# CHAPTER 05 :-

* **To Show Some Message in Output Log :-**

UE\_LOG(LogTemp, Warning, TEXT("Begin Play Called!")) ;

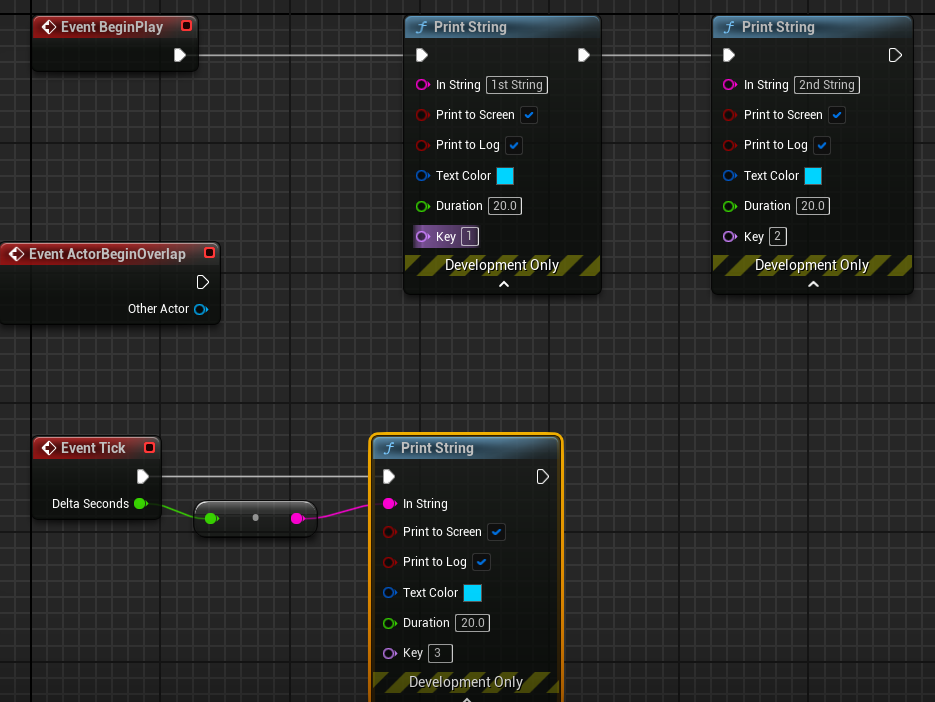
* **To Show Debug Message of Actor On Screen :-**

if (GEngine) // If it is 0 means nullptr and will not run so check pointer 1st.

{

GEngine -> AddOnScreenDebugMessage(1, 60.0f, FColor::Cyan, FString("Testing Item On Screen!"));

}



* **To Debug some format thing like frame seconds OR Actor\_Name :-**

FString Name = GetName() ;

FString Message = FString :: Printf(TEXT("Item Name = %s") , \*Name ) ;

FString Message = FString :: Printf(TEXT("DeltaTime = %f") , DeltaTime ) ;

GEngine -> AddOnScreenDebugMessage(1, 60.0f, FColor::Cyan, Message);

UE\_LOG(LogTemp, Warning, TEXT("Item Name : %s"), \*Name); // OR By Using Log also.

* **To Draw A Debug Sphere Over Actor while gameplay :-**

🡪 Important to include this header file #include<DrawDebugHelpers.h>

UWorld\* World = GetWorld() ;

if ( World )

{

FVector Location = GetActorLocation() ;

DrawDebugSphere( World , Location , 25.0f , 24 , FColor :: Blue , false , 30.f ) ;

}

// OR by using Macro

#define MY\_DEBUG\_SPHERE( Location ) { if( GetWorld( ) ) DrawDebugSphere( GetWorld( ) , Location , 25.0f , 12 , FColor :: Red , true ) ; } // If true so debug sphere will never go away

* **To Draw A Debug Line Along Actor :-**

UWorld\* World = GetWorld();

FVector Location = GetActorLocation();

FVector Forward = GetActorForwardVector() ; // It gets the forward (x-axis as we need to line to scale horizontally) unit vector (which means of 1 unit of UE i.e. 1cm ).

if( World )

{

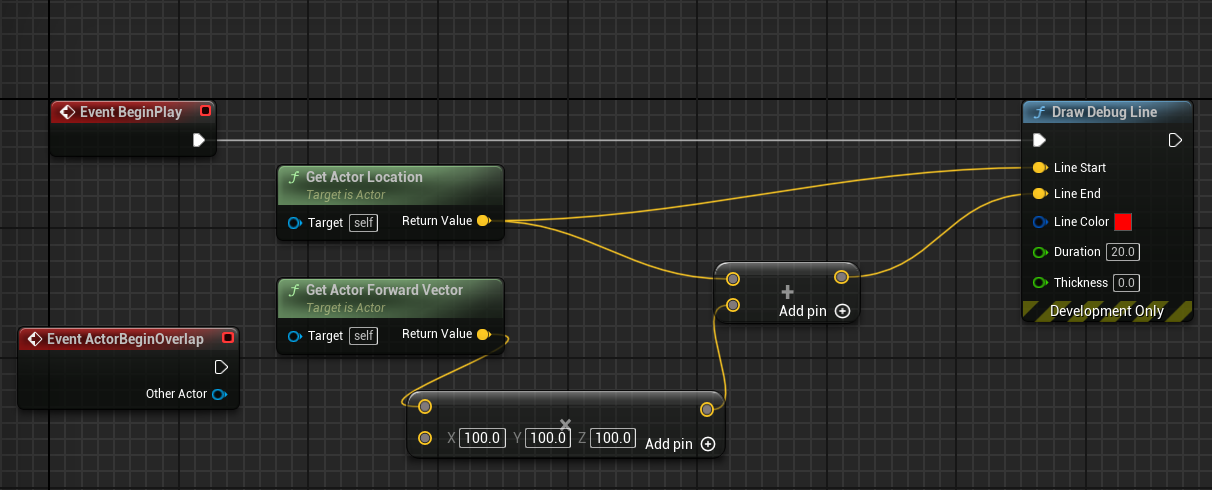
DrawDebugLine(World, Location, Location + 100.0f \* Forward, FColor :: Red, true , -1.0f , 0 , 1.0f ) ;

}

// OR

#define DRAW\_LINE( Start\_Location , End\_Location ) { if( GetWorld( ) ) DrawDebugLine( GetWorld( ) , Start\_Location , End\_Location , FColor :: Red , true , -1.0f , 0 , 1.0f ) ; }

🡪 Macro\_Call DRAW\_LINE(Location, Location + Forward \* 100.0f )



In Unreal Engine 5 (UE5), the “**Super”** keyword is used to call the implementation of a function in the parent class (or superclass). When you override a function in a subclass, you might still want to execute the functionality defined in the parent class before or after your subclass's code. This is where **Super** comes into play.

* **To Draw A Debug Point :-**

UWorld\* World = GetWorld();

if (World)

{

DrawDebugPoint(World, Location + Forward \* 100.0f, 15.0f, FColor :: Red, true);

}

// OR

#define DRAW\_POINT( Location ) { if( GetWorld( ) ) DrawDebugPoint( GetWorld() , Location , 15.0f, FColor :: Red, true ); }

DRAW\_POINT( Location + Forward \* 100.0f )

* **To Draw Both Debug Point and Line at the Same Place :-**

// BackSlash allows macro definition to be entered in new line also.

#define DRAW\_VECTOR( Start\_Location , End\_Location ) if ( GetWorld( ) ) \

{ DrawDebugLine(GetWorld(), Start\_Location, End\_Location, FColor :: Red, true, -1.0f, 0, 1.0f); \

DrawDebugPoint(GetWorld(), End\_Location, 15.0f, FColor :: Red, true); \

}

🡪 Macro\_Call DRAW\_VECTOR(Location, Location + Forward \* 100.0f)

* **OTHERS :-**

DrawDebugCircle(GetWorld(), CircleMatrix, 200, 50, FColor(0,104,167), true, -1, 0, 10);

DrawDebugCircle(GetWorld(), LocationFour, 200, 50, FColor(0,0,0), true, -1, 0, 10);

DrawDebugSolidBox(GetWorld(), MyBox, FColor(20, 100, 240), MyTransform, true);

DrawDebugBox(GetWorld(), LocationFive, FVector(100,100,100), FColor::Purple, true, -1, 0, 10);

DrawDebugLine(GetWorld(), LocationTwo, LocationThree, FColor::Emerald, true, -1, 0, 10);

DrawDebugCoordinateSystem(GetWorld(), Location, Rotation, 150.f, true , -1.0f , 0 , 1.0f )

DrawDebugDirectionalArrow(GetWorld(), FVector(-300, 600, 600), FVector(-300, -600, 600), 120.f, FColor::Magenta, true, -1.f, 0, 5.f);

DrawDebugCrosshairs(GetWorld(), FVector(0,0,1000), FRotator(0,0,0), 500.f, FColor::White, true, -1.f, 0);

DrawDebugCamera(World, Location, Rotation, 0.0f, 1.0f, FColor :: Red, true);

# CHAPTER 06 :-

**NOTE :-** If something doesn’t look correct just after you hot reloaded the C++ code ; just close the editor and build from Visual Studio.

* **To Set the location of Actor :-**

If we do this by blueprint and then debug shapes with C++ code ; we will see that debug shapes will be drawn at the new location instead of original one ; So it means if we do something in BeginPlay() in BluePrint then its C++ BeginPlay() code will happen after that.

SetActorLocation( FVector(0.0f, 0.0f, 100.0f) ) ; // The sweep flag is used for restricting movement. For example, if you want to prevent a player from walking through walls you can use the sweep flag to make sure the movement stops at the first blocking volume.

FVector Location = GetActorLocation();

FVector Forward = GetActorForwardVector();

MY\_DEBUG\_SPHERE(Location)

DRAW\_VECTOR(Location, Location + Forward \* 100.0f )

* **To Set the Rotation of Actor :-**

SetActorLocation( FVector(0.0f, 0.0f, 100.0f) ) ;

SetActorRotation( FRotator( 0.0f , 45.0f , 0.0f ) ) ; // Here if we set the teleport flag to true, the movement happens without modifying the character’s velocity, and the antenna stays perfectly still on the character’s head.

FVector Location = GetActorLocation();

FVector Forward = GetActorForwardVector();

MY\_DEBUG\_SPHERE(Location)

DRAW\_VECTOR(Location, Location + Forward \* 100.0f )

* **To Set Actor Offset :-**

**AddActorWorldOffset()** moves the actor in world space, while **AddActorLocalOffset()** moves the actor according to its local axes, considering its rotation and parent-child relationships.

* And to get single frame debug shapes ; turn persistent lines to False and then LifeTime to -1.0 so that it updates for each tick or frame-rate.

#define DRAW\_SPHERE\_SINGLE\_FRAME( Location ) { if( GetWorld( ) ) DrawDebugSphere( GetWorld( ) , Location , 25.0f , 12 , FColor :: Red , false , -1.0f ) ; }

// We used the fixed rates so that the pace of offset is not dependent on any particular FPS like actor moves fast for low FPS and slow for high FPS systems ; hence multiply the factor with DeltaTime so it can be adjusted according to the system specs. Such that this offset is intended to be framerate-independent, meaning it will result in a consistent displacement over time regardless of the frame rate.

float MovementRate = 50.0f ; // 50 units in UE = 50cm

float RotationRate = 45.0f; // 45 unit of rotation in UE = 45 degrees

// MovementRate \* DeltaTime() => Unit = ( cm/s \* s/frame ) => cm/frame .

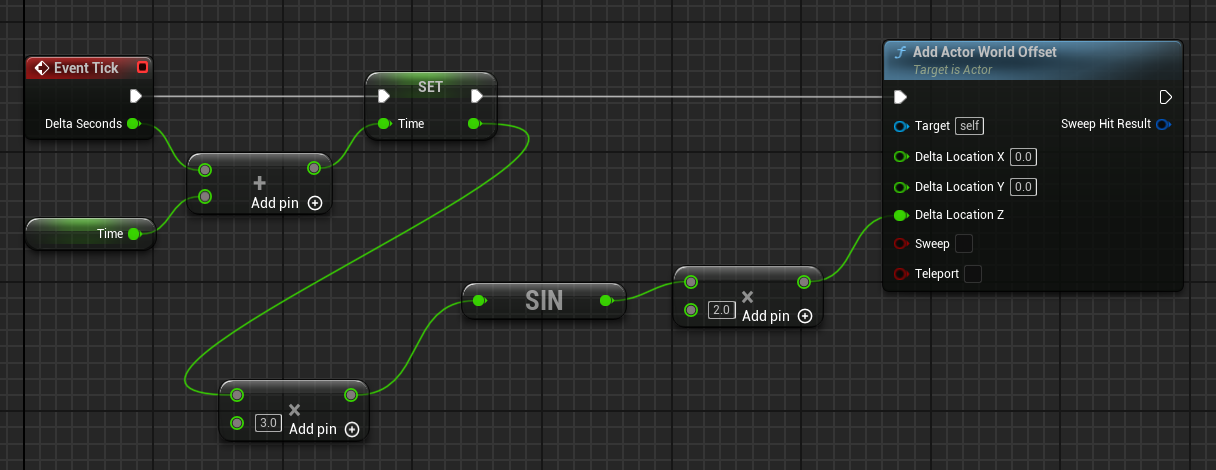
AddActorWorldOffset(FVector( MovementRate \* DeltaTime , 0.0f, 0.0f));

AddActorWorldRotation(FRotator(0.0f, RotationRate \* DeltaTime, 0.0f));

DRAW\_SPHERE\_SINGLE\_FRAME( GetActorLocation( ) )

DRAW\_VECTOR\_SINGLE\_FRAME( GetActorLocation( ) , GetActorLocation( ) + GetActorForwardVector( ) \* 100.0f )

* **To Offset Actor Periodically :-**



// Hard-Coded

Running\_Time += DeltaTime; // Made a member variable of AItem in Items.h

float Delta\_Z = 2.0f \* FMath :: Sin( Running\_Time \* 2.0f ) ; // Here Adding in Angle will speed up the sine wave So Process will hapen quickly while adding in sine answer will increase the amplitude of the offset.

AddActorWorldOffset(FVector(0.0f, 0.0f, Delta\_Z));

DRAW\_SPHERE\_SINGLE\_FRAME( GetActorLocation( ) )

DRAW\_VECTOR\_SINGLE\_FRAME( GetActorLocation( ) , GetActorLocation( ) + GetActorForwardVector( ) \* 100.0f )

// OR Initialize Members

Amplitude = 2.0f; // The method of initializing it in Constructor would be less-efficient as it will take 1 more step rather than direct variable initialization in "private" OR Constructor Initializer List.

In Items.h 🡪 class SLASH\_API AItem : public AActor

private:

float Running\_Time ;

float Amplitude = 0.25f ;

float Time\_Constant = 0.5f;

Running\_Time += DeltaTime;

float Delta\_Z = 2.0f \* FMath :: Sin( Running\_Time \* 2.0f ) ; // sin period = 2\*pi / k ;

AddActorWorldOffset(FVector(0.0f, 0.0f, Delta\_Z));

DRAW\_SPHERE\_SINGLE\_FRAME( GetActorLocation( ) )

DRAW\_VECTOR\_SINGLE\_FRAME( GetActorLocation( ) , GetActorLocation( ) + GetActorForwardVector( ) \* 100.0f )

* **To Expose C++ Variables to Blueprints :-**

UPROPERTY is a macro used to declare properties in classes derived from UObject . UPROPERTY provides various functionalities and metadata that can affect how properties behave and how they are handled by the engine. Like UPROPERTY( EditAnywhere )

EditDefaultsOnly 🡪 We can edit it but only in the default Blueprint.

EditInstanceOnly 🡪 We can edit and see it only using the Instance.

EditAnywhere 🡪 It gives flexibility as it is editable using both default and instances.

But it is least restrictive like if we changed the value in default blueprint without overwriting it in instance one so both are changed alongside ; but if we changed the default one in instance so changes of default blueprint did not take place in instance as it is manually overwritten unless resetted the values.

* **To Expose Variables Only NOT Editable :-**

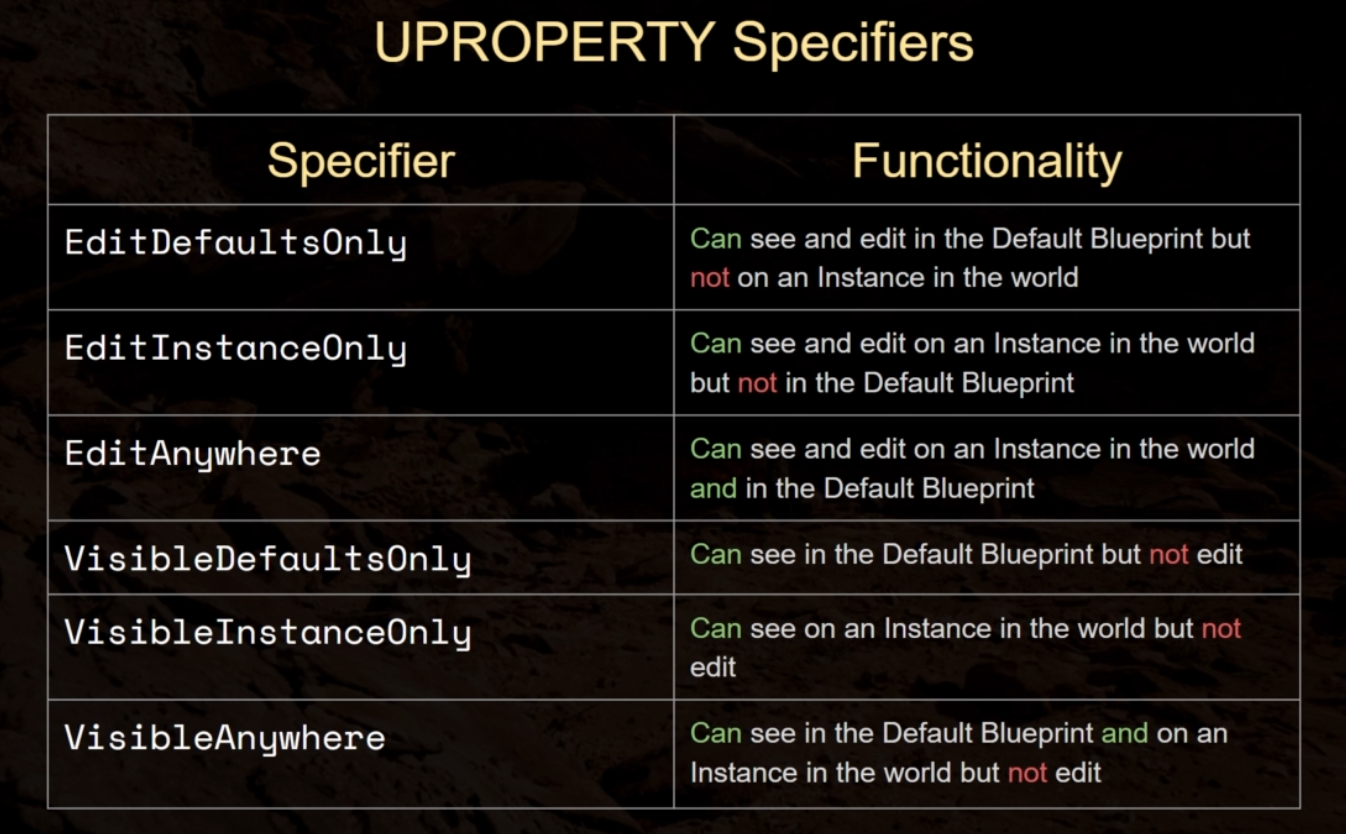
VisibleDefaultsOnly 🡪 Indicates that this property is only visible in property windows for archetypes, and cannot be edited.

In our example :- This Specifier does not have its own Tick function as its associated with Instance so we can not see it being updated.

VisibleInstanceOnly 🡪 Indicates that this property is only visible in property windows for instances, not for archetypes, and cannot be edited.

In our example :- This Specifier is Instance one so it has its own Tick function so the Running\_Time can be seen being updated.

VisibleAnywhere 🡪 Indicates that this property is visible in all property windows, but cannot be edited.



* **To Expose variables to Event Graph :-**

**Note :-** In order to use Blueprint Specifiers directly ; the members can’t be “private” so move them to “protected” section. As then it will potentially compromise ; the encapsulation and data integrity of the class.

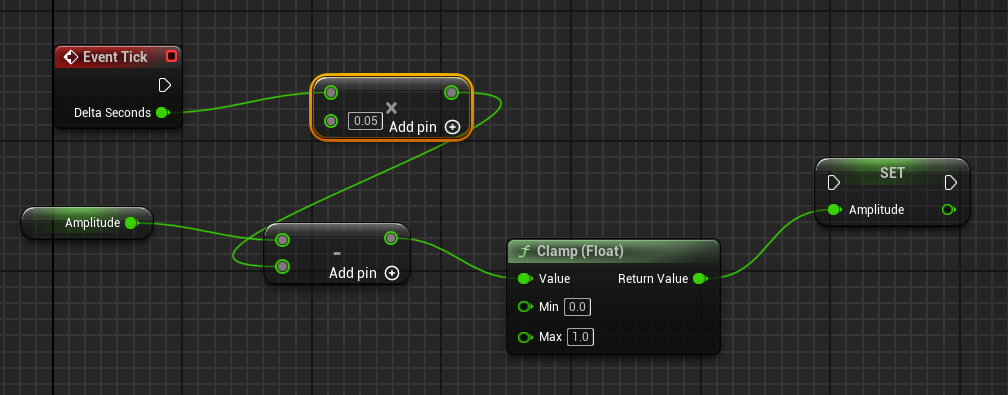
BlueprintReadOnly 🡪 This property can be read by Blueprints, but not modified. This Specifier is incompatible with the BlueprintReadWrite Specifier.

BlueprintReadWrite 🡪 This property can be read or written from a Blueprint. This Specifier is incompatible with the BlueprintReadOnly Specifier.

UPROPERTY(EditAnywhere, BlueprintReadWrite )

However if you still want to expose a property as read-write from within Blueprints while keeping it private in C++, you can use the AllowPrivateAccess meta specifier which enables access to the private property for reflection purposes.

UPROPERTY( VisibleInstanceOnly , BlueprintReadOnly , meta = ( AllowPrivateAccess = "true" ) )



// Clamp() function is used to restrict a value within a specified range. It ensures that a given value stays within a specified minimum and maximum bounds. If the value provided to Clamp() is outside the specified range, it will be "clamped" or adjusted to fit within that range.

* **To Change the name of Category in Blueprint and Instance Details :-**

UPROPERTY(EditAnywhere, BlueprintReadWrite , Category = "Sine Parameters" )

* **To Expose Functions to Blueprint Event Graph :-**

BlueprintCallable 🡪 It makes the function executable  in a Blueprint or Level Blueprint graph with an input and output execution pin.

BlueprintPure 🡪 It makes the function only retrieve/input its value in Blueprint or Event Graph but it can’t change the properties of another actor.

UFUNCTION(BlueprintPure)

float TransformedSine( );

UFUNCTION( BlueprintPure )

float TransformedCosine( );

float AItem::TransformedCosine()

{

return Amplitude \* FMath :: Cos(Running\_Time \* Time\_Constant);

}

* **To Change Type using Template Functions :-**

**Note :-** Some Built-in Members doesn’t have the overloaded functions so template function won’t be called on that like below FRotator doesn’t support division.

template<typename T> // In “protected” Section

T Avg(T First, T Second);

template<typename T>

inline T AItem::Avg(T First, T Second)

{

return ( First + Second ) / 2 ;

}

int32 AvgInt = Avg<int32>(4, 6);

UE\_LOG( LogTemp , Warning , TEXT("Avg of 4 and 6 is %d") , AvgInt )

FVector AvgVector = Avg<FVector>(GetActorLocation(), FVector :: ZeroVector);

DRAW\_POINT\_SINGLE\_FRAME(AvgVector);

FRotator AvgRotator = Avg<FRotator>(GetActorRotation( ), FRotator :: ZeroRotator ); => Not Supported

* **To Add Components :-**

Via Blueprints ; Open the viewport and add the component you like OR drop from the content browser.

// In “private” Section

UPROPERTY( VisibleAnywhere )

UStaticMeshComponent\* ItemMesh ;

// In “Constructor” Of Item

ItemMesh = CreateDefaultSubobject<UStaticMeshComponent>(TEXT("ItemMeshComponent")); // Unreal Engine uses factory functions for pointers rather than "new".

RootComponent = ItemMesh ; // Here we replace the Rootcomponent with ItemMesh for DefaultSceneRootComponent so Unreal's garbage collection system will see that since RootComponent points nothing so will delete its pointer automatically.

# CHAPTER 07 :-

* **Setting up Pawn using Capsule Component :-**

Since mesh is made up of large number of polygons ; so checking collision against mesh will be much expensive operation ; So we use a more basic shape like Capsule used as the simple collision representation for pawn classes.

#include <Components/CapsuleComponent.h>

#include "Bird.generated.h" // The generated.h must be the final header file else compiler will give error as when UE5 will include genereated.h ; pre-processor will paste in auto-generated tons of code and integrate it with reflection system.

private:

UPROPERTY( VisibleAnywhere )

UCapsuleComponent\* Capsule ;

ABird::ABird()

{

PrimaryActorTick.bCanEverTick = true;

Capsule = CreateDefaultSubobject<UCapsuleComponent>(TEXT("Capsule"));

//RootComponent = Capsule ; // OR

SetRootComponent(Capsule);

Capsule -> SetCapsuleHalfHeight(20.0f);

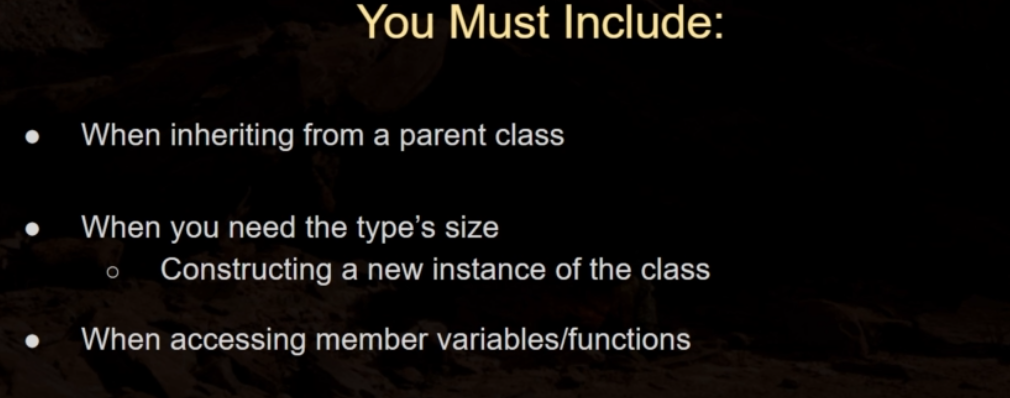
Capsule -> SetCapsuleRadius(15.0f);

}

* **Forward Declaration :-**

Since we only need the UCapsuleComponent but all the other code will also be included with the header file and when we include “Bird.h” in “Bird.cpp” so it will again have to include the same header file ; likewise this in a large project will result in **code bloat** which can slow down compile time a great deal. Also if we include 2 header files in each other which will give error of Circular Dependency like “Weapon.h” in “Warrior.h” and “Warrior.h” in “Weapon.h” . So only declare header file where you need it or using its instance.

Forward declaration is a technique used in programming, like in 5 (UE5), to declare a class or struct without fully defining it. Instead of including the entire header file where the class or struct is defined, you simply declare its existence thus reducing compile time and avoiding unnecessary dependencies.



class UCapsuleComponent\* Capsule ; // Forward Declaration in “Bird.h”

UCapsuleComponent\* Second\_Capsule; // Will not give error as forward declare above.

OR

class UCapsuleComponent ; // You can declare after declaraing all header files.

#include <Components/CapsuleComponent.h> // Included in “Bird.cpp” where we use its instance

Capsule = CreateDefaultSubobject<UCapsuleComponent>(TEXT("Capsule"));

* **Setup SkeletalMeshComponent :-**

class USkeletalMeshComponent;

UPROPERTY( VisibleAnywhere )

USkeletalMeshComponent\* BirdMesh;

#include "Components/SkeletalmeshComponent.h" // Included in “Bird.cpp”

BirdMesh = CreateDefaultSubobject<USkeletalMeshComponent>(TEXT("BirdMesh"));

BirdMesh -> SetupAttachment(GetRootComponent());

* **To Map The Controller For Initial Building Inputs (Old) :-**

AutoPossessPlayer = EAutoReceiveInput :: Player0 ; // To Posses Pawn ( In Constuctor )

First go to Project Settings and Engine 🡪 Input 🡪 Axis Mapping then Assign Keys :-

protected:

void MoveForward(float Value);

// We can give the function any name ; but we must give parameter as **float** as it is the type of function which can be bound to axis mapping ; Once done , we suddenly have the binding between Axis mapping and function ( which is called per every frame )

void ABird::MoveForward(float Value)

{

UE\_LOG(LogTemp, Warning, TEXT("Value = %f"), Value); // Value = ( 1.0 OR Scale ) when TRUE ; 0.0 for False

}

void ABird::SetupPlayerInputComponent(UInputComponent\* PlayerInputComponent)

{

Super::SetupPlayerInputComponent(PlayerInputComponent);

//PlayerInputComponent -> BindAxis(TEXT("MoveForward")); // OR

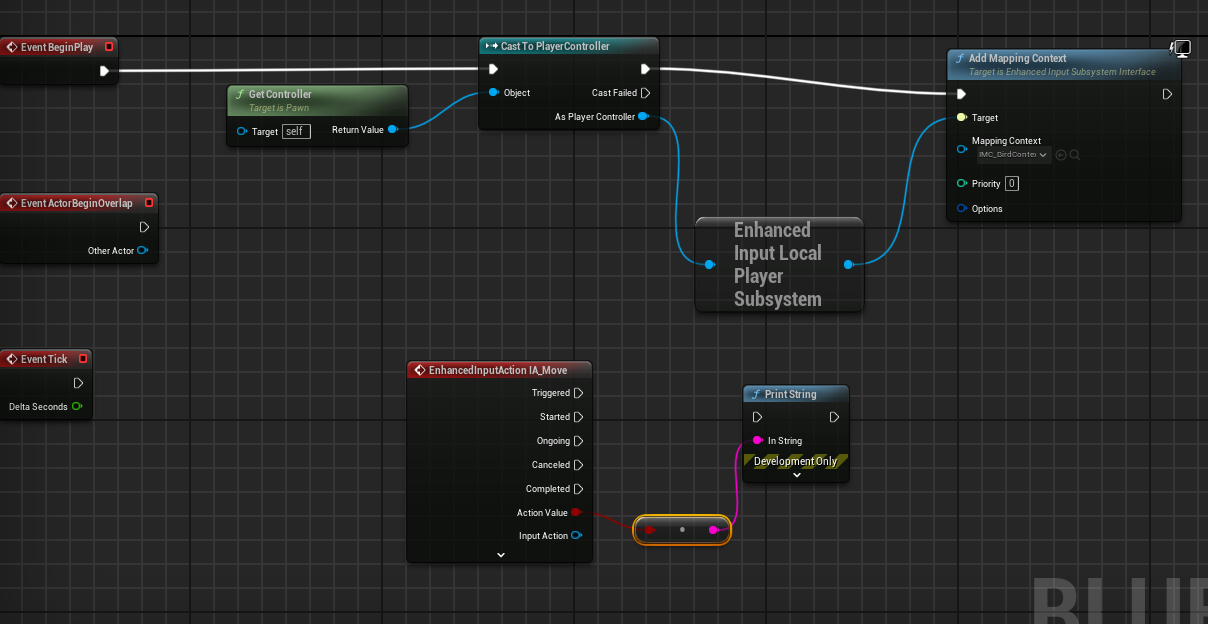
PlayerInputComponent -> BindAxis(FName("MoveForward") , this , &ABird::MoveForward ) ; // 1st Name of Axis Mapping to link ; 2nd User Class pointer ( User Class exists in higher level in hierarchy in UE and Bird Pawn is Descendant of that class so can pass its pointer ) ; 3rd User Object ( To which function we want to bind “Move Forward” function. )

}

* **Binding Enhanced Input ( New ) :-**

ShowDebugEnhancedInput 🡪 To check whether the actions and axis mapping is being triggered or not .

Also we an have different action keys depending on priority keys to handle collision between them.



“In Bird.h”

#include "InputActionValue.h" // As FInputActionValue is a struct with not returning pointer so can't forward declare it hence we simply include in Bird.h before "Bird.generated.h"

class UInputMappingContext;

class UInputAction;

protected:

virtual void BeginPlay() override;

// Note:- We must choose the mapping context and Move action from the Blueprint Details or else pointer will give NULL and program will not run as intended.

UPROPERTY( EditAnywhere , BlueprintReadOnly , Category = "Input" )

UInputMappingContext\* BirdMappingContext ;

UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Input")

UInputAction\* MoveAction;

void Move(const FInputActionValue& Value);

“In Bird.cpp”

#include "Components/InputComponent.h"

#include "EnhancedInputSubsystems.h"

#include "EnhancedInputComponent.h"

void ABird::BeginPlay()

{

Super::BeginPlay();

// OR Simply Controller ( as its in public section in UE 5.1) ( Also can declare directly )

if (APlayerController\* PlayerController = Cast<APlayerController>(GetController()) )

{

UEnhancedInputLocalPlayerSubsystem\* Subsystem = ULocalPlayer :: GetSubsystem<UEnhancedInputLocalPlayerSubsystem>( PlayerController -> GetLocalPlayer());

if (Subsystem)

{

Subsystem -> AddMappingContext(BirdMappingContext, 0); // Here 0 is priority key

}

}

}

void ABird::Move(const FInputActionValue& Value)

{

const bool CurrentValue = Value.Get<bool>(); // Has to be same type as IA\_Move

if (CurrentValue)

{

UE\_LOG(LogTemp, Warning, TEXT("IA\_Move Trigerred"));

}

}

void ABird::SetupPlayerInputComponent(UInputComponent\* PlayerInputComponent)

{

Super::SetupPlayerInputComponent(PlayerInputComponent);

// CastChecked will not only just Cast the input of UInputComponent to EnhancedInputComponent but if failed to do so will also crash the program.

if (UEnhancedInputComponent\* EnhancedInputComponent = CastChecked<UEnhancedInputComponent>(PlayerInputComponent))

{

EnhancedInputComponent -> BindAction(MoveAction, ETriggerEvent :: Triggered, this, &ABird :: Move);

}

}

* **Adding Movement Input ( Old ) :-**

void ABird::MoveForward(float Value)

{

// Placed ( GetController() ) condition so that only that Pawn moves which has been possessed.

if ( ( GetController( ) != nullptr ) && (Value != 0.0f))

{

FVector Forward = GetActorForwardVector();

AddMovementInput(Forward, Value); // It will automatically detect whether value is 0 ( do nothing ) ; +ve ( Consider Original Vector ) ; -ve ( Consider Negative Vector )

}

}

**Note :-** AddMovementInput() passes the direction offset to movement component ; but since Pawns by default have no movement component ; so add new component in Blueprint “Floating Pawn Movement” . Now through it we can change movement properties also.

**MY CHANGEMENTS**

void ABird::MoveSideways(float Value)

{

if ((GetController()) && (Value != 0.0f))

{

FVector Sideway = GetActorRightVector() ;

AddMovementInput(Sideway, Value);

}

}

void ABird::MoveUpDown(float Value)

{

if ((GetController()) && (Value != 0.0f))

{

FVector UpDown = GetActorUpVector() ;

AddMovementInput(UpDown, Value);

}

}

void ABird::SetupPlayerInputComponent(UInputComponent\* PlayerInputComponent)

{

Super::SetupPlayerInputComponent(PlayerInputComponent);

PlayerInputComponent -> BindAxis(FName("MoveForward"), this, &ABird::MoveForward);

PlayerInputComponent -> BindAxis(FName("MoveSideways"), this, &ABird::MoveSideways);

PlayerInputComponent -> BindAxis(FName("MoveUpDown"), this, &ABird::MoveUpDown);

}

* **Adding Movement Input ( Enhanced ) :-**

void ABird::Move(const FInputActionValue& Value)

{

const float DirectionValue = Value.Get<float>(); // Has to be same type as IA\_Move

if ((GetController() != nullptr) && (DirectionValue != 0.0f))

{

FVector Forward = GetActorForwardVector();

AddMovementInput(Forward, DirectionValue);

}

}

void ABird::SetupPlayerInputComponent(UInputComponent\* PlayerInputComponent)

{

Super::SetupPlayerInputComponent(PlayerInputComponent);

if (UEnhancedInputComponent\* EnhancedInputComponent = CastChecked<UEnhancedInputComponent>(PlayerInputComponent))

{

EnhancedInputComponent -> BindAction(MoveAction, ETriggerEvent :: Triggered, this, &ABird :: Move); } }

MY Moventment Function :-

void ABird::Move(const FInputActionValue& Value)

{

const FVector2D MyVector = Value.Get<FVector2D>();

if ( ( GetController() ) && ( ( MyVector.X != 0.0f ) || ( MyVector.Y != 0.0f ) ) ) {

FVector Forward = GetActorForwardVector();

AddMovementInput(Forward, MyVector.X);

FVector Right = GetActorRightVector();

AddMovementInput(Right, MyVector.Y);

}

}

* **Adding Camera and Spring Arm Components :-**

Add the “Camera” and “Spring Arm” components to Pawn Blueprint class and then attatch the spring arm camera to Capsule component and Camera to spring arm component by holding the menu button and dropping on it. Then you can adjust accordingly camera accordingly to you spring arm. It is usually recommended to include a Spring Arm to automatically control how the camera handles situations where it becomes obstructed by level geometry or other objects so it ZOOMS the object accordingly.

“In C++ ; if you get the error that details of newly added components are not showing up so change the component variable name ; close UE and build from VS. It’s a bug.”

class USpringArmComponent;

class UCameraComponent;

private:

UPROPERTY( VisibleAnywhere )

USpringArmComponent\* SpringArm;

UPROPERTY(VisibleAnywhere)

UCameraComponent\* ViewCamera;

#include "GameFramework/SpringArmComponent.h"

#include "Camera/CameraComponent.h"

// In Contructor

SpringArm = CreateDefaultSubobject<USpringArmComponent>(TEXT("CameraBoom"));

SpringArm -> SetupAttachment(GetRootComponent());

SpringArm -> TargetArmLength = 300.0f;

SpringArm -> SetRelativeRotation(FRotator(0.f, -40.f, 0.f));

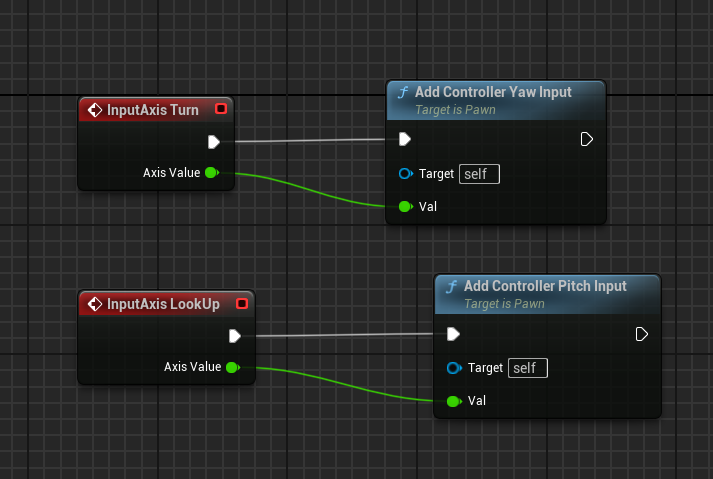
ViewCamera = CreateDefaultSubobject<UCameraComponent>(TEXT("ViewCamera"));

ViewCamera -> SetupAttachment(SpringArm);

* **Adding Rotation to Controller (Old) :-**

First add the axis bindings for Mouse like where MouseX refers to horizontal motion of mouse and MouseY refers to vertical motion. Now create the Blueprint then select Blueprint self to Set “Use Controller Rotation Pitch and Yaw” to True.

Here we bind “MouseX” with Yaw ( Y-Rotation ) and “MouseY” with Pitch ( X-Rotation ) as the Unreal World Positions are in that direction i.e. their X refers vertical and Y refers horizontal. Also we changed the scale of MouseY to -1.0 so to negate the motion i.e. when Mouse is moved to below ; Controller also rotates below instead of UP force.



void ABird::Turn(float Value)

{

AddControllerYawInput(Value);

}

void ABird::LookUp(float Value)

{

AddControllerPitchInput(Value);

}

void ABird::SetupPlayerInputComponent(UInputComponent\* PlayerInputComponent)

{

Super::SetupPlayerInputComponent(PlayerInputComponent);

PlayerInputComponent -> BindAxis(FName("MoveForward"), this, &ABird::MoveForward);

PlayerInputComponent -> BindAxis(FName("MoveSideways"), this, &ABird::MoveSideways);

PlayerInputComponent -> BindAxis(FName("Turn"), this, &ABird::Turn);

PlayerInputComponent -> BindAxis(FName("LookUp"), this, &ABird::LookUp); }

* **Adding 2D-Rotation to Camera ( Enhanced ) :-**

**Note:-** Remember to update the input action through Blueprint OR else will give null pointer to functions as Blueprints are executed 1st then C++ code.

UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Input" )

UInputAction\* LookAction;

void Look(const FInputActionValue& Value);

void ABird::Look(const FInputActionValue& Value)

{

const FVector2D LookAxisValue = Value.Get<FVector2D>();

if( GetController() )

{

AddControllerYawInput(LookAxisValue.X);

AddControllerPitchInput(LookAxisValue.Y);

}

}

void ABird::SetupPlayerInputComponent(UInputComponent\* PlayerInputComponent)

{

Super::SetupPlayerInputComponent(PlayerInputComponent);

if (UEnhancedInputComponent\* EnhancedInputComponent = CastChecked<UEnhancedInputComponent>(PlayerInputComponent))

{

EnhancedInputComponent -> BindAction(MoveAction, ETriggerEvent :: Triggered, this, &ABird :: Move);

EnhancedInputComponent -> BindAction(LookAction, ETriggerEvent :: Triggered, this, &ABird :: Look);

}

}

* **Setting Default Pawn :-**

If we open another map instead of that we are configuring ; then we can also see the Deafult\_Pawn sphere which the system automatically loads in and also if we fly our pawn Bird out of the section to other unloaded section ; then it will automatically be unloaded as Default\_Pawn is set as referenced one ; so to fix these things :

Create a Blueprint class of Game Mode and then open it and set default pawn class to BP\_Bird ; and then in “World Settings” change the GameMode Override to that BP\_BirdGameMode. Now wherever you position your camera OR your Game Start is placed from there Bird Pawn will automatically be possessed.