**Grabber Component:**

Grabber:

Create a default pawn BP

(play, eject, select, blueprint menu, convert selected actor to blueprint)

Create Game Mode BP based on c++ class

(set BP Game Mode to default, with pawn BP)

Create C++ class Actor Component - grabber

Add Component to BP\_Pawn, custom class, add grabber

// Pseudo code back in Grabber .cpp //

// In Tick - get player view point //

declare FVector, FRotator,

pass in these to PlayerController::GetPlayerViewPoint()

#define OUT right below #includes

// DrawDebug - simulate Ray Cast //

Declare float Reach, FVector LTStart, LTDirection, LTEnd

LineTraceStart = PlayerViewLocation

LineTraceDirection = PlayviewRotation.vector();

LineTraceEnd = LTStart + (LTDirection \* Reach);

DrawDebugLine(GetWorld(), LTStart, LTEnd, FColor::Red, false, 0.0f, 0, 10.0f)

// last 4 params - persist line bool, time persist f, depth priority int, thickness f

\* Set Simulate physics on objects you want to be grab, set collision to physics, and overlap true

// LineTracebychannel will work but will ignore physics body filter

// FCollisionObjectQueryParams - Object Types Query interested (Object Trace Channel)

// FCollisionQueryParams - TraceParams(FName(TEXT("")), false, GetOwner())

// TraceParams (traceTag not hitTag, collisiontype(complex or simple), actor ignore)

// Use 5th override for ObjectQueryParams and 2nd override for QueryParams

// FCOQP vs FCQP – FCOQP looks for object trace channel (object type based on collision preset) for which the trace will hit and return actor that belongs to that object type and

Query Params is a struct for holding parameters defining collision functions (simple vs complex)

Declare FHitResult HitResult, FCollisionQueryParam TraceParams

GetWorld()->LineTraceSingbleByObjectType(OUT Hit Result, LTStart, LTEnd,

FCollisionObjectQueryParam(ECollisionChannel::ECC\_PhysicsBody), QueryParams)

AActor\* ActorHIt = Hit.GetActor();

Create a function in grabber that takes this Trace out of Tick, and then calls it.

Physics Handle part of Grabber Class:

Find Components by Class Grabber: <Need Physics Handle>

Add Component to pawn, Physics Handle

In grabber.h -

UPhysicsHandleComponent\* PhysicsHandle = nullptr;

UInputComponent\* InputComponent = nullptr;

// In BeginPlay look for attached physics handle

PhysicsHandle = GetOwner()->FindComponentByClass<UPhysicsHandleComp>();

InputComponent = GetOwner()->FindComponentByClass<UInputComponent>();

if (!PhysicsHandle) {return;}

if (!InputComponent){return;}

// Go to Input Binding, Project Settings, Action Map "Grab" to right click

InputComponent->BindAction("Grab, IE\_Pressed, this, <point to function>)

InputComponent->BindAction("Grab, IE\_Released, this, <point to function>)

// Create Grab and Release functions - void Grab(); void Release(); //

// Release () {PhysicsHandle->ReleaseComponent();}

**Blueprint Communication**

Macros (Inline Function)

* Macros are the opposite of this document’s purpose, which is discuss the many ways to communicate between blueprints (BPs) or classes. Macros are limited to the current blueprint for which it resides or is defined in.
* A Macro is exactly like creating a function but the return value of the Macro will never be used outside of the current class (or blueprint).
* *Macro vs Function:* 
  + A function will at some point produce a return value that will be useful outside of its current class (eg. Like a set health function on a player character, because other classes will want to manipulate the health value of a player).
  + A Macro is a function that only ever manipulates data on the current class (or blueprint).
  + Can be used with latent functions (delays), wildcard variables, and multiple execution pins (versus a function which cannot use any of them).
  + Custom event, function calls, timelines, and add component nodes cannot be used
  + A function only compiles once whereas a Macro compiles on every usage
* *Code Reuse / Encapsulation:* A Macro can be re-used multiple times within a class/BP and uses a single node to represent all the logical nodes that were defined when the nodes were collapsed (makes things neat and tidy)
* *Code Reuse / Encapsulation:* Prevents logic errors and overwriting code by being able to change one thing within a macro that would apply to all the uses of that macro.

**Casting (1-1 Blueprint Communication)**

* One Blueprint talks to another Blueprint via a cast node.
* Sender gets a reference to the receiver (usually on overlaps, ray casts, or get all actors of class).
* Once you get the receiver’s reference, cast to that reference’s class blueprint (via the cast node) to get access to that receiving blueprint’s functions and values. (drag off ref., type cast to <BP>).

**Blueprint Interfaces**

* Sender needs a reference of the receiver to send the message (the reference drives who it is you are communicating to).
* Interface sends out messages (through any asset, once Interface is created) to listening objects
* *When to create a BP Interface*: The object sender doesn’t care what happens to the message once the receiver retrieves the signal (if the receiver implements the interface, then great, but if it doesn’t, it doesn’t matter because the sender’s job is over at that point).
* *How to create a BP Interface*: Content Browser, right click, Blueprints submenu, BP Interface
* *How to implement an interface on receiving class*: Receiving class of message needs to implements the interface in the Class Settings, add interface (Be sure to compile immediately after adding the interface to get access to it in receiving BP).
* *How to use BP Interface on receiving class*: Once you have implemented the interface in class settings, you can just create an event by using the same name of the function defined inside of the BP Interface.

**Event Dispatchers**

* (Listener) – think Twitter
* Communication between one sending BP/class to many receiving BPs/classes (typically many receivers, but none are required).
* Dispatcher allows you to hook up the logic from the receiver, "listen for this event from that sender”.
* Unlike a BP Interface, the sender doesn’t bother with a reference of the receiver; it’s just going to make a call to whomever is listening
* On the other hand, the receiver needs a reference from the sender.
* Once the receiving BP receives that reference (via a cast, or whatever), it assigns or binds a custom event, which then performs some logic.
* Even though the receiver is doing the bulk of the work, the sender is the one making the call, so the sending BP/class is the place where you create the Event Dispatcher (directly below variables section of the blueprint panel inside of a blueprint).
* One source - all object executes
* The source does not need to know all targets, only targets need to know, which means you can dynamically spawn and destroy targets and retain control over them.
* A common example would be to create an Event Dispatcher on your player character blueprint for an interaction key press. Create a key press event (like ‘E’), and all you need to do is drag your dispatcher onto the graph, drop it, and set it as a ‘call’. Then, for each other BP/class that you want to use a single key press (like ‘E’) to perform some specific logic (like toggle light, open door), all you need to do is cast to your player character on event begin play. From there, drag off your player character cast node’s return actor pin and type the name of dispatcher you just created. Select Assign <name of your dispatcher>. Then name your custom event, and create some interact functionality for that specific BP/class.



***When to use each form of BP communication****:*

* When you need one to one communication, just cast to and call function
* When one actor needs communicate to many actors - use event dispatchers
* When you need get actors of multiple classes into list (and then call their function) - use blueprint interface (it has a bit more useful functionality than that.
* When you want to communicate to multiple objects with same class - get all actors of class, cast to and call function in loop (but interfaces are more elegant and flexible here)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Functions | Events | Macros | Collapsed Graphs |
| Execution Paths | One | One | Any | Any |
| Non-exec Outputs | Any | None | Any | Any |
| Latent actions | No | Yes | Yes | Yes\* |
| Add components/timelines | No | Yes | No | Yes\* |
| Other events | No | N/A | No | Yes\* |

\* *A collapsed graph inherits the limits of the graph it is in, e.g., a collapsed graph inside of a function still cannot contain latent actions or timelines.*

**Unreal Tips:**

Show Collision in viewport -

Top of editor, select lit for dropdown, select Collision, select Player collision for simple collision, and visibility collision for complex colliders (mesh colliders).

Line Trace can determine whether trace hit should use simple collision or visibility collision (complex collider)

Landscape

* Landscape resolution adjusted by using larger scale. 1000 scale. 15x15 quads. 7x7 Components
* User Layer blend node in material editor to create a multi-purpose material for landscaping
  + Don’t’ forget to click on main material , go to usages, check used with landscapes
* Once material complete, left click on landscape in World outliner, add the material to landscape
* Go back to landscape mode, select paint for your brush, under target layers, click plus for each layer.
* For low poly, use World Position node to DDX and DDY. Cross product to normal.

Add Texture to Terrain

* Go into material with layer blend set up
* Instead of vector parameter for a flat color, add two texture sample nodes
* Add Landscape Layer Coords and feed them into those two texture sample nodes

Export your own heightmap

* Landscape -> Scope -> Right click on height map under targeted layer, export to file

Import height map for creating a new landscape

* Landscape (with no landscape already in there) - > Manage -> Import from file

Change Map from high to low

* Landscape -> Manage -> SelectionTool -> ChangeComponentSize, lower quad size, and resample

For resizing images automatically in a Blueprint Widget

* Add a scale box to the canvas panel
* Make the image panel a child of the scale box
* Set Scale Box Anchor to Fill Entire Canvas, and set all offsets to 0, 0, 0, 0.
* Then under Stretching of the details panel, use dropdown of stretch, and set it to Scale to Fill.

Adding Custom Fonts (be sure to read the copy write license):

* Dafont.com, download font, unzip.
* Import ttf into a folder in unreal by selecting the import button top of content browser
* Change the font to the imported font in the Widget Blueprint, under text in details panel.

Expose a variable to set that on the instance.

[Random Unit Vector] \* a float value like 100 to make a random transform.

Add Static Mesh Component in a for loop in constructor for randomly generated content