# CHAPTER 05 :-

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

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

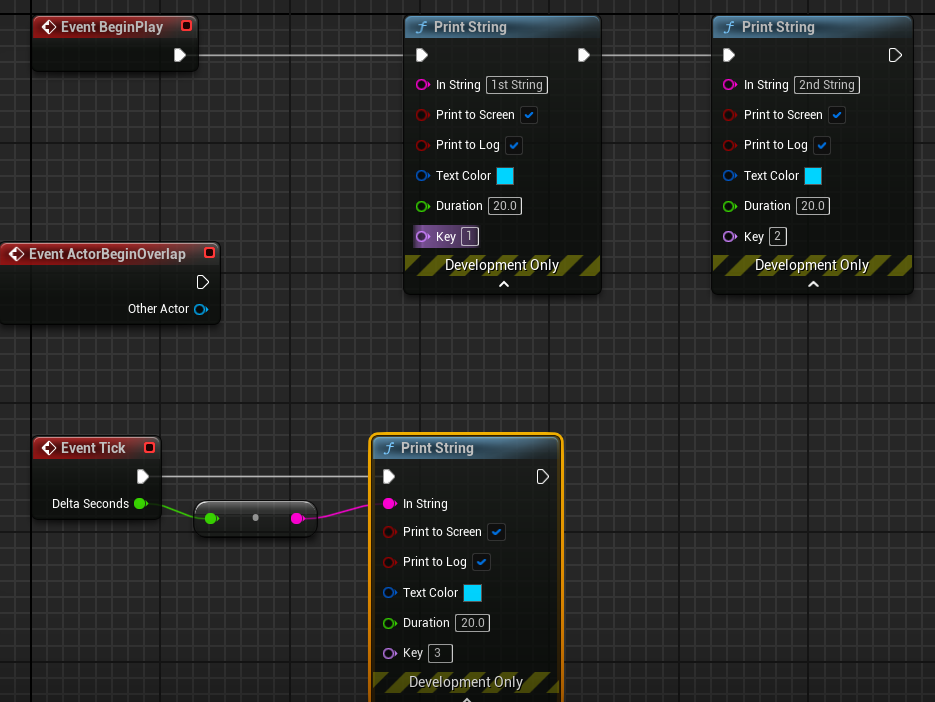
* **To Show Debug Message of Actor On Screen :-**

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 )

{

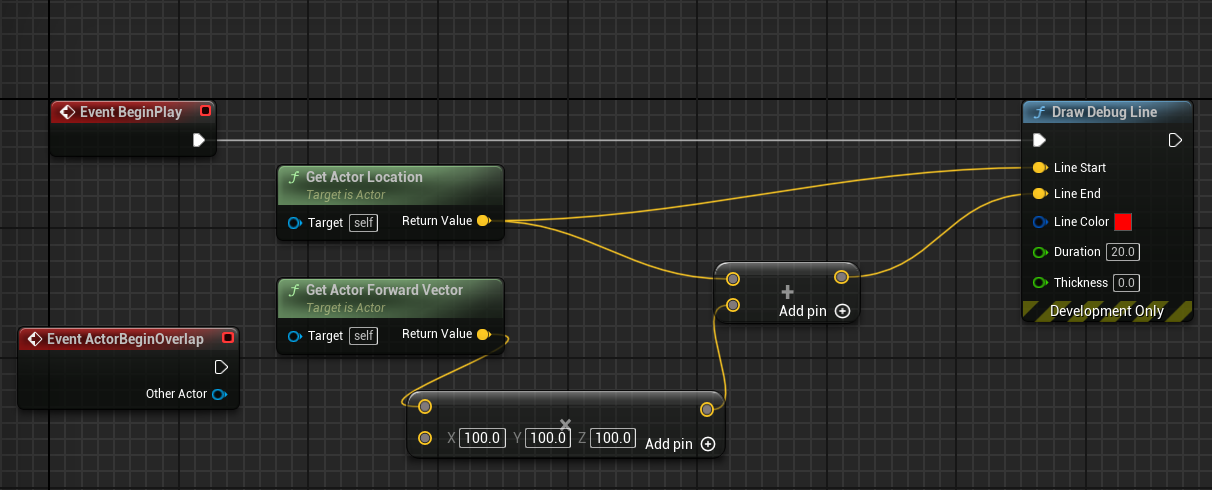
DrawDebugLine(World, Location, Location + 100.0f \* Forward, FColor :: Red, true , -1.0f , 0 , 1.0f ) ;

}

// 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 )



* **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 .

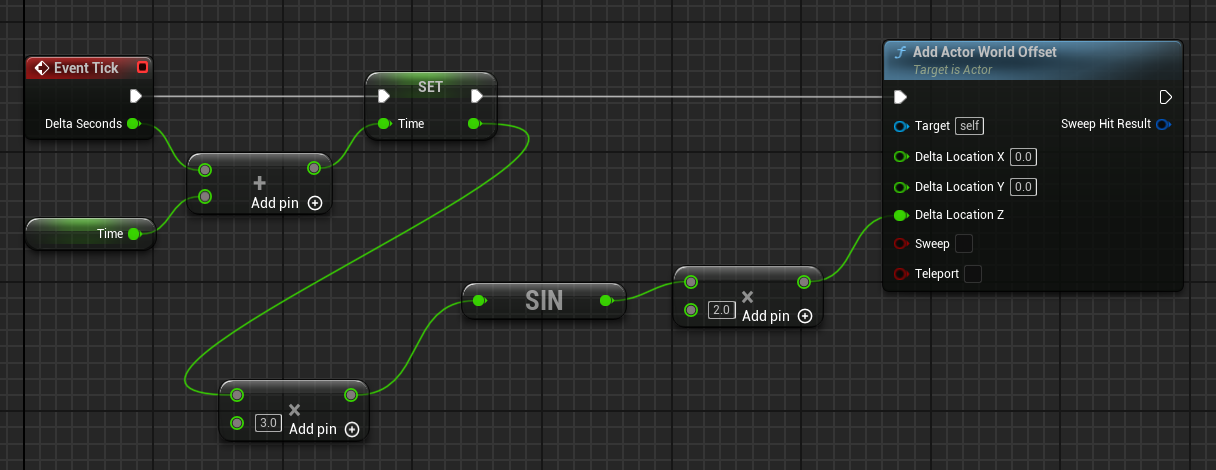
AddActorWorldOffset(FVector( MovementRate \* DeltaTime , 0.0f, 0.0f));

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 :-**

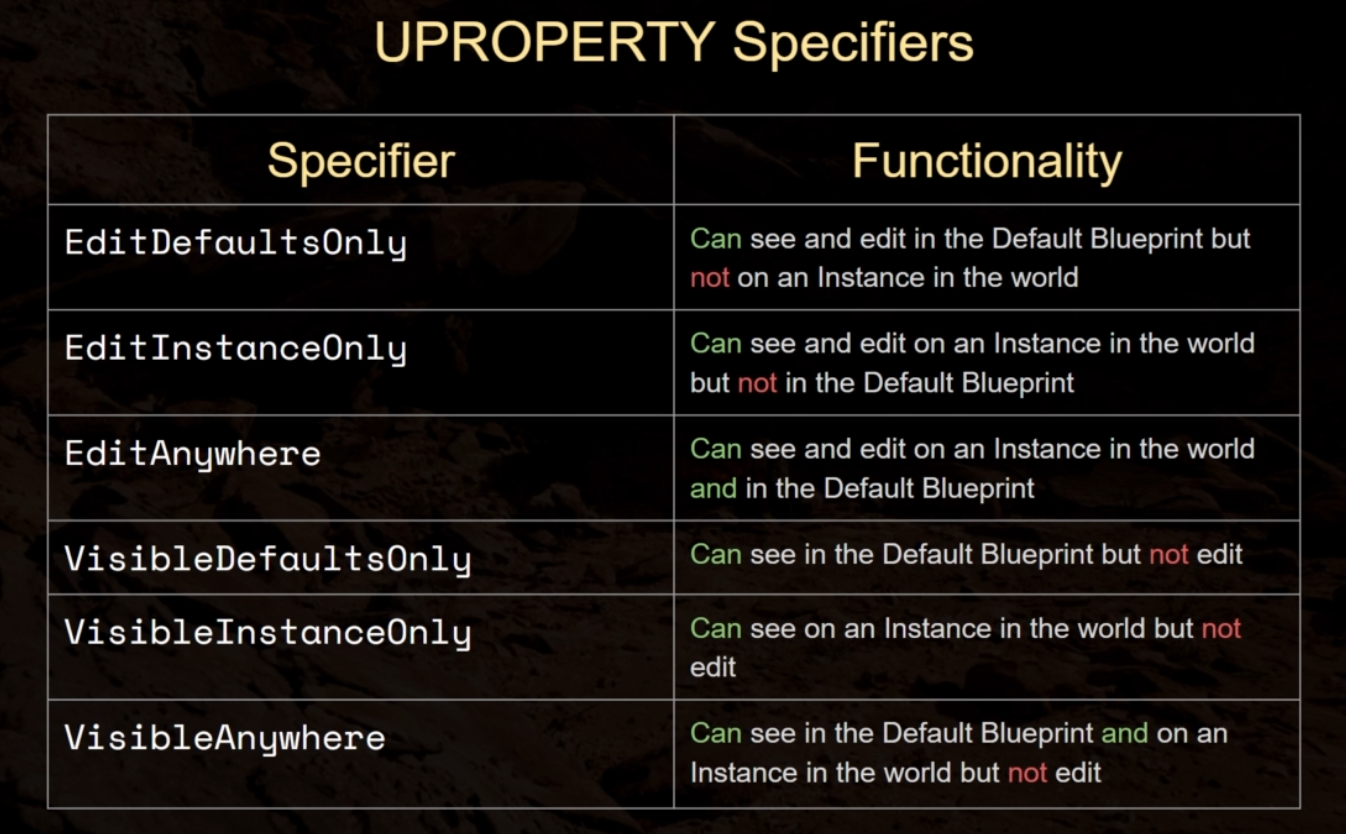
VisibleDefaultsOnly 🡪 Indicates that this property is only visible in property windows for archetypes, and cannot be edited.

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.

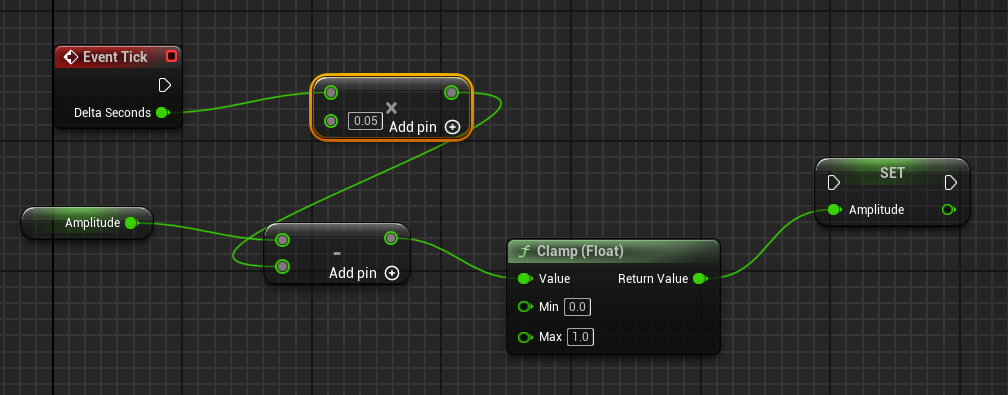
BlueprintReadOnly 🡪 This property can be read by Blueprints, but not modified. This Specifier is incompatible with the BlueprintReadWrite Specifier.

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.