

001

---

**Galaxy Explorer  
Development and Design Document**

---

Daniel George Mark Dore

# BRIEF

---

## CONTENTS

---

<b>Design</b>	<b>3</b>
1 Overview	3
<b>Development</b>	<b>3</b>
1 Base Character	3
2 Base Interactable	4
3 Panel Interactable	5
4 Ship Manager UI	6
5 Fleet Manager	7
6 Station Managers	8
7 Interact Mode Scroll	8
8 Small Interactable Rework	9
9 Spawning Ships	9
<b>Usage</b>	<b>9</b>
<b>Bibliography</b>	<b>10</b>
1 Video References	10

## LIST OF FIGURES

---

1 Interact Mode . . . . .	3
2 The InteractWidget . . . . .	4
3 On and Off States . . . . .	5
4 The ShipManagerUI hierarchy . . . . .	6
5 The ship manager object . . . . .	7
6 UpdateFleetList function . . . . .	7
7 A default FleetManagerObject . . . . .	8
8 The current widget and SC's version . . . . .	8
9 Port Olisar in Star Citizen . . . . .	8
10 Interact Mode design . . . . .	9

# 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.cpp
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));
}
```

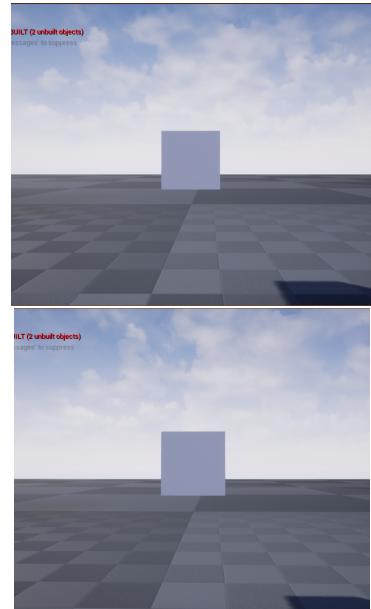


Figure 1. Example of the InteractMode - the camera zooms in and mouse controls are enabled

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.cpp
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 slightly further.

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

Next, I implemented the BaseInteractable class. Interactables are objects 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 InteractionWidgetPos, which allows the user to change where the widget will appear on a hovered interactable.

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

//ABaseInteractable.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.

```
//ACharacter::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.

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

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

I also added a simple jump to the player.

```
//ACharacter.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> InteractionPointsPowerOff;

// 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 interaction mode or move too 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, detach.
    if (FVector::Dist(GetActorLocation(),
        InteractableLockedTo->GetActorLocation()) >
        DetachDistance) {
        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.

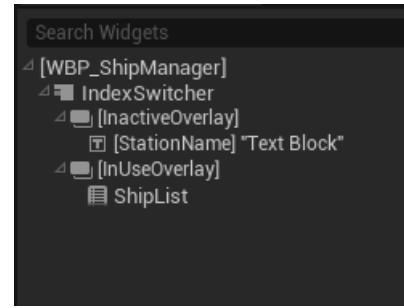


Figure 4. The ShipManagerUI hierarchy

Next, I added a ShipInventoryComponent Actor-Component. 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 its 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);
}
```

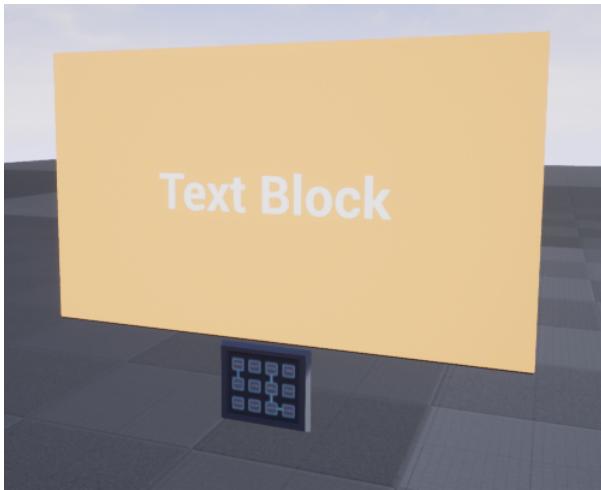


Figure 5. The ship manager object - the button below can be used to turn the panel on and off

## 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's 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 FleetManagerObject class is constructed which is then added to the List View component. Finally, the data is displayed as individual FleetManagerObjects in the list.

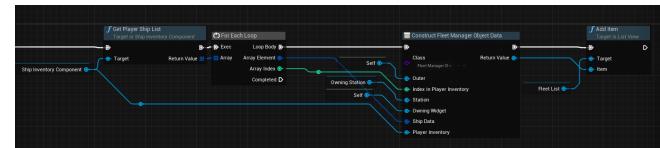


Figure 6. The UpdateFleetList function. I created this in BP as the interface used by the list object cannot be used in c++



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.

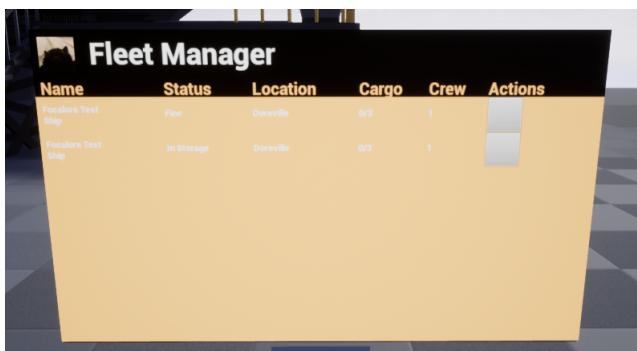


Figure 8. Comparison between the current widget's design and Star Citizen's version



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, manage 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 these 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(StationName
    & this);
}
```

## 6 Station Managers

[Commit](#)

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

## 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](#)

## Usage

## Bibliography

### 1 Video References