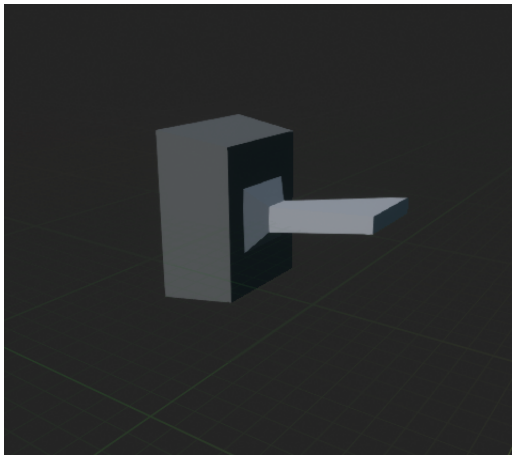

Figure 11. The ShipButton Blueprint class. Currently, it can only use this static mesh, but will be updated later to allow for different meshes

11 Landing Gear and Ship Rebuild

Commit

In this commit, I implemented the landing gear scene component and rebuilt the ship. The LandingGear is similar to the VTOL gimbal, but also features a location vector that can be used to move between locations.

```
void ULandingGear::LandingGearTimelineProgress(float Value)
{
SetRelativeLocation(FMath::Lerp(Gear_Disabled.GetLocation(),
Gear_Enabled.GetLocation(), Value));

SetWorldRotation(FMath::Lerp(Gear_Disabled.GetRotation(),
Gear_Enabled.GetRotation(), Value));
}
```

Due to an error occurring when compiling the project while UE4 was still open, any VTOL Gimbal or ShipInteractable in the ship would become slightly corrupted - they would still exist in the BaseShip class, but couldn't be used and when deleted caused the whole editor to crash. To remedy this, I deleted the ship class and rebuilt it again.

12 Ship Doors

Commit

In this commit, I implemented ShipMoveables, updated Interactables with an OnCasted function and added door buttons to the ship.

I started by implementing the ShipMoveable SceneComponent. ShipMoveables are close to identical to the LandingGear component added in the previous commit - also featuring a timeline and two transforms that are lerped between - but utilize Unreal's GameplayTags system to sort different moveables into separate pools.

Figure 12. Example of the Ship's ShipMoveables being sorted into separate map entries base on their gameplay tags

These pools are created on ship spawn, stored in a FMoveablesList struct - as you cannot nest a TArray in a map, but you can have a TArray in a struct in a map. Each Moveable can be contained in multiple map entries. For example, LeftDoor has 'Doors' and 'LeftDoor' gameplay tags.

```
// Get all ShipMoveables and sort them in the Moveables map
↪  based on their tags
TArray<UActorComponent*> foundMoveables;
foundMoveables =
↪  GetComponentsByClass(UShipMoveable::StaticClass());
for (int i = 0; i < foundMoveables.Num(); i++) {
        TArray<FName> tags = foundMoveables[i]->ComponentTags;
        for (int j = 0; j < tags.Num(); j++) {
        // Check if index in map exists.  If no, add a new
↪  index
                if (Moveables.Contains(tags[j])) {
                        Moveables[tags[j]].Components.
   Add(Cast<UShipMoveable>(foundMoveables[i]));
                }
                else {
                        Moveables.Add(tags[j], FMoveablesList
   (Cast<UShipMoveable>(foundMoveables[i])));
                }
        }
}
```

These moveables can be moved by calling the ToggleMoveables function on the ship, which plays or reverses the timeline of all moveables with a corresponding tag.

```
//ABaseShipInteractable::Interact...
case 4:
// Toggle state of a tag name
OwningShip->ToggleMoveables(InteractableTags);
if (OwningShip->doorsOpen != 0) {
        bEnabled = true;
}
else {
        bEnabled = false;
}

// Then update the Interactees interaction widget
Interactee->UpdateInteractWidget();
break;

//ABaseShip::ToggleMoveables(FName TagName)
{
        FMoveablesList* moveablesToModify =
↪  Moveables.Find(TagName);
        for (int i = 0; i <
↪  moveablesToModify->Components.Num(); i++) {
                doorsOpen +=
↪  moveablesToModify->Components[i]->ToggleMoveable(1);
        }
}
```

Next was implementing the door buttons. Two buttons are added for each door - one internal and one external - which call the ToggleMoveable function based on the tag given to the button. An additional button is added to change the state of all doors of the ship. If no doors are open, the button opens all the doors. If one or more doors are open however, all open doors should be closed.

Which way all doors should be moved is tracked by counting how many doors of the ship are currently open. Every time a moveable is toggled, it is checked if it is a door. If it is, the doorsOpen int is incremented if the moveable is played, else it is decremented if it is reversed.

```
int UShipMoveable::ToggleMoveable(int State)
{
if (State == 1) {
bMoveableActive = !bMoveableActive;
if (bMoveableActive) {
        MoveableTimeline.Play();
        return 1;
}
else {
        MoveableTimeline.Reverse();
        return -1;
}
}
else if (State == 2) {
        bMoveableActive = false;
        MoveableTimeline.Reverse();
        return -1;

}
else if (State == 3) {
        bMoveableActive = true;
        MoveableTimeline.Play();
        return 1;
}
```

When the CloseAllDoors function is called, it checks if doorsOpen is greater than 0. If it is, it closes

all doors and resets the int to 0 - as even doors already closed will 'close', decrementing the int past 0. Else, it opens all doors.

```
void ABaseShip::CloseAllDoors()
{
FMoveablesList* moveablesToModify = Moveables.Find("Doors");
int stateToSet = 3;
if (doorsOpen != 0) {
        stateToSet = 2;
}
for (int i = 0; i < moveablesToModify->Components.Num(); i++) {
        doorsOpen +=
↪  moveablesToModify->Components[i]->ToggleMoveable(stateToSet);
}
if (doorsOpen < 0) {
        doorsOpen = 0;
}
}
```

Finally, to update this button to it's correct interact state, the OnCasted function is added to all interactables. This is called whenever the interactable is first casted to in interact mode.

```
//ABaseCharacter;;Tick...
if (LastInteractedObject == nullptr) {
        // Cast to the object
        LastInteractedObject = Cast<ABaseInteractable>
    (TraceHit.Actor);
        LastInteractedObject->OnCasted();
        UpdateInteractWidget();
}
```

An example of the door buttons is available below.

Video

Usage

Bibliography

1 Video References