Notes on Unreal Engine

Lance Putnam
Computing, Goldsmiths College

January 30, 2023

Contents

1 Unreal Editor

1.1 What Is It?

Unreal Editor is a visual front-end to the Unreal game engine [48]. The Editor allows you to edit scenes (like other CAD software), manage assets (3D models, textures, sounds and any other static data) and program game logic. Programming comes in two flavors: visual using Blueprints and textual using C++. Blueprints is generally more suitable for beginner or novice programmers and C++ for more advanced programmers. Since the Unreal Engine is written in C++ (originating from the first-person shooter game from the 90s called Unreal), the visual and textual programming interfaces are tightly coupled and share many of the same principles. Even if you are a seasoned C++ programmer, it is wise to learn Blueprints first to understand how the engine works and what is possible.

1.2 Before Plunging In

First things first: do not get overwhelmed by your tools. They are meant to help you. Most advanced software tools have a tendency to accumulate features (AKA bloat) as they age. That said, underneath the bloat is typically a central, and hopefully principled, core. Fortunately, Unreal Editor has a solid and principled foundation based on making real games. Our plan will be to focus on a handful of core principles so that you can go off with confidence and explore its other features. It would be futile to attempt to cover the editor's myriad features anyway. We will also try to keep the focus on tool-agnostic concepts so whatever you learn here will transfer to other similar tools.

1.3 The Interface

The Unreal Editor interface [26] has a few standard panels that are worth getting familiar with (Figure 1). The center panel is the *Level Editor* and rendering preview. You can move the camera by either dragging with the mouse left and right buttons or using the `WASD` keys and select objects in the scene by clicking on them. Clicking on an object will create an outline around it and present three colored arrows that allow you to move the object. Just above the Level Editor is a bar with a *Play* button which will let you play the level. The two panels on the far right provide information about scene objects or *Actors* [10] as they are called in Unreal. The top-right panel *World Outliner* lists all the objects in the scene and the bottom-right panel *Details* reveals editable attributes about the currently selected object. Nearly all scene objects have a *Transform* attribute that lets you translate, rotate and scale the object. The bottom panel *Content Browser* gives you access to all the project's assets (3D models, textures, audio files, levels, etc.). The leftmost panel *Modes* allows you to select various entities to place into the scene. In UE5, the Modes panel has been largely replaced by a Quick Add button  at the top.

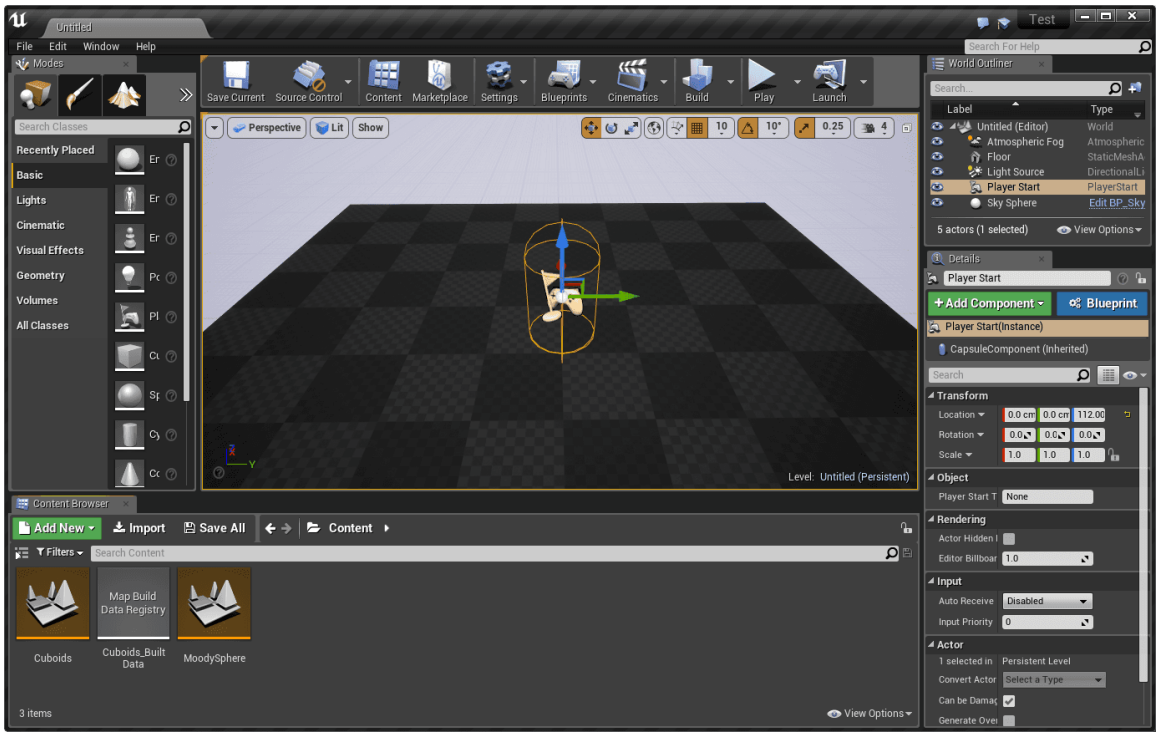


Figure 1: Unreal Editor interface.

1.4 Our First Scene

We will get familiar with the editor by creating a very basic scene that shows some cubes in front of the player. Begin by dragging a cube StaticMeshActor from the Modes panel (or Quick Add > Shapes) over to the Level Editor. Place the cube somewhere near the far end of the plane. You can move any object "freely" by clicking and dragging on it, however, it will not always go where you want it to. You will notice when you click an object it has colored RGB arrows around it. These arrows allow you to move the object more deliberately in the direction of the arrow. Move the cube to the right by clicking and dragging on the green arrow (Figure 2). Also notice in the Details panel under Transform, the Location of the actor changes as you move it.

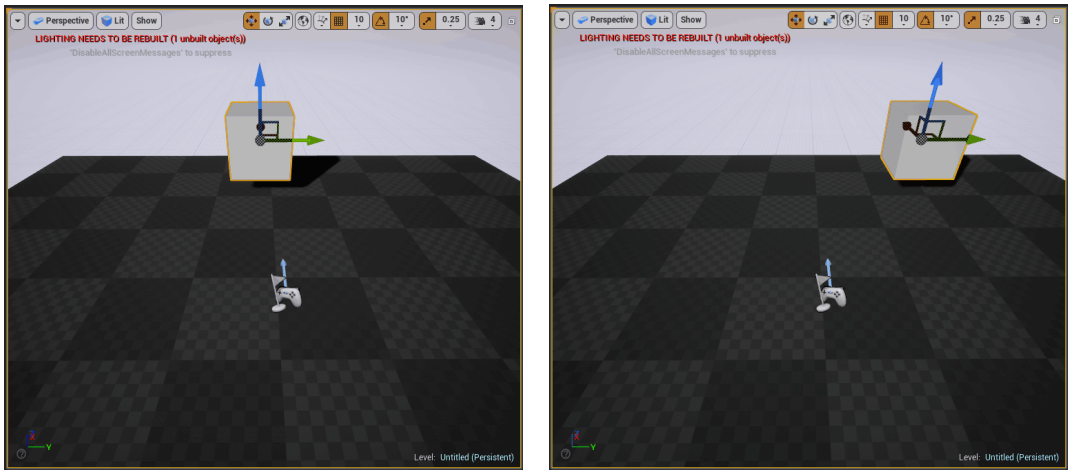


Figure 2: A cube added to the level. The cube (or any other actor) can be moved around by clicking on it and dragging on the colored arrows.

Next we will turn the cube into a vertical cuboid by modifying its Transform values (Figure 3) in the Details panel. Click the rightmost value of Scale and change it to 4. Note that Unreal uses a *left-handed coordinate system* so that x is forward (red), y is right (green) and z is up (blue).

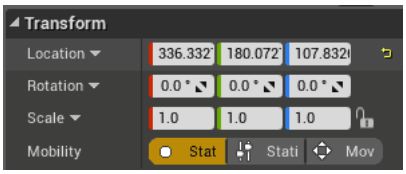


Figure 3: Actor Transform settings in the Details panel.

After scaling the height, it is likely the cuboid will be in the floor. Move it up by dragging upwards on the blue arrow in the Level Editor (Figure 4).

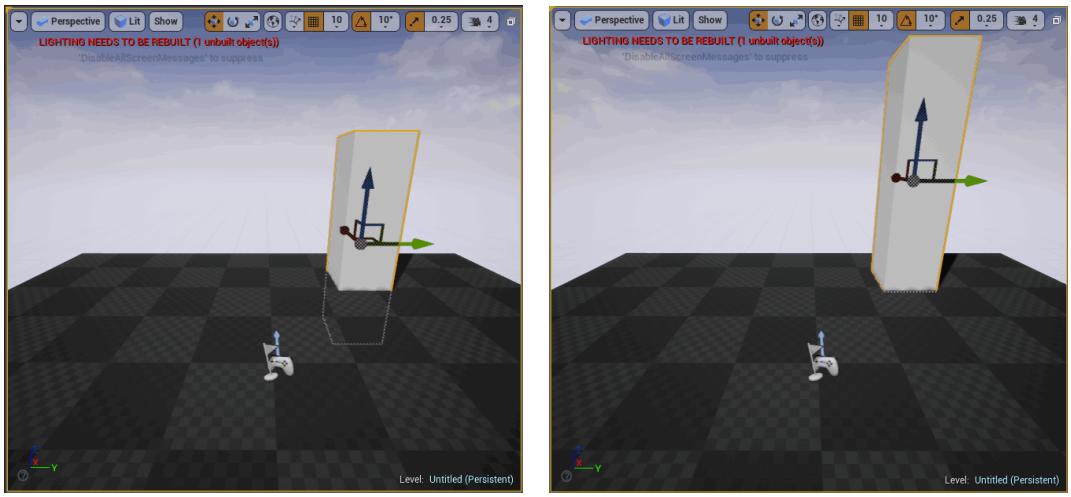


Figure 4: Our cube turned into a cuboid by modifying its scale transform. The dashed outline indicates the cuboid is behind the floor, so we move it up.

Next, we will create another vertical cuboid on the left. Rather than going through the whole process just detailed again, we will instead duplicate the shape. This is done by clicking the cuboid and then pressing `control-c` to copy it and then `control-v` to paste a new copy into the level. After doing so will will notice a banded pattern on the surface indicating two objects are overlapping (Figure 5). Drag the green arrow to the left to move one of the objects.

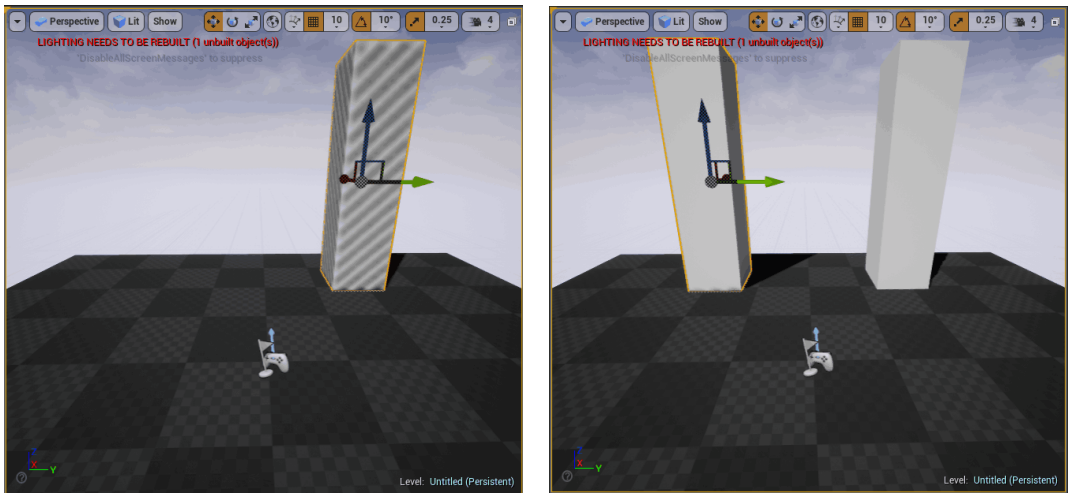
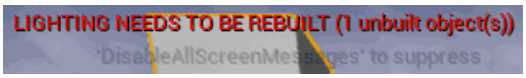


Figure 5: Duplicating the cuboid using copy and paste.

By now you probably have noticed this somewhat annoying message popping up every time you modify a scene object:



The reason you see this is because Unreal Engine precomputes (or bakes) lighting effects (like global illumination and shadows) into static scene geometry using a technique called light mapping. This is a general class of ahead-of-time (AOT) optimization that moves work from the run-time to the build (or compile) stage. During editing it is okay to ignore these messages, but before deploying your creation, you will want to click the *Build* button on the topmost panel to rebuild the light maps.

Press the Play button on the top panel to preview the level. You can rotate by dragging with the mouse and move with the `WASD` keys. Press `Escape` to exit the preview. Let's move the player back away from the cuboids a little. The icon with a flag and gamepad in the Level Editor is the *PlayerStart*. The PlayerStart determines the pose (position and orientation) of the player when the game starts. Move it away from the cuboids by dragging backwards on the red arrow (Figure 6).

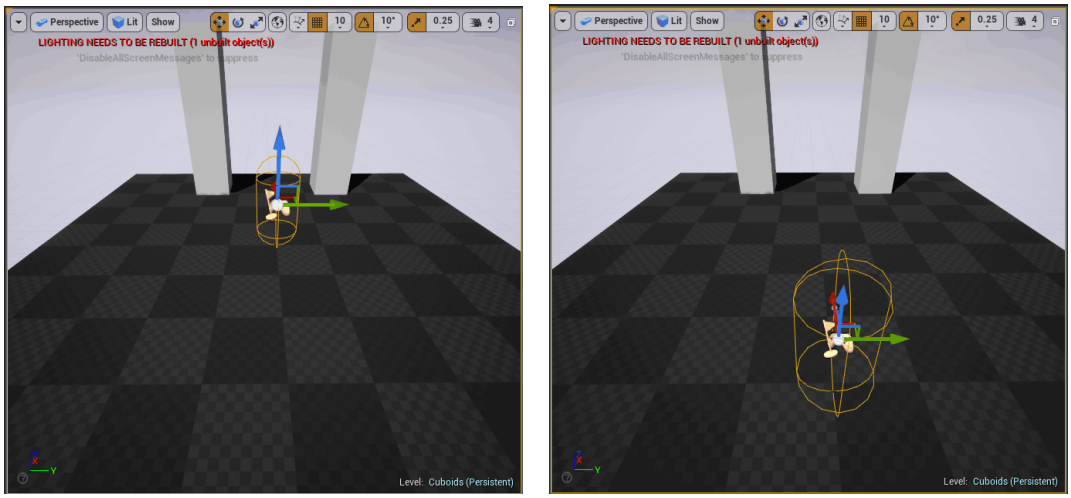
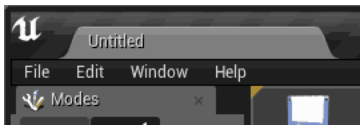


Figure 6: PlayerStart is an actor that determines the starting pose (position and orientation) of the player. Here we are just moving it away from the scene geometry.

At this point, it would be a good idea to save the level we are working on. Typically, a project will have several distinct levels with their own geometry and actors. When you create a new project, you are working on an Untitled level as indicated on the tab in the upper left corner of the editor:



Press `control-s` to bring up the Save Level As dialog box (Figure 7). If you had already saved some levels in your project, they will show up here. Change the Name of the level to "Cuboids" and press the Save button.

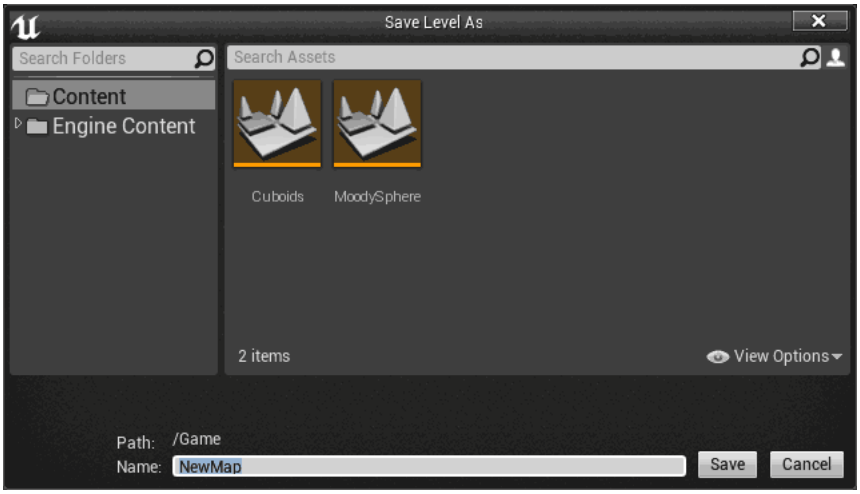


Figure 7: Actor Transform settings in the Details panel.

That concludes the introduction to the Unreal Editor. For a slightly more advanced editor tutorial, see https://docs.unrealengine.com/4.27/en-US/BuildingWorlds/LDQuickStart/.

2 Objects

Unreal Engine, being based on C++, has a strong notion of object-orientation throughout. *Object-oriented programming* (or OOP) is a paradigm that involves the use of "objects" that encapsulate data and functions on that operate on that data [44]. A *class* defines the data and functions that belong to an object. An *object* is a particular instance of a class that has its own state. Some non-programming analogies can help with understanding classes and objects. In architecture, a blueprint is a class and a building is an object. In biology, a genotype is a class and a phenotype is an object. In culinary arts, a recipe is a class and the prepared food is an object. The class-object relation is one-to-many—one class and multiple objects of that class. A key tenet of OOP is *encapsulation*—hiding implementation details behind an interface.

Another key tenet of OOP is inheritance. *Inheritance* allows a class to inherit the data and functions from another class and add its own data and functions. The class being inherited from is called the *base* or *super class* and the class doing the inheriting is called the *derived* or *sub class*. Many OOP systems have an inheritance tree that describes a set of "is-a-type-of" relationships between various classes. Nearly everything you work with in UE is an object of some type of class. Some of the most important classes include `Actor`, `GameModeBase` and `Level`. At the root of the inheritance tree is an abstract class called `Object` that all other classes inherit from. Many of the classes you will be working with are of type `Actor`, a subclass of `Object`. Many classes inherit from `Actor` including `Character`, `GameMode` and `PlayerController`. `Actor` and `ActorComponent` form

a "has-a" relationship. That is, an `Actor` may *have* zero or more `ActorComponent`s, but there is no inheritance going on. Listing 2 shows the inheritance tree of several common UE classes with `Object` at the top (root) of the tree.

```
Object
|
|-- Actor
|   |
|   |-- Pawn
|   |   |
|   |   |-- Character
|   |
|   |-- Info
|   |   |
|   |   |-- GameModeBase
|   |   |
|   |   |-- PlayerState
|   |
|   |-- Controller
|       |
|       |-- PlayerController
|       |
|       |-- AIController
|
|-- ActorComponent
|   |
|   |-- SceneComponent
|       |
|       |-- PrimitiveComponent
|           |
|           |-- ShapeComponent
|           |
|           |-- StaticMeshComponent
|
|-- World
|
|-- Level
```

Listing 1: Inheritance tree of commonly used UE classes

Class	Is a(n)	Description	Use Cases
Object	*n/a*	Base class of all UE4 objects	
Actor	Object	A game object with a transform that can be placed in the world	Audio source, player start
Pawn	Actor	An Actor that can also receive input from a controller or AI	NPC, projectile
Character	Pawn	A Pawn with the ability to walk around (has a mesh, collider and movement logic)	Player
ActorComponent	Object	Reusable behavior that can be shared between Actors	Movement algorithm, lights, sound playback, inventory, visual mesh
SceneComponent	ActorComponent	Has a transform, but no rendering or collision	Camera, spring arm, light, forces
PrimitiveComponent	SceneComponent	SceneComponent with geometric representation	Sprites, collision volume, trigger
ShapeComponent	PrimitiveComponent	PrimitiveComponent that represents a shape	Arrow, Box, Sphere, Capsule, Spline
StaticMeshComponent	PrimitiveComponent	Instances a static mesh—a piece of geometry that consists of a static set of polygons	Level geometry, floor, sky dome, scenery
Info	Actor	Root of all information holding classes, can be replicated	
GameModeBase	Info	Actor that holds game rules (held on server)	Set player character, configure camera
GameStateBase	Info	Holds game state of all players (replicated on server and clients)	
PlayerState	Info	Holds all player state (replicated on server and clients)	
Controller	Actor	Non-physical actors that can possess a Pawn to control its actions	
PlayerController	Controller	Maps user input to a Pawn	
AIController	Controller	Manage AI for Pawns they control	

Table 1: Description of some common UE classes.

3 Scene Graph

Before getting into the more advanced features of the Unreal Engine, we will first familiarize ourselves with the concept of scene graphs. A *scene graph* is a data structure that describes geometric relationships between objects [45, 40, 27]. One of the earliest uses of a scene graph in computer graphics was in the PHIGS (Programmer's Hierarchical Interactive Graphics System) [31] developed in the 1980s. A scene graph allows one to construct complex objects from simpler ones by defining hierarchical relationships between the parts. For example, a car object might be composed of a body and four separate wheels. When we move the car around the scene, we would like all the parts to move as a single unit. The wheels should also be about to turn independently of the rest of the car. A scene graph consists of a tree of *nodes* where each node stores a geometric transformation and, typically, a geometric mesh. The top-level node in a scene graph is called the *root node* and defines the base for all transformations. Under the root node are *child nodes* that inherit all transformations of their parent nodes. Thus, each child node describes a local coordinate system relative to its parent. In the case of the car, the body of the car might be the root node and the four wheels child nodes of the body. Figure 8 shows an abstract object built out of multiple parts and its underlying tree data structure.

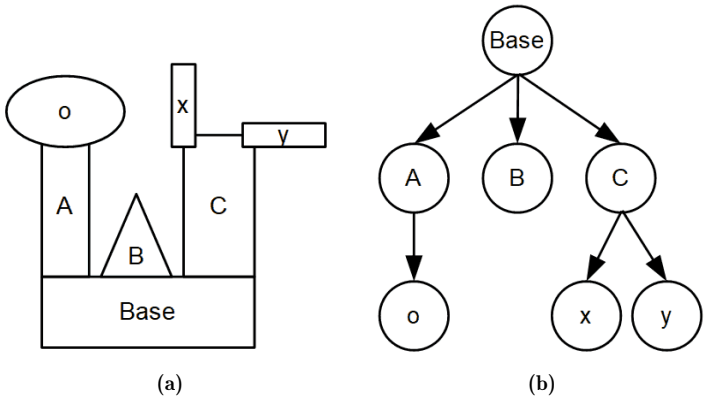
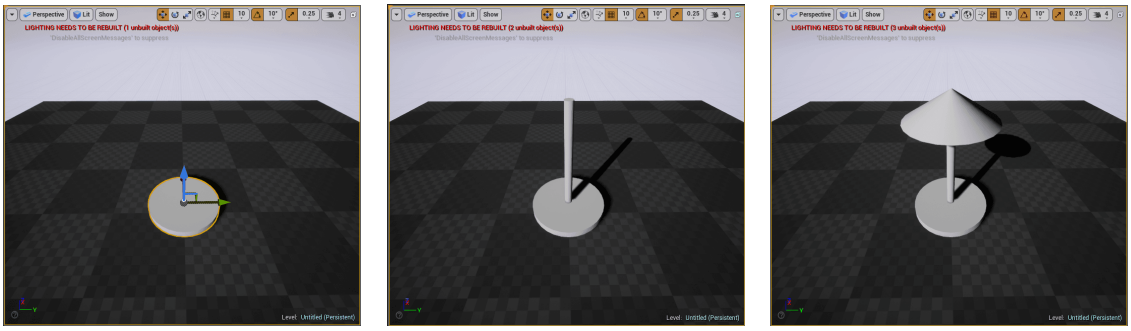


Figure 8: A scene graph as used in a graphics system: (a) a composite object and (b) its underlying scene graph. The Base part acts as the root node. Parts A and C have child nodes that move along with the part. Parts B, o, x and y are leaf nodes.

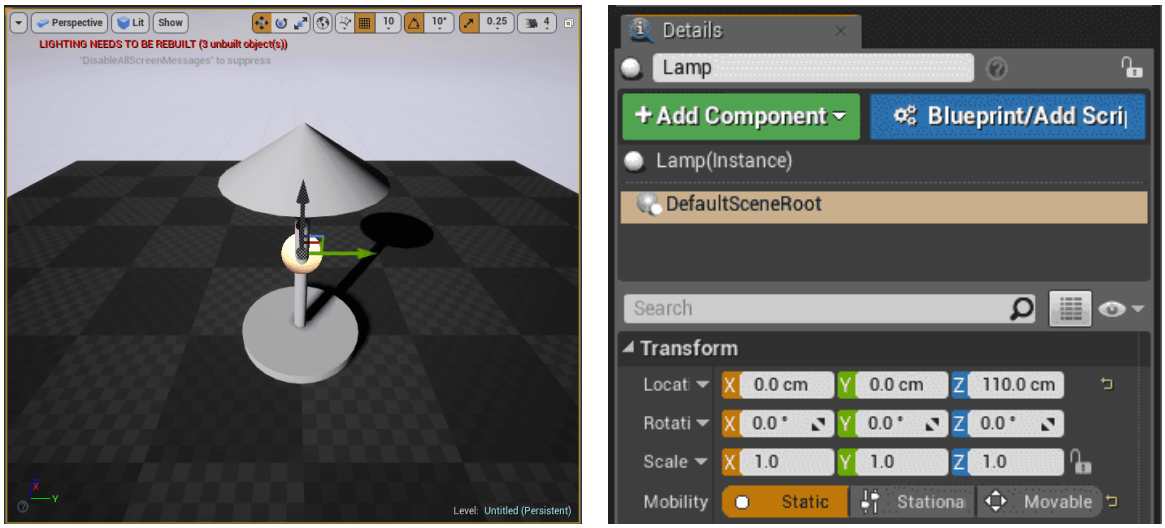
We will create a lamp to learn how scene graphs can help us build more complex scene objects. Begin by creating a new level (after making some edits, you can save it as "Lamp"). Drag a cylinder into the center of the level from the Modes panel. (You may want to move the PlayerStart down so it is out of the way.) This will be the base of the lamp (Figure 9a). Set the cylinder's x,y,z scale to 0.7, 0.7, 0.1, x and y positions to 0 and then move it up so it is just out of the floor. Place another cylinder into the level and change its x and y scale to 0.1. This will be the central post of the lamp (Figure 9b). Move the post vertically so it is just touching the base. Finally, place a cone into the level just above the post. This will be the lamp shade (Figure 9c). Set its z scale to 0.3.



(a) (b) (c)

Figure 9: Constructing a lamp from primitive shapes.

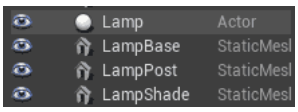
Next, we need to group the shapes together into a scene graph so we can treat the lamp as a single unit. To do this, we will use another actor as the root node of the scene and then add all the lamp parts to the root. Place a new Actor in the level from the Modes panel. Position the actor so it is roughly in the middle of the post (Figure 10a).



(a) (b)

Figure 10: Setting up an actor to serve as the lamp root node in a scene graph.

In the World Outliner rename the recently placed actor to "Lamp" and the lamp shape parts as follows:



Next, drag LampBase, LampPost and LampShade on top of the Lamp actor in the World Outliner to attach them to Lamp. This effectively makes them children of Lamp in a scene graph. But wait... we get this error

The error is because actors placed in the level have a dynamic scene graph by default. The problem is that our lamp geometry is static (cannot be moved at run-time). You cannot mix and

match dynamic and static objects like this. Since a lamp is likely to be static, we will fix the error by making the Lamp actor static as well. Select Lamp in the World Outliner and then under details, select DefaultSceneRoot. Under Transform, a new field Mobility [9] should appear. Select 'Static' so the actor cannot be moved during run-time. Figure 10b shows the correct setting. Now you should be able to attach the lamp geometry to the actor. The children will show up as slightly indented:

As our final task, we will attach a light source to the lamp. Drag a Point Light (light bulb) into the level from the Modes Panel. Position it just under the lamp shade. Once you are happy with the position, attach it as a child to the Lamp actor. We now have a lamp object! The Point Light has several properties like intensity and color that can be customized. To see your lamp in action, try creating a wall and placing the lamp next to it. Remember since we are using static actors, you must press the Build button before playing the level to rebuild the lighting.

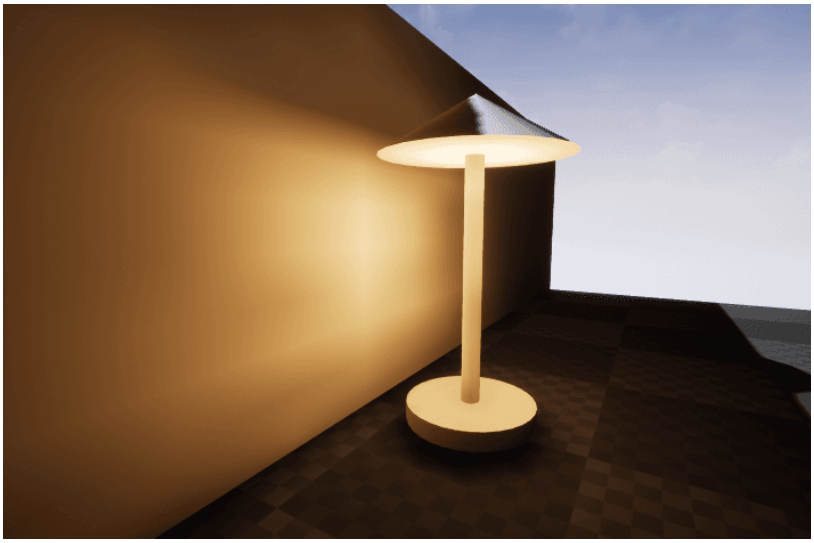


Figure 11: Final lamp object placed in scene.

3.1 Why Not Use a Group?

The astute programmer will probably notice the option to group actors in the scene so that they all transform together. When you create a group a special kind of Actor called a GroupActor is created. This GroupActor in fact acts just like the root node of a scene graph. Why not just use groups then? The answer is that groups are not as flexible as a scene graph as they can only have one level of hierarchy (a root node and children). Groups are handy for simple organization of objects, like level geometry, but for more complex relationships the method outlined above is required.

4 Blueprints

Blueprints [13] is a visual programming interface in the Unreal Editor that is used to specify program logic. Blueprints is based on the notion of a *dataflow* (or *stream processing*) *network* [29, 3, 43] where data is passed between *nodes* that execute a specific function and have a variable number of inlets and outlets. Nodes operate like black boxes in that it is not essential to know how they work, but rather what they do. Mathematical functions and the functions and objects seen in programming languages are also black boxes.

Blueprints has many of the same concepts seen in textual programming languages such as variables, functions, conditionals (if/else) and classes/objects. For beginners, this is both good and bad as one learns proper programming, but there is also a slight learning curve. However, practice and learning the basics step-by-step can help overcome this obstacle.

4.1 Blueprint Editor

Before diving into Blueprints programming, we will first familiarize ourselves with the Blueprints Editor (Figure 12). When you edit a Blueprint, you are in fact editing a Blueprint *class*. The central pane is the *graph editor* where we can place and connect nodes. The leftmost pane contains the *attributes* (variables, functions, etc.) of the Blueprint class. In OOP parlance, these would be called the members of the class. The rightmost pane displays *details* of the currently selected node. The top panel lets us compile the Blueprint, debug it and perform other common Blueprint-wide operations.

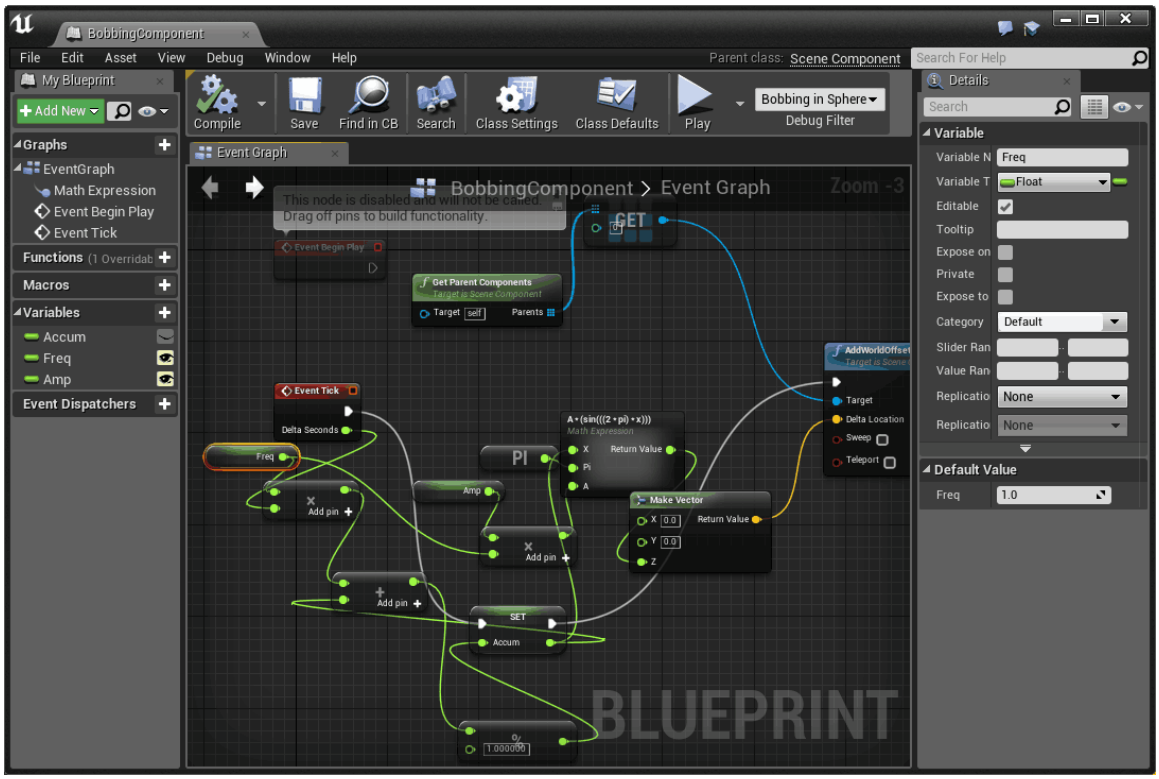


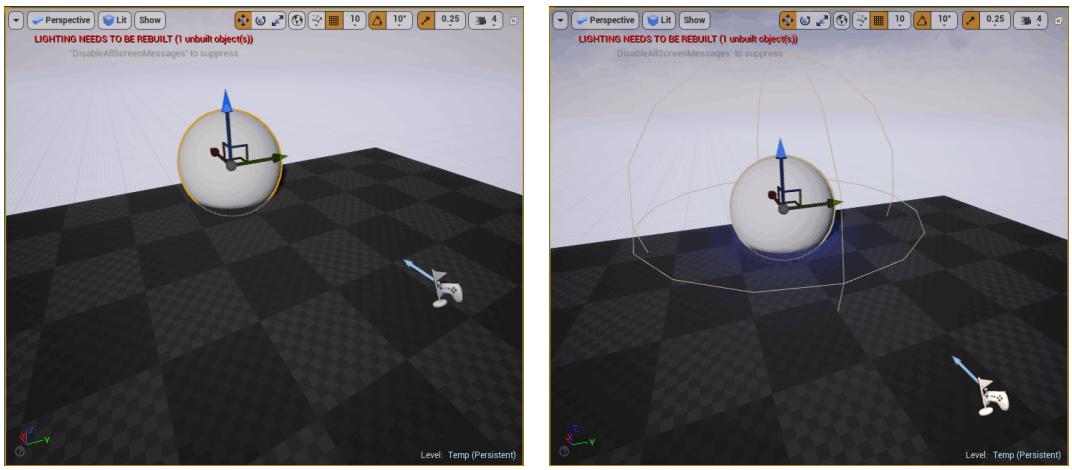
Figure 12: Blueprint editor.

The following sections are step-by-step tutorials to help you learn Blueprints from basic principles. Before moving on, familiarize yourself with how Blueprint nodes [22] work and how to connect them [14]. Most of the tutorials are presented in the spirit of "gray boxing" [39] or "blocking" where we work with only basic, unshaded (gray or white) shapes. With this approach, we can focus on programming and other logic and not get distracted by visual presentation (although simplicity has a beauty in itself).

4.2 Our First Blueprint: A Proximity-Reactive Sphere

As our first foray into Blueprints, we will construct a sphere that lights up when approached by the player. Begin by creating a new level (File > New Level... > Default) and dragging a sphere mesh into the scene from the Modes panel. Move the sphere to the far edge of the plane and scale it by

2 (Figure 13a). Next, add a Point Light to the sphere mesh via "+ Add Component" (UE4) or "+ Add" (UE5) under the Details panel. Change the light color to blue or whatever else you prefer. We want our sphere to react when we are in proximity to it, not necessarily touching it, so we must add a collider. Add a Sphere Collider component to the sphere mesh. By default, the radius of the sphere mesh is 50, but the collider may be something else, like 32 (check under Shape > Sphere Radius). Make the radius of the collider also 50. Now, since the sphere collider is a child of the sphere mesh and its scale is 1, it will hug the boundary of the mesh. We want the collider larger than the mesh so we can get a collision event when near the sphere, not touching it. Change the scale (under Transform) of the sphere collider to 4 so it is 4 times larger than the mesh. If all done correctly, you should see a faint light around the sphere and a large spherical outline indicating the collider (Figure 13b).



(a) (b)

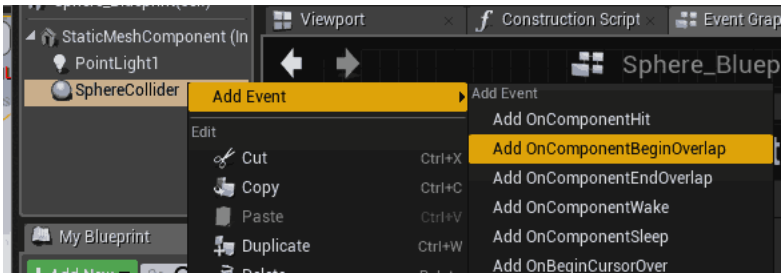
Figure 13: Setting up our lit sphere in the main editor.

We now have our basic scene set up. Next we will use Blueprints to program our interaction with the sphere. Make sure the sphere mesh (actor) is selected and then in its Details panel:

- UE4: Click the big blue button "Blueprint/Add Script" 

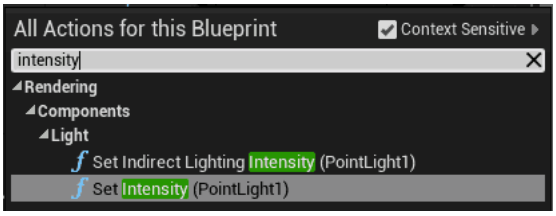
. You will be prompted to select the path where the Blueprint is saved. You can just click "Create Blueprint" and use the defaults.
- UE5: Click the tiny graph icon  next to the Add + button. A window will pop up asking you for a Creation Method. Leave the defaults and click the Select button at the bottom.

Now, the Blueprint editor will come up. You may also notice that your Blueprint shows up in the main editor Content Browser. Blueprints are treated like any other assets in the editor. In the Blueprint editor, click on the "Event Graph" tab and you will see three red nodes that give you access to events common to all objects. You can ignore these as we will not be using them. We need some way to detect when the player overlaps with the collider. In the Components pane, right-click on the sphere collider and select Add Event > Add OnComponentBeginOverlap



Repeat the above, but for `OnComponentEndOverlap`. You should see two red nodes in the editor

These nodes will fire an event from the gray topmost outlet (Exec) when their condition is satisfied. The next step is to make these events turn the light on or off. We will do this by telling the overlap events to set the intensity of the light. Right-click an empty spot on the canvas and type "intensity" in the search. Select the function `Set Intensity`.



We will need two of these, so in the graph editor, right-click on the node you just added and select Duplicate. Arrange your nodes as shown in Figure 14. Ensure that the point light is connected to the Target inlet of both of the Set Intensity nodes. Connect pins by dragging and dropping with the left mouse button. Disconnect wires by alt-clicking on the wire or either of the pins. Enter the value 5000 into the top Set Intensity node and 0 into the bottom one. Lastly, connect the `Exec` outlets of the overlap events to each of their respective `Set Intensity` inlets. Doing so causes the intensity value to be set when those respective events fire.

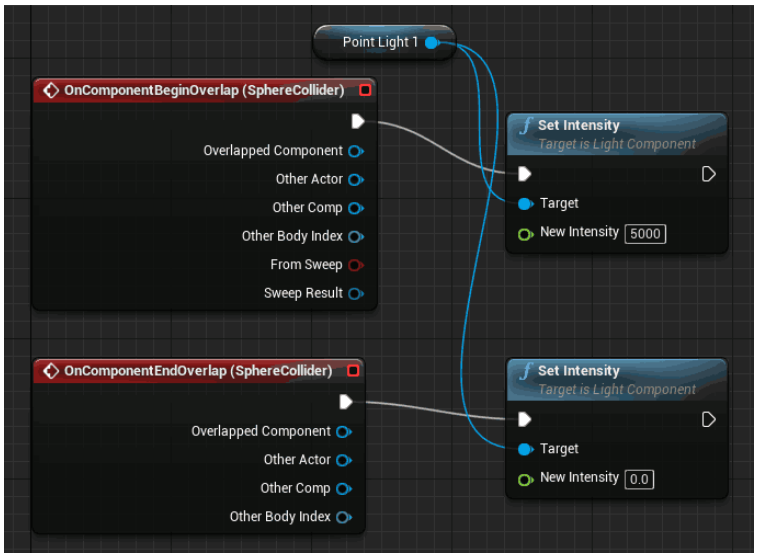


Figure 14: Our final Blueprint graph of a proximity-reactive sphere.

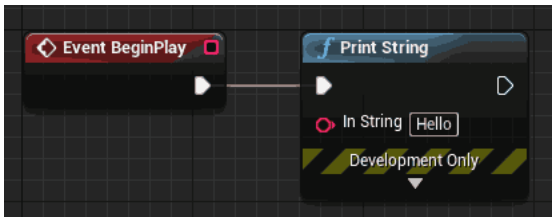
Press Play in the Blueprint editor to test out your program. Move towards and away (using `W` and `S` keys) from the sphere in the main editor viewport and ensure that the light behaves correctly. It should turn on when you are close and turn off when you are far away. You may notice that the light is on even when you are far away when you first start. This is due to the unfortunate fact that the `OnComponentEndOverlap` event does not fire on startup (when perhaps maybe it should). It will only You can resolve this by setting the default light intensity to 0 or by connecting the `Exec` outlet of the `BeginPlay` event to the `Set Intensity` node having value 0.

4.2.1 Challenge

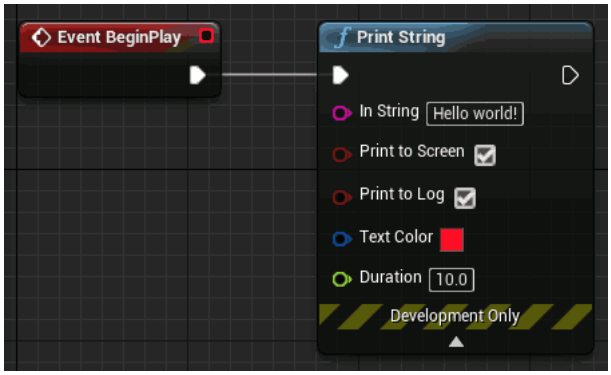
Can you get the light to turn on from an arbitrary actor you add to the scene? The player has certain attributes enabled by default that will trigger overlap events. A generic Actor does not. Hint: Check for meaningful fields under the Collision category. There will be two flags you need to enable.

4.3 Debugging

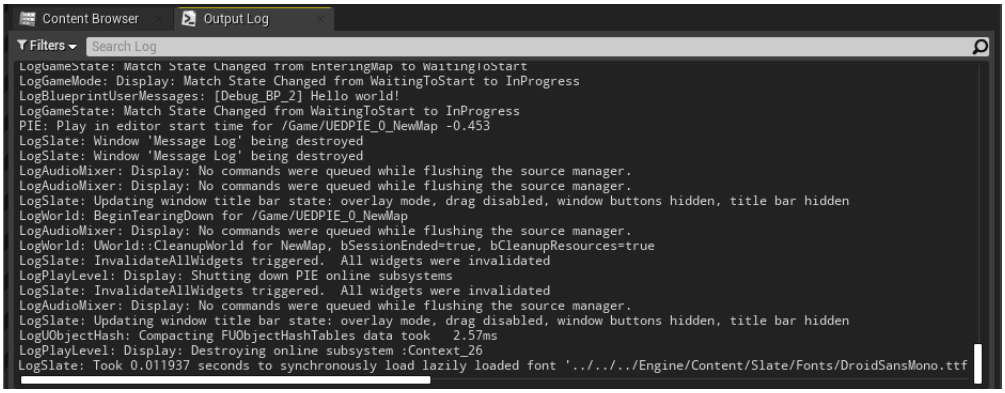
No matter how experienced you are with programming, at some point you will need to debug (find errors in) your code. A handy way to do this in Blueprints is via the `Print String` node. To try this out, create a new Actor Blueprint and drag an instance into the scene. Go to the Blueprint graph and add a `Print String` node. Connect the `BeginPlay` event to the `Exec` input of `Print String`.



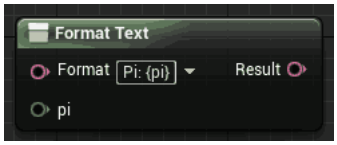
You can change the text in the `In String` inlet or leave it as is. Now press Play and you should see your message show up in the upper-left corner of the window. If you click the down arrow at the bottom of the `Print String` node, you can see other inlets. Try changing the duration of the message to 20 seconds and its color to red.



You will also notice the option `Print to Log`. This refers to the editor-wide Output Log. You can show the Output Log by going to Window > Developer Tools > Output Log. The Output Log contains a history of messages from many other subsystems, but if you search you should find the message from your Blueprint prefixed with something like LogBlueprintUserMessages: [NameOfYourBluePrint_2] where the number at the end is the Blueprint instance number.



For more advanced printing, there is a `Format Text` node that allows you to plug variables into a string. To create a string variable, enclose the variable name in curly braces `{}`. After doing so, an inlet should appear with that variable name.



You will notice the inlet is gray and if you hover over it with the mouse, it will tell you it's something called a 'wildcard'. A wildcard is a variable that recognizes its type once something is plugged into it. Create a `Get Pi` node and plug it into the `pi` variable inlet. You will see the inlet change color to green indicating that it recognizes the input as a float.

You can have as many string variables as you like in one format string. Each one will create an inlet. The result of `Format Text` is a (localized) string that can be plugged into `Print String`.

4.4 Classes: Building a Mood Light

Our next exercise will be to create a mood light with a smooth color animation. In the process, we will learn how classes and objects work in Blueprints and also how to create math expressions and expose parameters to control our light in the main editor. Create a new level and add a point light to the scene. You may also want to create a wall to the side of the light to better see it radiate (Figure 15).

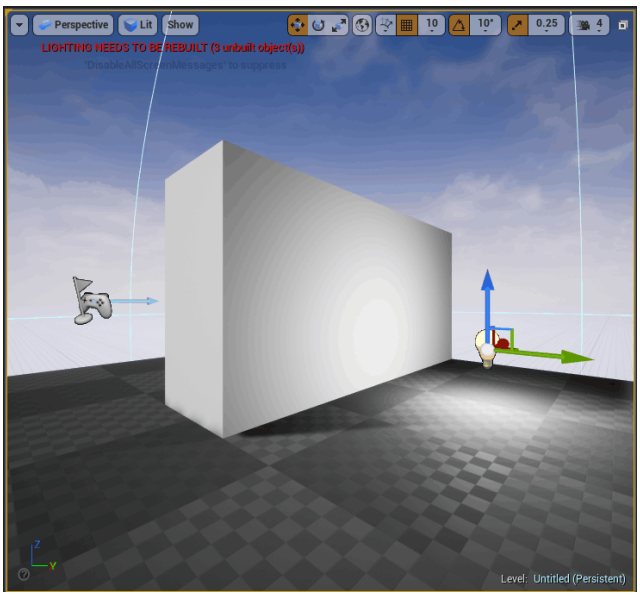
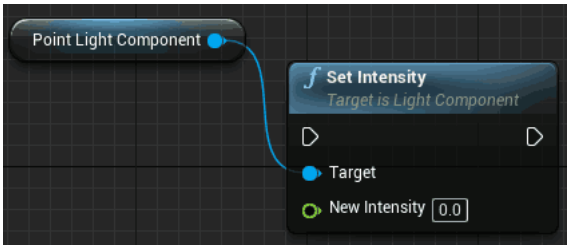
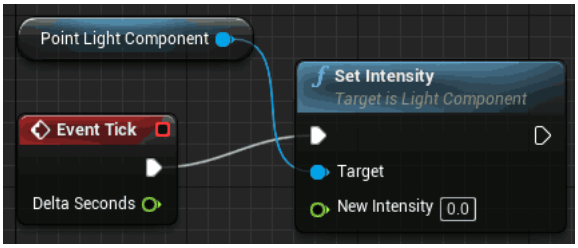


Figure 15: The mood light scene.

The next step is to convert the point light into a Blueprint class that we can add custom animation to. Note that there are myriad ways to create Blueprint classes [15]—we will only cover one method here. Select the light and then in its Details panel, create a new Blueprint for it (see instructions above in 4.2). Once in the Blueprint editor, go to the Event Graph. Right-click on the canvas, type 'intensity' and select `Set Intensity (PointLightComponent)`. You should see a `Set Intensity` node with the `Point Light Component` attached to its `Target`.

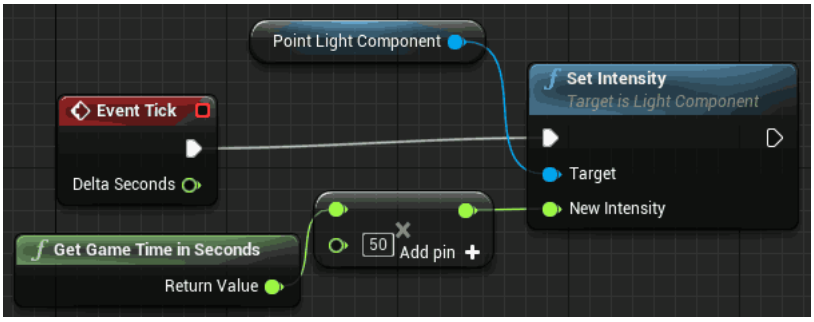


The `Target` inlet specifies which object the `Set Intensity` function will operate on. In OOP, the `Point Light Component` is the object and `Set Intensity` is a member function that acts on the object. If we press Play at this point, nothing will actually happen. Why? Because we are not actually calling the function. That is what the gray `Exec` inlet at the top is for. If we want to change the intensity of the light, we must connect the Tick event Exec outlet to its inlet.

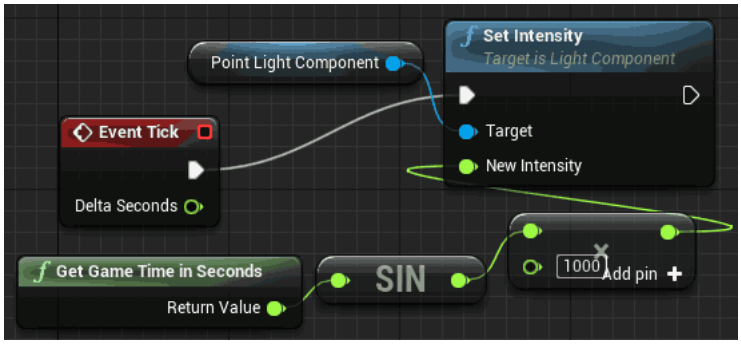


Pressing Play now will turn off the light, as expected. Not that interesting, but it's progress.

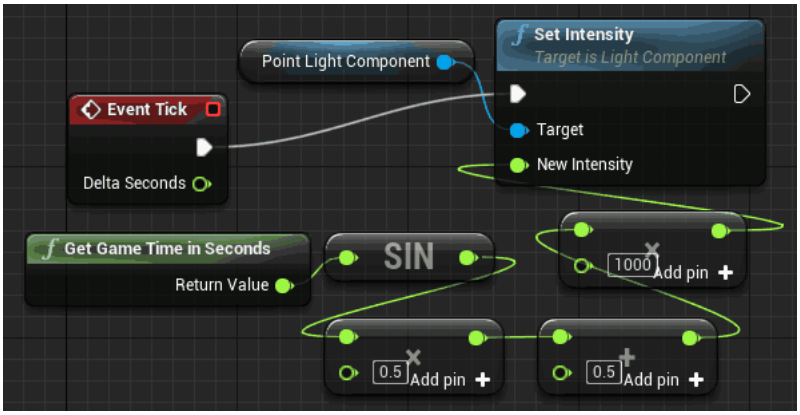
Our next task will be to animate the intensity of the light. To do any kind of animation, we need a time variable. We will use the *game time* which is the number of seconds since the game started and can also be paused and slowed-down. This is not the best way to do animation, but is simple and will suffice for now (more on this later). To get the game time, add a `Get Game Time in Seconds` node. We could connect this to the light intensity, but since the intensity is in lumens, we will have to wait a while to see any effect. Add a `float * float` function to scale the time value by 50 and connect this instead to the light intensity.



Press Play and you should see the light slowly fade in. Great! However, we want the intensity to have a more interesting oscillation pattern. Try inserting a `Sin (Radians)` function between the time and the multiplier.



The light will now oscillate, however, you might notice it is dark half the time. This is because sine has a signed output range so we are actually feeding negative intensity values to the light which it just treats as zero. One way to fix this is to go MADD! MADD stands for multiply-add and is a common strategy for mapping one range onto another. Since sine has an output range of $[-1, 1]$ and we would like it to be $[0, 1]$, we use the MADD (AKA linear) mapping $f(x) = \frac{1}{2}x + \frac{1}{2}$. Here is our improved mapping that oscillates in the unsigned range $[0, 1000]$ lumens.

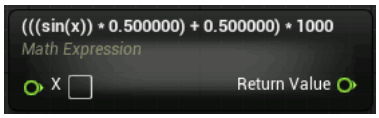


Already, we can see our graph is getting quite complex, so let us try to make it more compact and easier to maintain. Since most of the Blueprint is just doing simple math, we will replace those nodes with a single node having a unified expression. This can be accomplished with a `Math Expression` node [21]. Math expressions are a very powerful feature of Blueprints and should be your go-to solution for anything involving non-trivial math. The behave very much like *functions* in programming languages, albeit math-only. Proceed to add a `Math Expression` node to the canvas. In the name field of the node it will say UnexpectedTokenType

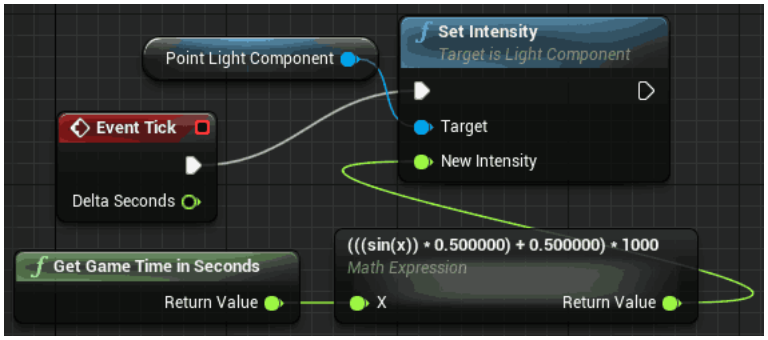
This is normal. It is just indicating, in a somewhat unfriendly manner, that no expression is defined yet. (A *token* is a keyword recognized by a compiler or interpreter.) Enter the following expression in the name field:

```
(sin(x) * 0.5 + 0.5) * 1000
```

The node should change to the following

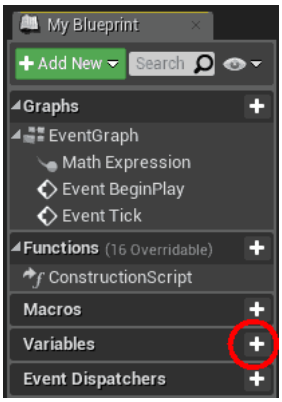


The editor may auto-format the expression, but it should still be equivalent to what you entered. (Tip: If you find the auto-formatting annoying you can create a comment in the Blueprint by pressing the 'c' key and editing your equation there and pasting it into the node when completed. This also lets you experiment more easily with other equations as we will get to below.) Notice that expression node detected the variable `x` in the expression and automatically created an inlet for it. You can now replace the four math nodes we had previously with this single node

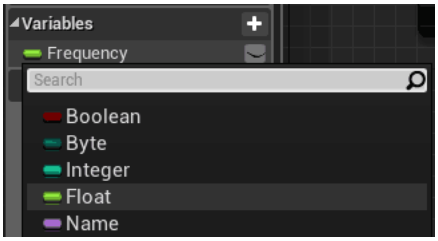


That is much cleaner! Play the scene to ensure the expression is working correctly. In general, it is good programming practice to take small steps and test your code after each iteration. This makes it easier to spot bugs in your code since you can identify and address them as soon as they arise.

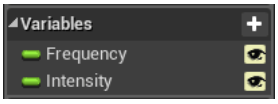
Our last task will be to parameterize the Blueprint so we can control the effect from the main editor. In OOP, we would call this creating an *interface* to the object. Remember the key idea of encapsulation is to hide and lock-away implementation details of an object from the user. We will add two parameters: frequency and intensity. Start by going to the My Blueprint panel on the left side and click the `+` button under Variables.



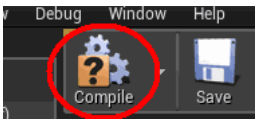
Name it "Frequency" and change its type to `Float` (green).

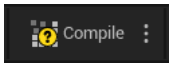


Create another variable called "Intensity" and make its type also `Float`. Notice the closed eye (yes, that's what it is!) to the right of each variable. This indicates that the variables are visible only within the Blueprint and cannot be seen from the main editor. In OOP, these are known as *private* members (variables or functions)—members only accessible from within the class. We want our variables visible outside the Blueprint, so click the closed eye to make them visible.

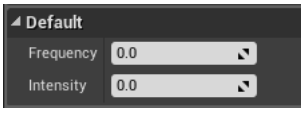


The variables are now public and thus can be accessed from the main editor. In OOP, a *public* member is a member that can be accessed outside the class through an object (instance of the class). Before our changes take effect, we must recompile the Blueprint. Click the Compile button located on the left side of the top panel. You should see a question mark over it indicating a recompile is needed.



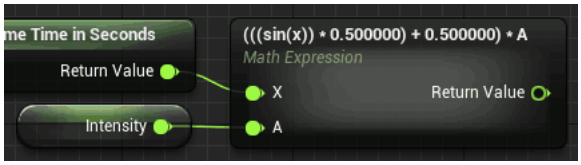
or


After recompiling, the variables should show up in the main editor in the Blueprint actor's Details pane.



The variables will be under a default category called "Default". You can change this if you like from the Blueprint editor. Next, we need to actually use our variables! Drag and drop the `Intensity` variable onto the Blueprint canvas. A small window will pop-up with a selection between 'Get' or 'Set'. Choose 'Get' and you should then see a new node created with a single outlet.

We now have a *getter* node that gives us the variable's current value. We would like to use this in our math expression instead of the hard-coded value 1000. To do so, replace the 1000 in the expression with the variable name "A" (for Amplitude). After making the replacement, you should see an extra inlet created on the math expression node with the variable name. Input variables to functions are often called *arguments*. Connect the `Intensity` member variable to this new inlet.

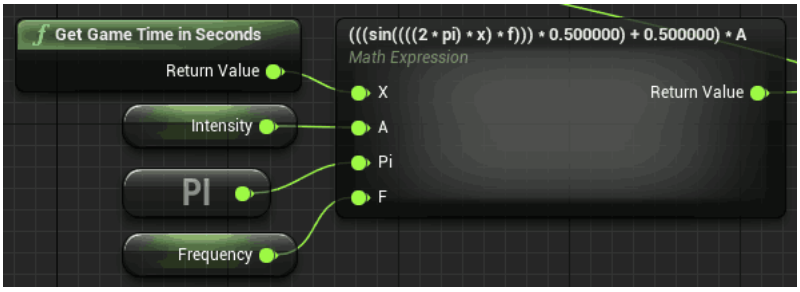


Note that the two variables *do not have to have the same name*. What happens under the hood is that our member variable gets *passed by value* (copied) into the math expression variable and it is the copy that gets used in the expression. You will notice that Intensity variable has a value of zero by default. This is standard for any newly created variables. Change its default value in its Details pane under Default Value to 1000. Recompile the Blueprint and test that the Intensity parameter works in the main editor.

We will follow a similar process to hook up our Frequency parameter. Change the math expression to

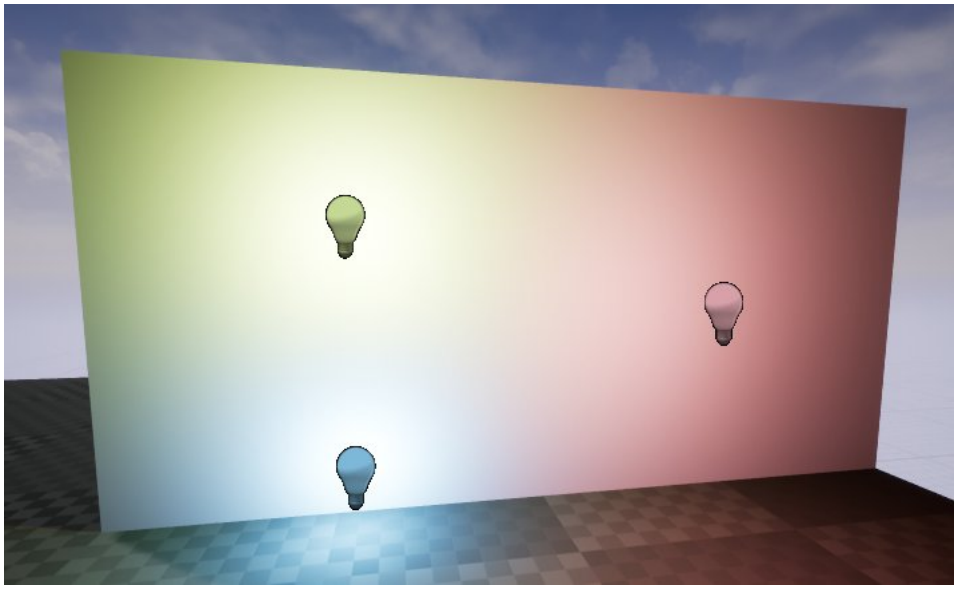
```
(sin(2*pi*x*f) * 0.5 + 0.5) * A
```

and connect the newly created inlets.



The argument to `sin` is an angle in $[0, 2\pi]$ radians. We pass the expression `2*pi*x*f` into `sin` so that the argument `f` is a value in Hz (cycles/seconds). Unfortunately, the math constant π ($= 3.14159\ldots$) is not available directly in the expression editor, so we pass it in as an argument using a `Get Pi` node. Recompile the Blueprint and test the `Frequency` parameter in the main editor.

As a final task, we will place multiple instances (objects) of our Blueprint into the scene. Fortunately, this is as simple as dragging-and-dropping the Blueprint in the Content Browser into the scene editor window. This effectively instantiates the Blueprint class, i.e., creates a new object. Create a few instances with different positions, colors and frequencies against a wall or other backdrop, press Play and watch them animate.



If you are curious, you can try changing `2*pi` to `8*pi` in the math expression, recompile the Blueprint, press Play and see all lights oscillating four times faster. Remember that all instances (objects) are affected by changes to their Blueprint class.

To create a more interesting pattern, you can try raising the output of the raised sine function by some power. For example, try using this math expression to cause the lights to flash momentary

```
pow(sin(2*pi*x*f) * 0.5 + 0.5, 8.) * A
```

You can also experiment with sums of sines with different frequencies to create a variety of patterns, for example

```
((0.8*sin(2*pi*x*f) + 0.2*sin(16*pi*x*f)) * 0.5 + 0.5) * A
```

```
((0.5*sin(0.9*2*pi*x*f) + 0.5*sin(1.1*2*pi*x*f)) * 0.5 + 0.5) * A
```

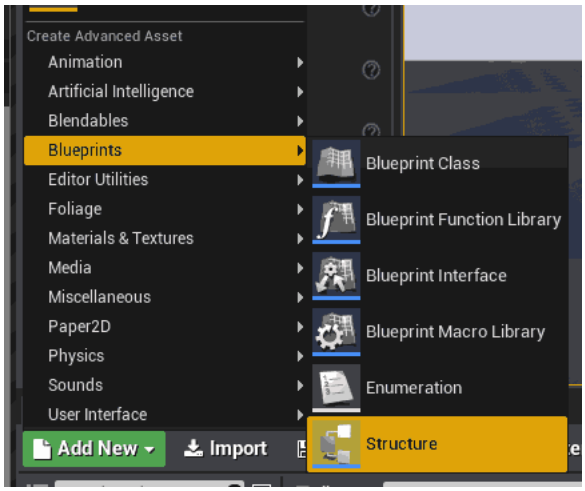
A Small Problem

In order to do animation for the light, we used the game time which counts up in seconds since the game was started. This works fine for quick prototyping, however, it is generally considered bad practice. The reason being that eventually the time variable will grow large and we will lose precision in the least significant bits. Every power-of-two increase in value loses one bit due to the way floating point numbers are represented [28, 46]. For example, counting from 0.5 to 60 seconds loses around 7 bits of precision from beginning to end. This becomes worse when we multiply the time by values larger than 1, as when mapping through other functions. A better solution is a free running phase accumulator that keeps its phase value bounded in a fixed range. This is covered in the next section.

4.5 Structs and Macros: Creating a Reusable Accumulator

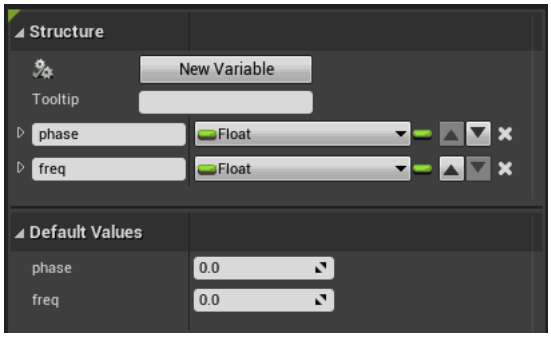
In the previous section, we employed a periodic phase accumulator to drive some animation. While the accumulator logic is simple, it is not exactly trivial and we do not want to copy around it's graph every time we need a new one. In order to create a reusable phase accumulator, we will use two new concepts: structs [25] and macros [16]. The struct will store the accumulator state, phase and frequency, and the macro will do the math to update the accumulator. (This explicit separation between data and function is the same paradigm used by C and other "structured" programming

languages.) First we will create a new type of struct called `Accum`. In the Content Browser, click Add New > Blueprints > Structure.



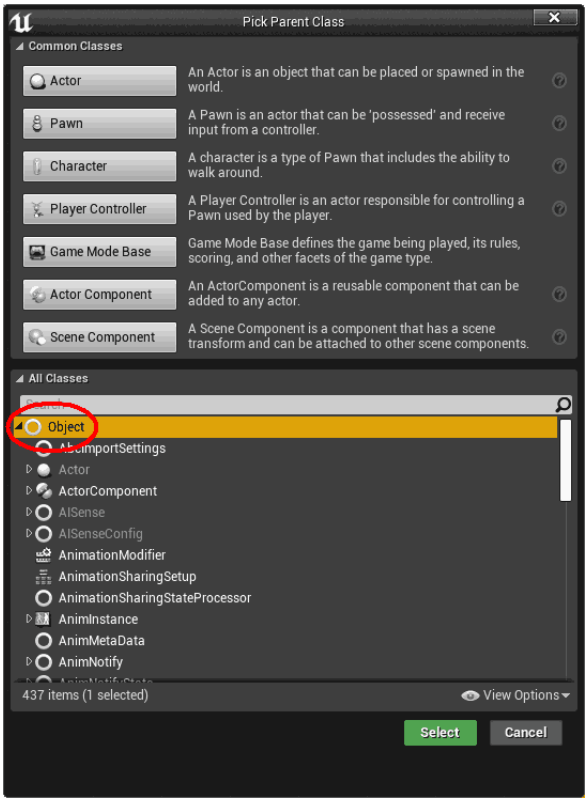
You will see an icon for a new struct asset appear in the Content Browser:  . Name the struct "Accum".

Double-click its icon in order to edit it. In this panel, you can add new member variables to the struct. There will be one member variable there already. Rename it to "phase" and set its type to `Float`. Click New Variable and create a second `Float` variable called "freq".



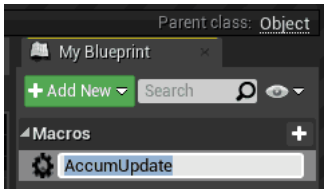
You will notice that you can assign default values to each variable. Leave them both at 0.0. Save the struct when you are done.

Next, we will create something called a Macro Library [12] that will store a macro that updates the `Accum` struct. In the Content Browser, click Add New > Blueprints > Blueprint Macro Library. You will then be asked what parent class to use. Make sure to select 'Object' so that the macros can be used with any object.



The rule is that the macros in a Macro Library will only appear in classes inheriting from the same parent. By choosing "Object" as the parent class, we ensure that the macros in our Macro Library will work with any Blueprint class. After selecting the parent, you should see a new asset with this icon in your Content Browser:  . Double-click the newly created Macro Library asset and you will be presented with a graph editor similar to other Blueprints. However, in this editor you may only create and edit macros. By default, a single, empty macro is created for you. There are two nodes present: Inputs and Outputs.

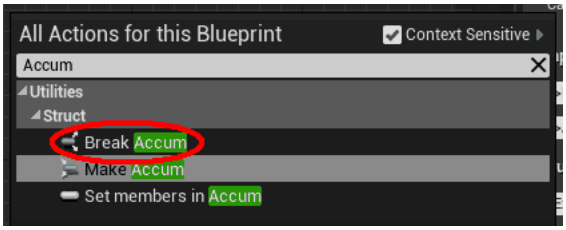
These determine which inlets and outlets are present to the user of the macro. A macro will appear like any other Blueprint node. In fact, it is simply a "collapsed" set of nodes, but one that can be reused. Begin by renaming the macro to "AccumUpdate" in the top-right pane.



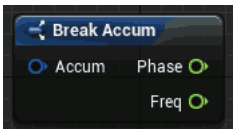
Next, we will add an `Exec` inlet and outlet. In the macro Detail pane in the lower-right, go to Inputs and press the + icon to add a new inlet.

Ensure that the pin type is Exec and name it ">Exec". Create an Exec outlet and name it "Exec>". When finished, your graph should look like this

Now we will create a special node to access the members of an `Accum`. All structs have a respective `Break` node that allow us to *get* their member variables. Right-click the canvas, type "Accum" and then select `Break Accum`.



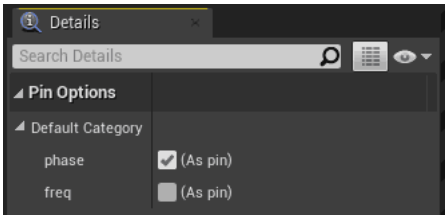
This will create a node with an `Accum` inlet on the left and all of its member variables on the right.



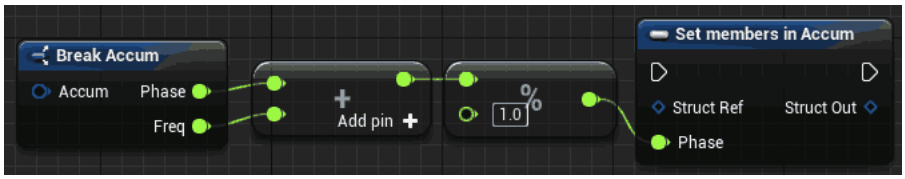
Next, we do the math that updates the phase of the accumulator. Connect the following math nodes to sum the phase and freq variables and then modulo the sum by 1.

Since a `Break` node is read-only (a getter), we need a way to write our new phase value back into the struct. For this, we use another type of node called `Set members in`. Right-click a blank part of the canvas, type "Accum" and this time select `Set members in Accum`.

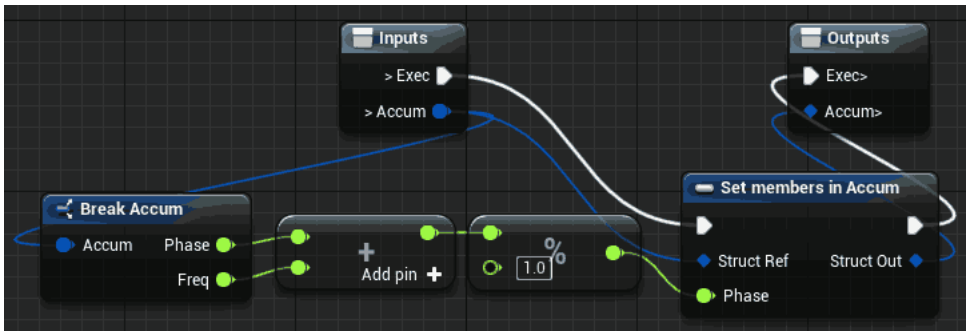
You will see a new node with a struct ref inlet and struct out outlet. As you can see, there are no inlets exposed for the member variables when you create a new `Set members in` node. We need to expose an inlet for the phase variable so we can *set* it. Click the `Set members in Accum` node and go over to its Details pane. Under Pin Options > Default Category you should see some check boxes where you can expose pins (inlets). Check the box next to 'phase' to expose an inlet for it.



Now connect the output of the modulo operation to this new phase inlet.



Now we need to feed the `Accum` struct to the inlets and outlets of the macro so they are exposed to the outside user and ensure the `Accum` setter gets executed (notice the `Exec` pins on it). The graph should look something like the following:

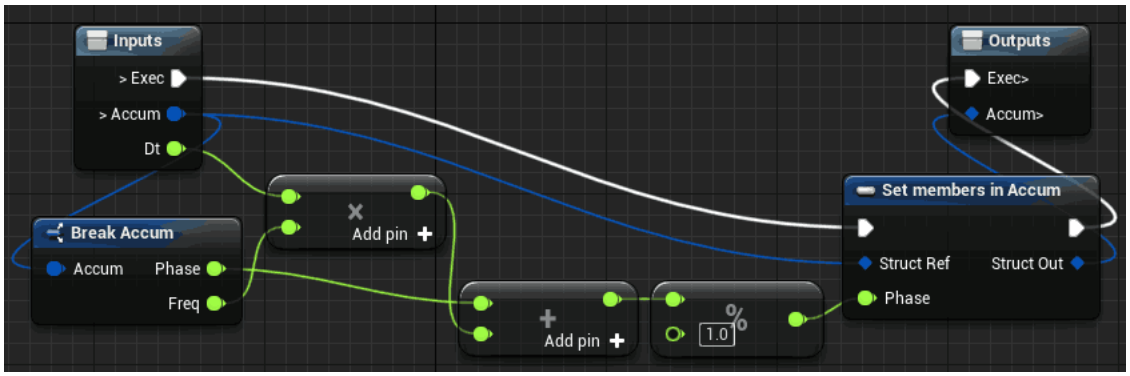


Note that you can drag from the struct pins in the `Set members in Accum` node to their respective `Inputs` or `Outputs` nodes to automatically creates pins for them. You should see 

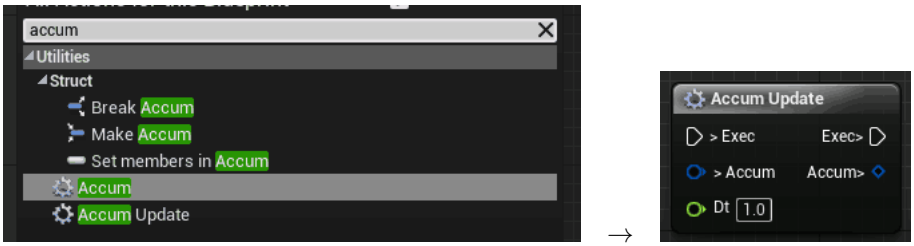
when hovering over the destination node. This is faster than adding them in the Details pane, although you will want to rename them there.

As a final step, we will find it helpful down the road to be able to pass a delta time value into the phase update. This way, the `freq` variable can have more meaningful units, like Hz. Add another `float` input called "dt" to the macro and set its default value to 1.0.

Insert a `float * float` node to multiply `freq` by `dt` before adding to `phase`. The final graph for the macro should look something like the following:



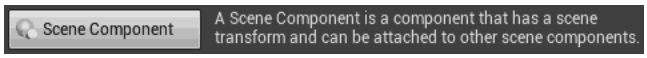
To use the macro, open another Blueprint, type "Accum" and then select `Accum Update`.



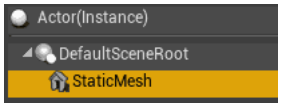
You should see a node appear with the inlets and outlets we defined in the macro. Congratulations, you now have an extremely versatile construct for performing all kinds of procedural animation! We will use `Accum` in subsequent sections to do animation.

4.6 Actor Component: Bobbing Shape

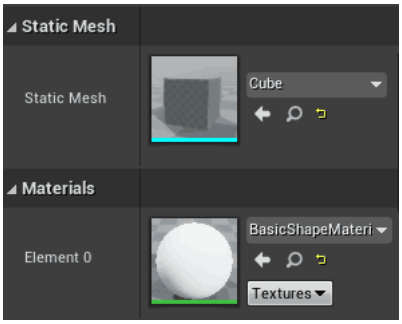
As our next exercise, we will create an Actor Component that can be added to any scene object to cause it to bob (oscillate) up and down. An Actor Component is a reusable behavior that can be added to any Actor. (In OOP speak, we would say that an Actor may *have an* Actor Component.) We will work with a subclass of Actor Component called Scene Component. A Scene Component *is an* Actor Component with a geometric transform (translation, rotation, scale). Begin by creating a new Blueprint class and select Scene Component as the parent class.



Rename the Blueprint "Bobbing_BP". Next, we will create a new level with an Actor to test out the component as we build it. Create a new level called "Bobbing" and add an Empty Actor to the scene. Next, we will add a cube mesh to the actor. With Bobbing_BP selected, click `+ Add Component` and select StaticMesh. This will add the mesh component as a child of the actor.

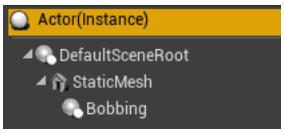


Ensure that the cube mesh mobility is set to Moveable. You will notice that StaticMesh does not have a mesh or material assigned to it by default. Set its mesh to Cube and its material to BasicShapeMaterial.

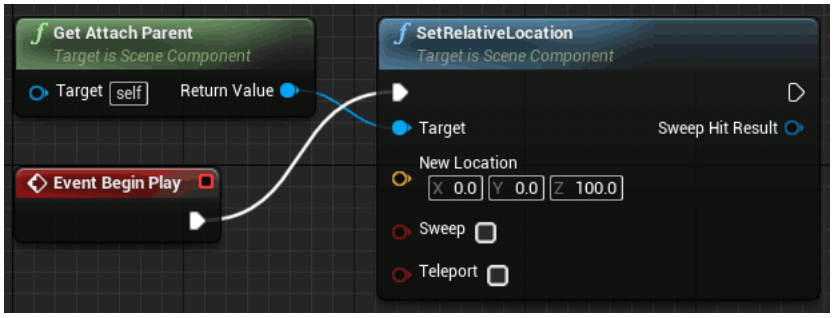


Make sure the actor is selected and then move it so the cube is out of the floor and in front of the camera. It is important to only move the actor and not its components since we will be setting the component locations in the Blueprint.

Note that we cannot simply drag our Bobbing component into the scene since it must be attached to an Actor or Actor Component. With the Static Mesh selected, click `+ Add Component` and then type in "bobbing" and select Bobbing_BP. This will attach the component as a child of the mesh. The actor scene graph should look like this:

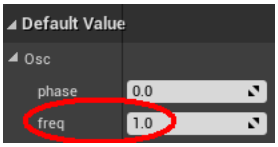


As an initial programming goal, let us just get the cube to change position. Open the Bobbing_BP Blueprint and then add a `SetRelativeLocation` node to the graph. Under `New Location`, change the `Z` value to 100. Create a `Get Attach Parent` node and connect it to the target of `SetRelativeLocation`. What we want to do is set the relative location of the parent of the bobbing component (the mesh). Connect `Event Begin Play` to `SetRelativeLocation`.

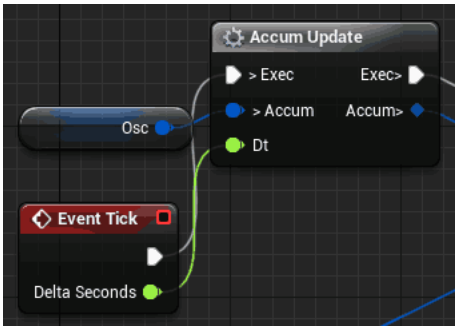


Compile and press play. If everything is correct you should see the cube move up 100 units from its current position. The reason we use relative location (rather than world location) is because we only want to offset the position of the cube, not set its absolute position. The world space position of the cube is determined by the actor's default scene root.

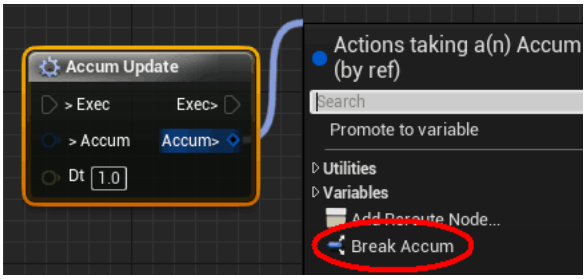
Now we will create the code to make the cube oscillate. Remember the `Accum` struct we created? We will use it to create an oscillator that we can plug into the relative location Create a new member variable named "Osc" and set its type to `Accum`. In the Details pane, set the default value of `freq` to 1.0 (you may have to compile the Blueprint first to see the defaults).



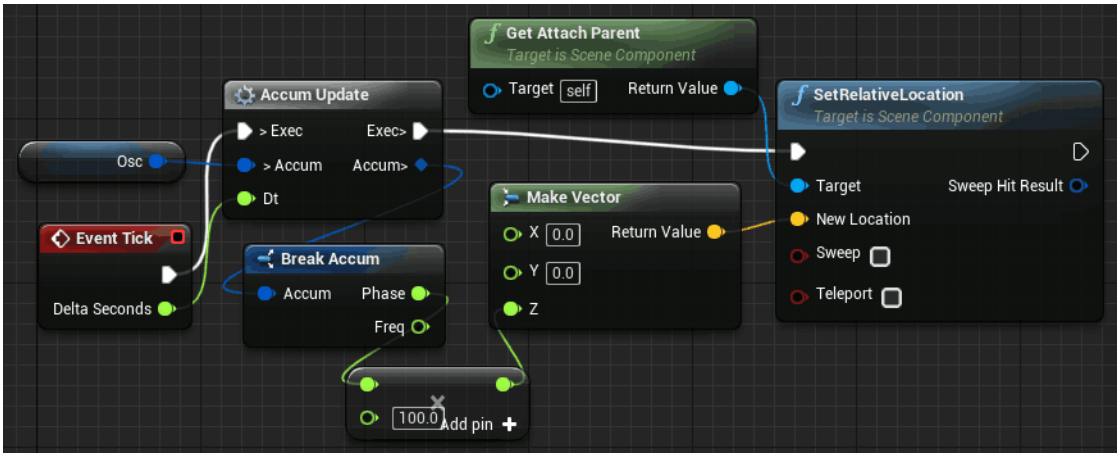
Create an `Accum Update` and a `Get Osc` node. Connect them as follows to `Event Tick`:



This is the "boilerplate" we need to update the phase of our oscillator. Next, drag off the blue (object) pin outlet of `Accum Update` and select `Break Accum`.



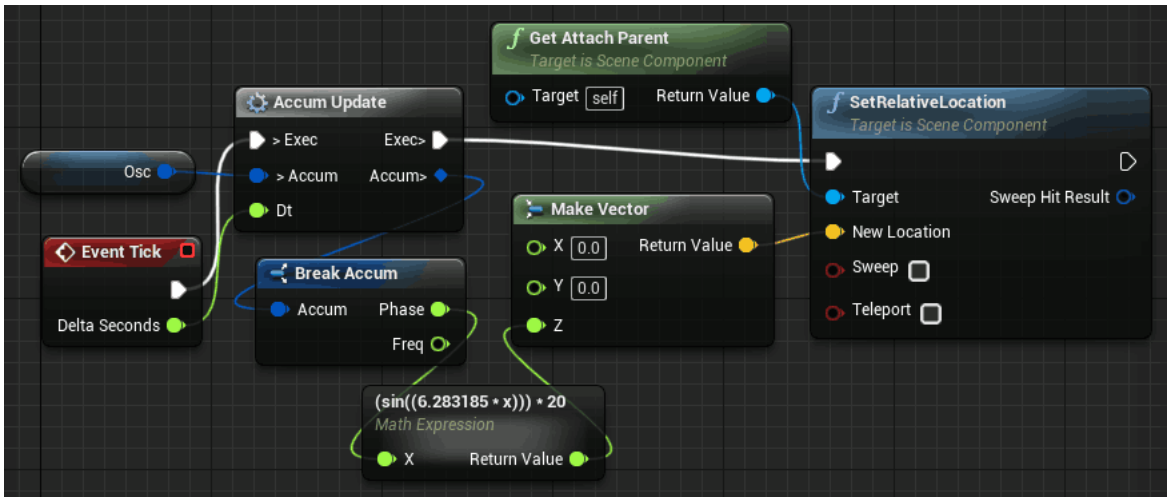
`Break` nodes are like a getter for all members of a struct and are created automatically by the editor for all structs. To test the oscillator phase, we will hook it into the `SetRelativeLocation` node. Since the range of our `Accum` phase is $[0, 1]$ it may be hard to see the cube moving without some amplification. Multiply the phase by 100 and then feed it into the `z` value of a `Make Vector` node. Connect `Make Vector` to `SetRelativeLocation`.



The output of the oscillator is currently a periodic ramp function which causes a very abrupt movement at the end of its cycle. We will map the periodic ramp through a sine function to get a smoother movement. Create a `Math Expression` node and enter the following:

```
sin(x * 6.283185307)
```

Replace the multiplier with this expression.



That's it! We have a bobbing cube. The Bobbing component may be set as the parent of any other scene component to make it animate. Since the output of sine is a signed value, you may want to move the cube mesh up in the editor so it does not pass through the floor.

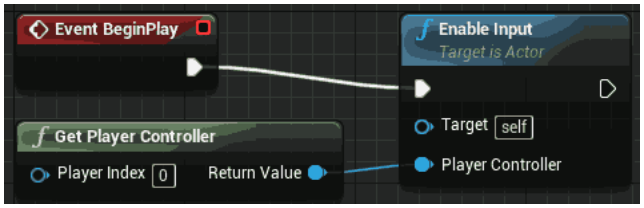
As you might gather, there are many possibilities for animation with this basic setup, for example

- Sway side-to-side
- Move in a circle
- Spin (rotate) around an axis
- Compound motion of any of the above

5 Input Events

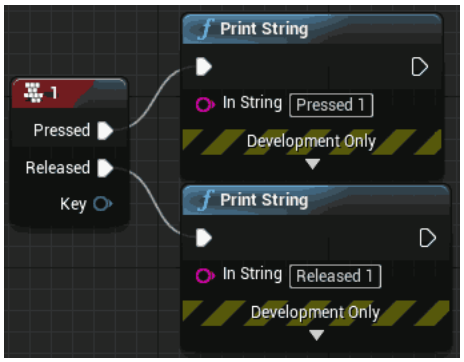
5.1 Keyboard and Mouse Events

In this section, we go over how to get user input (keyboard and mouse) events in a Blueprint. Fortunately, working with user input events is straightforward, however, we must first enable input on the Player Controller. To learn how this is done, create a new Blueprint Actor and connect up an `Enable Input` node as follows:



We can now receive user input events. You can just leave the Player Index at 0, which is the first player. Next, we see how to read keyboard events. Right-click the canvas and type "keyboard 1" and select the node with a 1.

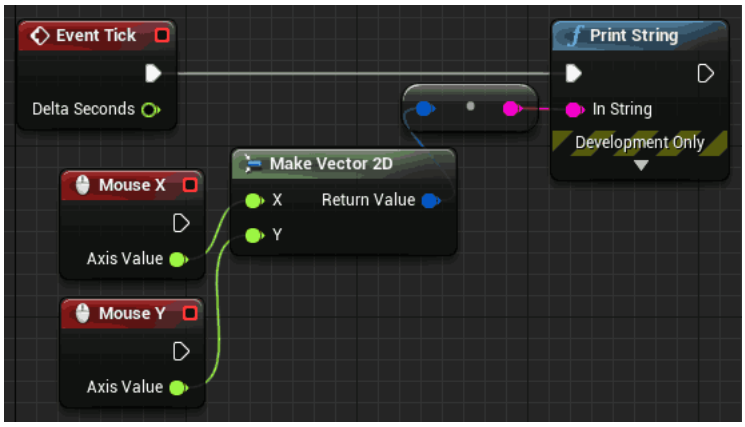
A keyboard event for key 1 is created and you will notice `Exec` outlets for when the key is pressed and released. The pressed and released events fire *once* the moment they occur. Try connecting the key events to a `Print String` as follows:



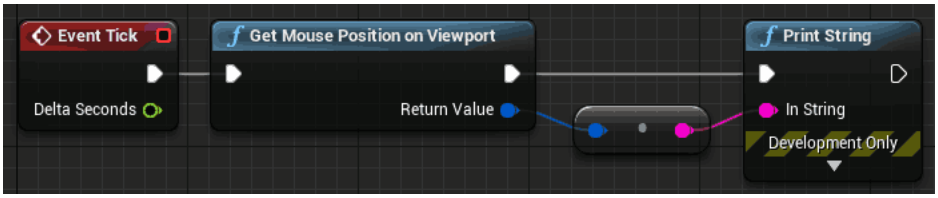
Press Play, click the window and press and release the 1 key to see the messages appear.

Mouse events may be read using a similar process. If you right-click the canvas and type "mouse events" you can see what is available to us. Mouse button events work just like keyboard key events—after all, they are all just buttons. Try creating a `Left Mouse Button` node and connecting it to a `Print String` just the way we did with the 1 key. Try the same with `Right Mouse Button` and `Middle Mouse Button`.

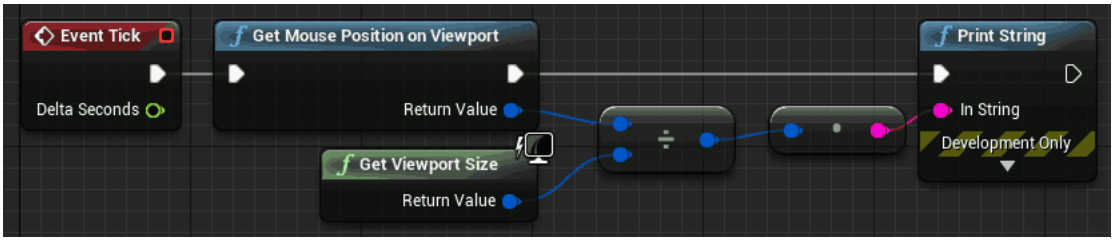
Next, create a `Mouse X` and `Mouse Y` node and connect them to a `Make Vector 2D` node. Draw the returned vector value to a `Print String`.



If you look carefully at the printout, you will notice that the values only seem to change when the mouse moves and are zero otherwise. This is because `Mouse X` and `Mouse Y` are actually reporting the *change* (delta) in the mouse position. But what if we want the cursor's absolute position in the window? To get this, we must use a different node called `Get Mouse Position on Viewport`. Create one of these nodes and print its output.

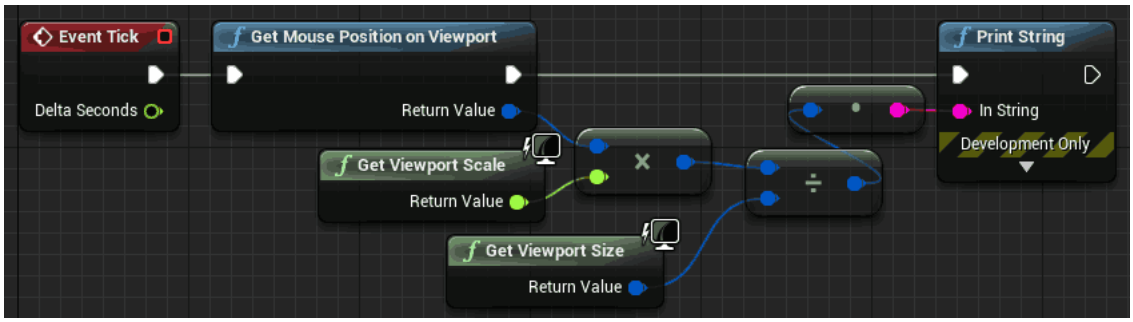


The output now reports the pixel position of the mouse in the window where (0,0) is the top-left corner. Note that there is a similarly named `Get Mouse Position on Platform`, however, this reports the mouse position across the entire screen/monitor. It is often helpful to normalize the mouse position so it lies in the interval $[0, 1]$. This way we can use the mouse position in a resolution independent manner. To normalize the mouse position, add a `Get Viewport Size` node and divide the mouse position by it. To perform the division, use a `vector2d / vector2d` node. This performs an element-wise division, which is what we want. Connect the outlet of `Get Mouse Position on Viewport` to the top inlet (the numerator) of the `vector2d / vector2d` node and the outlet of `Get Viewport Size` to its bottom outlet (the denominator).



Press Play and you will see... the values are in the right ballpark, but are not quite right. They do not seem correct on the right and bottom of the window. Why is that? This is due to DPI

scaling—the window and viewport may have different sizes. We need to compensate for this DPI scaling. Add a `Get Viewport Scaling` node and multiply the mouse position by it. Use a `vector2d * float` node to perform the multiplication.



Phew. We finally have a normalized mouse position that is (0,0) on the top-left and (1,1) on the bottom-right.

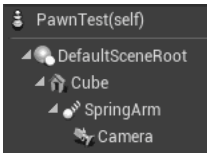
5.2 Player Control

In this tutorial, we will learn how to create a custom Pawn with our own movement logic and input mappings. Recall that a Pawn [23] is an Actor that can be "possessed" by a player or AI in order to move it around.

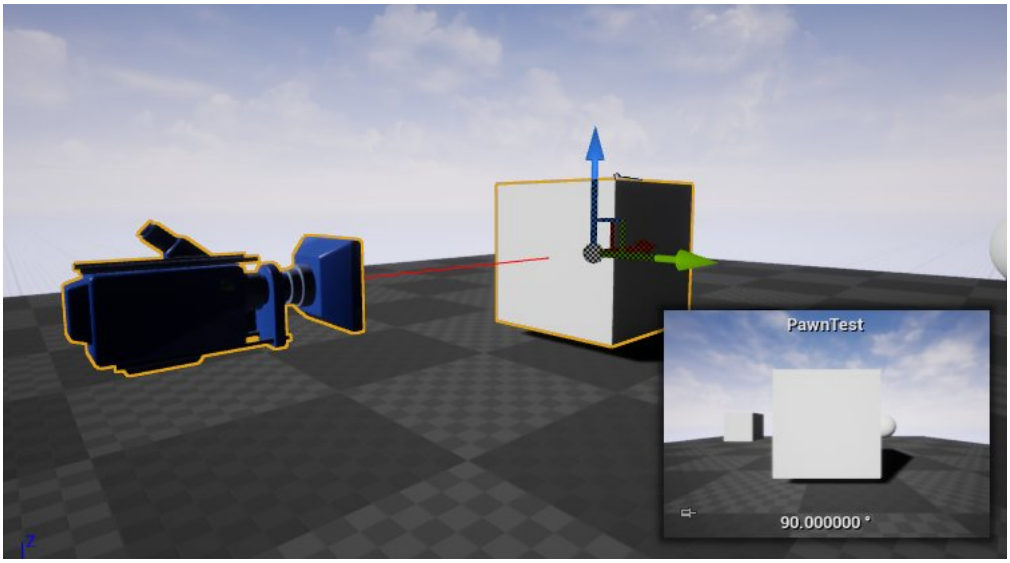
Create a new default level and name it "PlayerPawnTest". Create a new Blueprint with parent class Pawn and name it "PlayerPawn_BP". Note that it may be tempting to select the Character class, but that includes movement and other logic that we will not be using in this tutorial. Place an instance of the Pawn Blueprint class into scene. In Details, go down to the Pawn category and change Auto Possess Player to Player 0.

We do this so that our pawn will take over the player, rather than the default pawn constructed by the engine upon play.

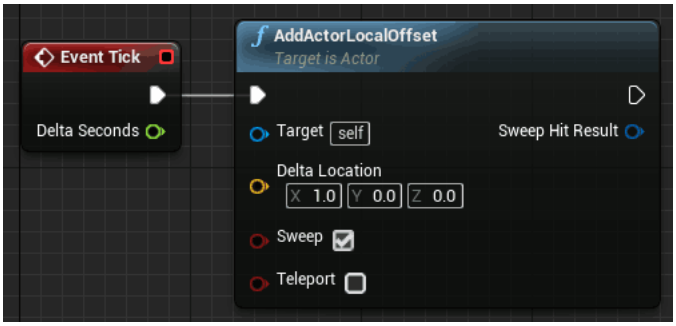
Before continuing, we will supply our Pawn with a mesh and a camera. Open the Pawn Blueprint. Add a Cube Static Mesh Component to the default scene root. To the cube, add a `SpringArm` component and then to the `SpringArm` add a `Camera` component. Your scene graph will look like this



`SpringArm` is very useful for creating a third-person perspective. A `SpringArm` will try to maintain a fixed distance from its parent, but shorten if it collides with something. This helps prevent the camera from being occluded by other objects in the scene. Select the `SpringArm` and set its Target Arm Length to 200. At this point, go over to the level editor and move the Pawn so it is out of the floor. You will also notice the camera is visible and there is a picture-in-picture view of what the camera sees.

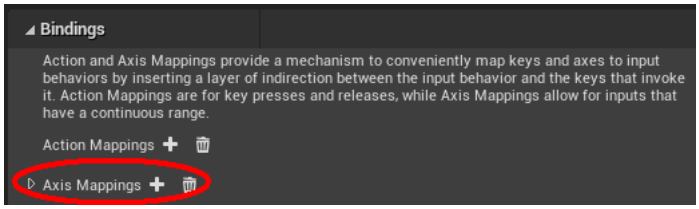


Next, we will edit the Pawn Blueprint to make the player move forward at a uniform velocity. Add an `AddActorLocalOffset` node and set the Delta Location value to (1, 0, 0). Connect `Event Tick` to the node to update it every frame.



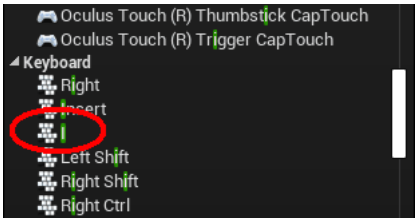
Press play and you should find yourself (a cube!) moving forward at a uniform velocity.

We would like to control the movement of the Pawn using custom keys. We could use key and mouse events directly from within Blueprints, but there is a far more flexible way to do this. We will configure *axis mappings* so we can assign collections of input events to the same movement event. On the menu bar, click Edit > Project Settings... and then go to Engine > Input. Under Bindings, you will see an area where we can add Axis Mappings.



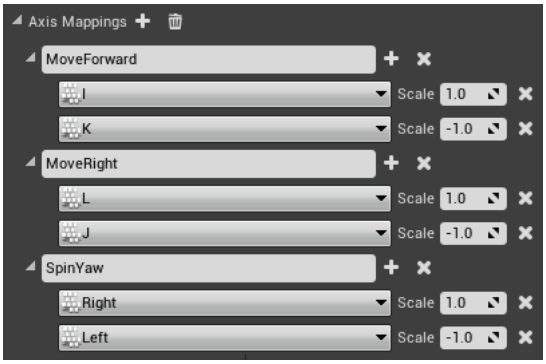
These mappings allow us to assign continuous (or discrete) events to custom variables that we can access from within our Blueprints. We will assign keyboard keys to forward and right movement variables. We will create an IJKL movement scheme so that I/K move forward/backward and L/J move right/left. Add a new Axis Mapping by clicking the plus symbol. Name the mapping "MoveForward". You will notice a key input already created, but it is mapped to None.

Click the drop-down and search for "i" and select the key I in the Keyboard category. You may have to scroll around a bit to find it.

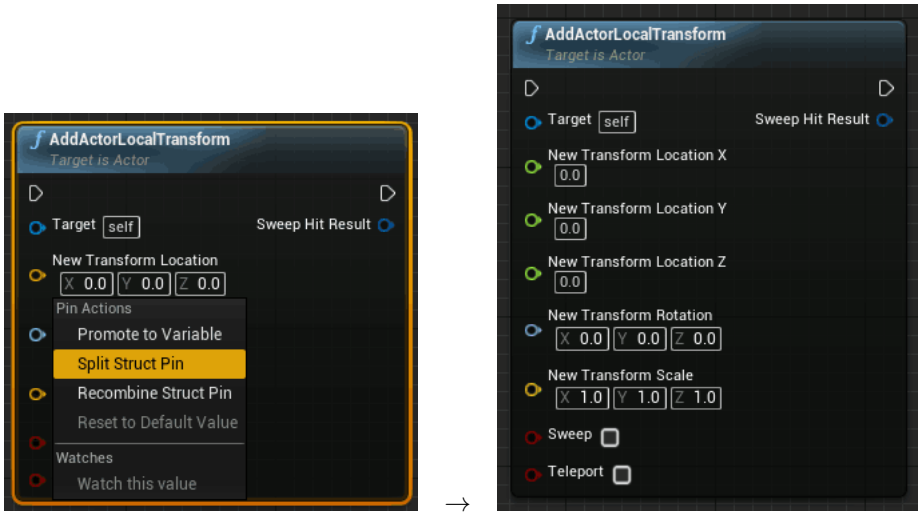


Leave the scale parameter at 1.0. Add another key K and set its scale to -1.0. The final forward movement axis mapping should look like this:

Repeat the same process to create a Move Right mapping mapped to the L and J keys and a SpinYaw mapping mapped to the left and right arrow keys. The completed axis mappings should look like the following:



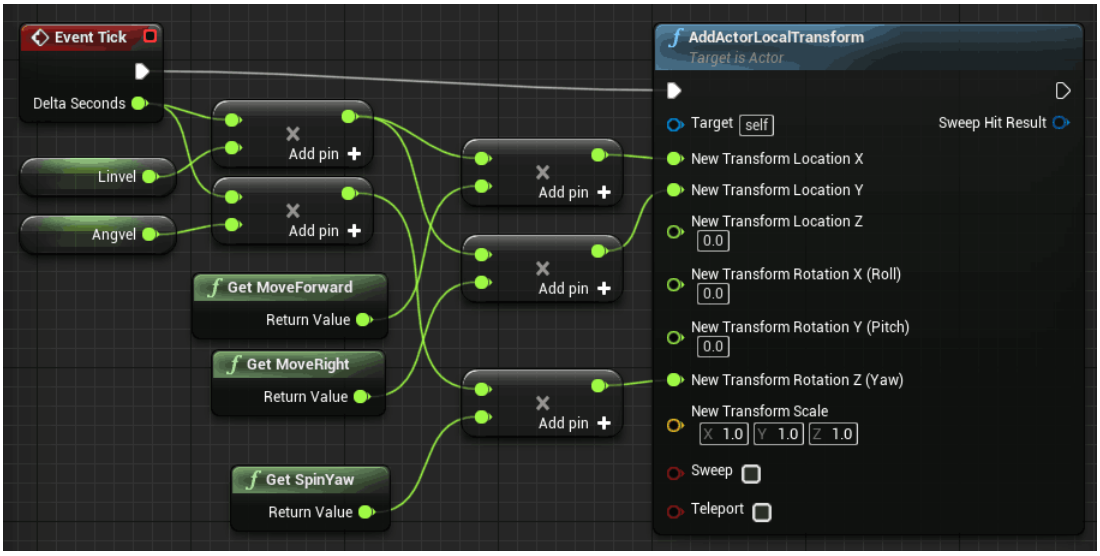
Now go back to the Pawn Blueprint created earlier and add an `AddActorLocalTransform` node. This node allows us to add offsets to the position and orientation of an actor. To simplify wiring, we will "split" a couple of the inlets on the `AddActorLocalTransform` node to expose the members of the underlying structs. Right-click New Transform Location and select Split Struct Pin.



After doing so, you should see the X, Y and Z members of the underlying Vector. Do another split on the New Transform Rotation inlet. The last step is to hook the axis mappings we made above into `AddActorLocalTransform`. The axis mapping nodes are available as `Get MoveForward`, `Get MoveRight` and `Get SpinYaw`. Create these three nodes and connect them, along with `Event Tick`, to `AddActorLocalTransform`.

If you play the level, you should be able to move the Pawn around with the keyboard. However, we are not quite done yet...

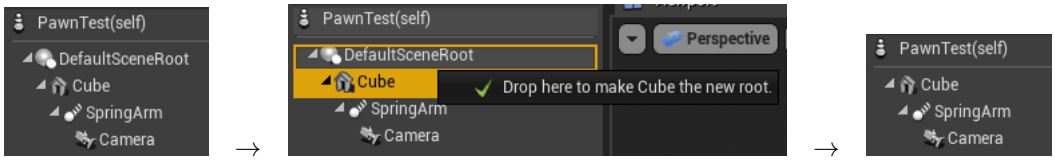
At the moment, our movement is dependent on the frame rate. This is bad practice. We must incorporate the delta time between frames into the delta position and rotation values. The best way to proceed is to create variables to store the linear and angular velocities. Create two float variables called 'linvel' and 'angvel'. Set their default values to 100.0. Create getters for each and then multiply them by the `Delta Seconds` outlet of `Event Tick`. Finally, multiply these products by their appropriate axis mapping nodes. Your graph should look something like this:



I

Ghost Pawn?

If you move the pawn around a level with other objects in it, you may notice that the pawn passes through objects like a ghost. The key thing to understand is that for collisions to be active for an actor, it must have a collider at its root. The engine will always check for collisions at the root component of an actor. If there is no collider at the root, then no collision checks occur. The problem with our pawn is that it uses the DefaultSceneRoot which has no collider. Fortunately, the cube static mesh component has a collider built into it, so we can use it. In the Blueprint, simply drag the cube component onto the DefaultSceneRoot to make the cube the new root component.



The last step is to tell the actor to generate overlap events whenever it moves. This can be accomplished by enabling Sweep on the `AddActorLocalTransform` node.

Now play the level and the pawn should collide with other objects.

6 Audio

6.1 Playback Basics

Unreal Engine supports a very basic means of generating audio not unlike playing vinyl records on a record player. There are two main pieces involved with audio playback: the audio sample and the player. An *audio sample* is a digitized recording of a sound, specifically its pressure values over time. Audio samples are typically stored in files that are then loaded into computer RAM for playback. A *player* reads through a sample and sends its values to the DAC (digital-to-analog converter) which is a hardware audio output device. Sample playback is also called *sampling synthesis* and dates back at least to the 1920s when the first variable speed phonographs were invented [35, pp.117–133].

6.2 Importing Sounds

To import a your own samples, simply drag and drop an audio file into the Content Browser or click the Import button [11]. When new files are added, their icon in the Content Browser has a small asterisk symbol in the corner indicating they are in an unsaved state.

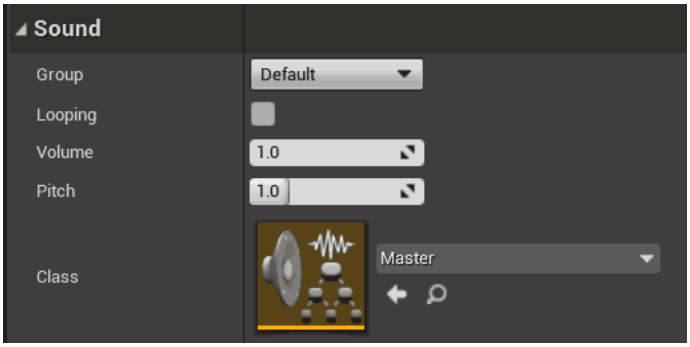
By default, imported audio files are referenced from disk and are not saved with the project. You must save the audio file to convert it into a persistent asset.

Many file formats may be imported including WAV, AIF, SND, FLAC, MP3, OGG and AAC. There are several trade-offs to consider when choosing the file type [49]. Some of these are summarized in Table 2. In general, the longer the audio sample you want to use is, the more compression you will need to keep memory consumption down. However, decompression takes up CPU cycles. For very short sounds (< 200 ms), a good choice is an uncompressed format such as WAV, AIF or SND since maximum quality is obtained with relatively low memory use and decoding time. For longer sounds, like background music or dialogue, a lossy compression format such as MP3 will be a better choice since the quality is still very good, but the memory usage will be around 10x less than that of an uncompressed file. A good rule of thumb is to use the shortest sounds possible as they have the least impact on memory and computation time.

File Format	Compression	Quality	Memory Usage	Decode Speed
WAV, AIF, SND	None	Highest	100%	Fastest
FLAC	Lossless	Highest	60%	Moderate
MP3, OGG, AAC	Lossy	Good to Very Good	10%	Slowest

Table 2: Basic comparison of uncompressed and compressed audio file formats. The different formats trade-off between quality, memory usage and decoding speed.

If you double-click an audio sample in the Content Browser, you can see its Details pane where a variety of options can be set such as volume, pitch and looping.



The *volume* can be used to adjust how loud the audio sample is. It is important to know that hearing is logarithmic with respect to the amplitude (pressure) of sounds. This means every doubling of amplitude results in a linear increase in perceived loudness. This is why loudness, L, is typically given in decibels (dB) which is on a logarithmic scale ($L(p) = 20\log(p)$ where p is pressure). Every doubling or halving of amplitude results in a change of 6.02 or -6.02 dB, respectively.

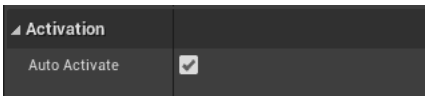
Pitch changes the rate of playback. This effect is known as *varispeed* [38] and the result is to change both the pitch and duration of the sound. For example, lowering the pitch by 0.5 will double the length of the sound.

Looping plays a sound indefinitely by restarting it from the beginning once it reaches the end. Looping is useful for ambient sounds (environmental effects, machine noises, etc.) and background music.

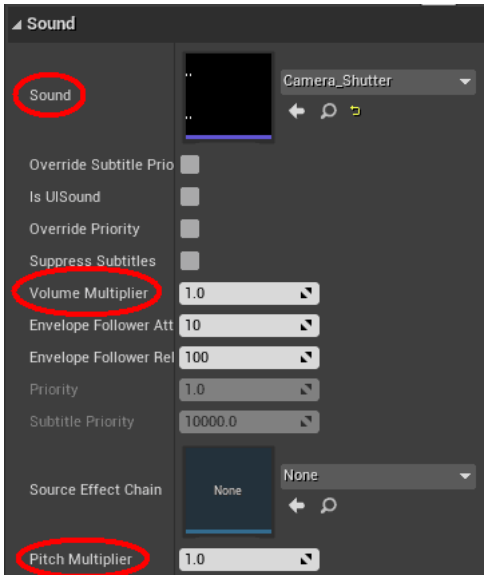
Now with some understanding of how audio works in the engine, we will try adding audio to our scene.

6.3 Ambient Sound

Try importing an audio file into the Content Browser and then dragging and dropping it in your scene. You will notice it gets converted into something called an `Ambient Sound`. If you check the Details pane, you will see that the `Ambient Sound` is an `Actor` and has an `Audio Component`. An `Audio Component` is an `Actor Component` that adds sound playback functionality to an `Actor`. By default, an `Audio Component` will play its sound when you start the application. This can be adjusted through the `Auto Activate` member.



Next, find the Sound category in the Details pane. Here we find several parameters of primary interest, namely: Sound, Volume Multiplier and Pitch Multiplier.

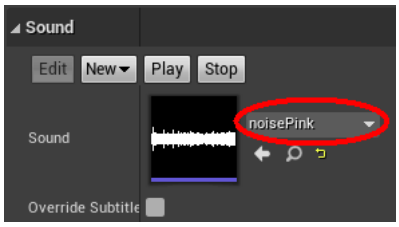


Sound specifies the audio sample to play, which may be changed to whatever you like. The volume and pitch multipliers allow you to adjust the Audio Component playback relative to the volume and pitch of the source audio sample. For example, if the source audio sample pitch is set to 0.5 and the Audio Component pitch multiplier is set to 0.5, the playback pitch will be 0.25 (0.5 x 0.5).

6.4 Ambient Sound: Wind Effect

In this tutorial, we will create a synthesized wind effect. In the process, we will learn how to modulate audio parameters programmatically from within a Blueprint.

Begin by creating a new `AmbientSound` Blueprint class and name it "AudioWind_BP". Before proceeding, we will prepare an audio sample to be used as the basis of the wind. Import the pink noise sound file into the Content Browser. Double-click the audio sample and enable Looping so that it plays indefinitely. Save the changes. Next, drag add an AudioWind instance into the scene and then open its Blueprint. You will notice that it has an `AudioComponent` at its root. Click `AudioComponent` and in Details, set its Sound to the pink noise.



Press play and you should hear the noise playing continuously.

At the moment, we hear something like the crash of a waterfall. However, the sound of wind is more dynamic as it is constantly changing in intensity over time. In order to capture these dynamics, we will change the volume of the sound over time using a random function. The random function should vary smoothly over time and we would also like to be able to control the rate at which the noise fluctuates. A good candidate for this is value noise. Value noise takes a position (phase) as input and produces a smoothly varying noise function as output [1] (Figure 16).

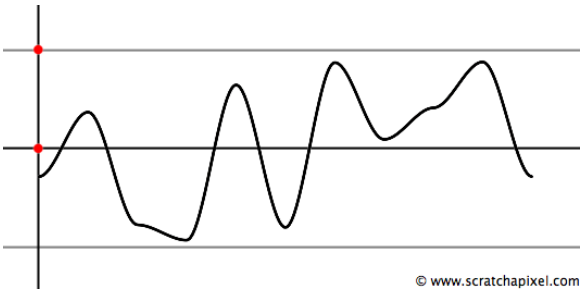
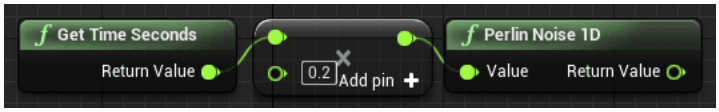


Figure 16: One-dimensional value noise takes a position as input (x axis) and produces a smoothly varying random value as output (y axis).

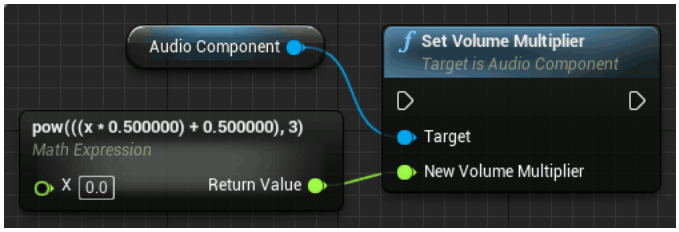
In Blueprints, one-dimensional value noise is available as `Perlin Noise 1D`. Create a `Perlin Noise 1D` node and connect to its input a `Get Time Seconds` sent through a multiplier with value 0.2.



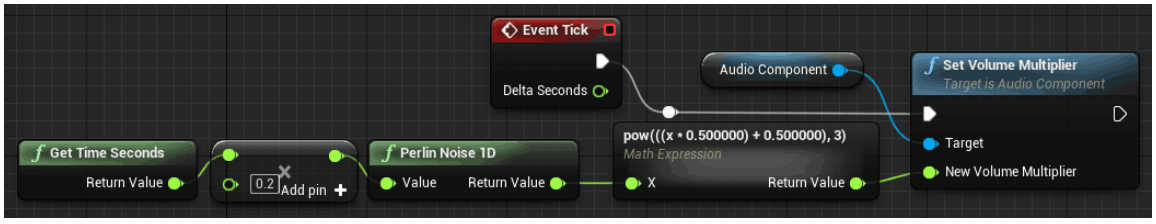
The multiplication factor corresponds to the frequency of the noise.

Next, we will map the value noise to the amplitude of the audio playback. Create a `Set Volume Multiplier (AudioComponent)` node and connect a Math Expression to its Volume Multiplier input. Since the value noise outputs values in $[-1, 1]$ and the volume is in $[0, 1]$, we need to apply the mapping $f(x) = \frac{1}{2}x + \frac{1}{2}$ to the value noise. We also would like the wind to be mostly quiet with some sporadic gusts. To approximate this, we can apply a cubing operation to the value going into the volume. Create this mapping by setting the math expression to

```
pow(x*0.5 + 0.5, 3)
```

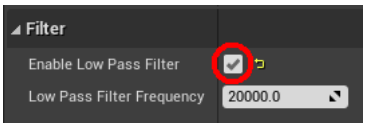


Connect the output of the value noise to the input of the math expression. We want to modulate the volume continuously, so connect `Event Tick` to the Exec input of `Set Volume Multiplier`. Altogether, your graph should look like the following



Press play and you should hear a randomly modulating wind sound. To add a bit more realism, we will make louder gusts brighter and quieter gusts duller. We can accomplish this by using a low-pass filter. A low-pass filter attenuates frequencies above some cut-off frequency and passes all

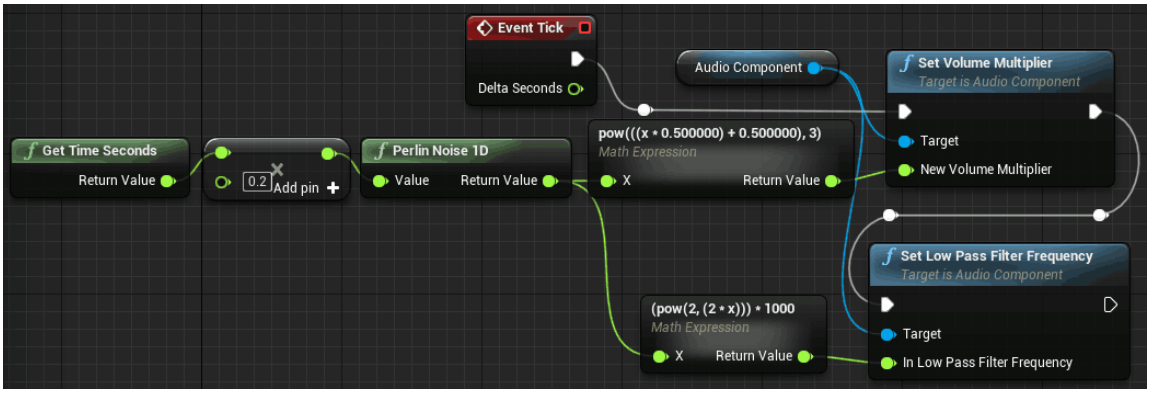
frequencies below the cut-off. Fortunately, all Audio Components have a built-in low-pass filter, however, it must be enabled. Click on AudioComponent in the Components panel and then in the Details panel Enable the Low Pass Filter.



We will need to use a different mapping to go from the value noise to the filter frequency, since we hear frequencies on a logarithmic scale. For this, we can use the mapping function $g(x) = f \cdot 2^{cx}$ where f is the base frequency and c is the spread, in octaves. If we plug value noise into, we get frequencies varying on a logarithmic scale in the range $f[2^{-c}, 2^{c}]$. For example, if $f = 1000$ and $c = 1$, the range is $1000[\frac{1}{2}, 2]$ or $[500, 2000]$. Create another Math Expression node and set its expression to

```
(pow(2, (2 * x))) * 1000
```

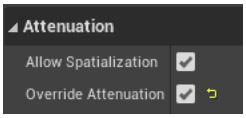
To set the filter frequency, create a `Set Low Pass Filter Frequency` node. Join the aforementioned nodes with the value noise to end up with the final graph



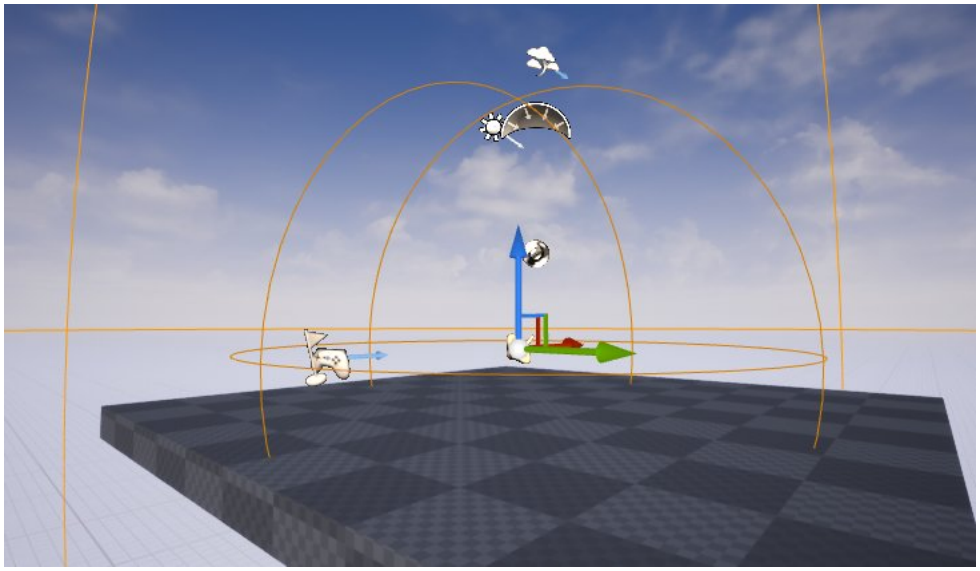
Press play and you should hear the wind change in brightness according to its amplitude.

6.5 Distance Attenuation

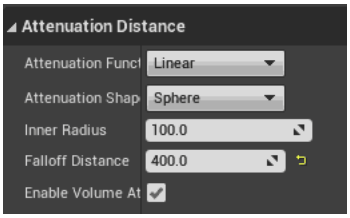
Like light, the intensity of sounds diminish as a function of distance. Sounds are louder when close by and quieter when far away. This is due to energy dissipation as the sound waves travel through air. We can simulate this effect by adding distance-based attenuation to our sounds. Distance attenuation is just one of several perceptual effects we can utilize to *spatialize* audio. By default, `Audio Components` have attenuation disabled so they are the same volume regardless of distance. We will build a simple scene with a sound that gets quieter as you move away from it. Move your `Ambient Sound` actor to the center of the scene. Next, ensure that the audio sample assigned to the `Audio Component` is set to loop, so we get a continuous sound. Select the `Audio Component` and under Attenuation check Override Attenuation.



You should see a two outline spheres appear around the Actor in your scene—one small and one very large.



These spheres indicate the inner radius and falloff distance of the sound attenuation. For distances inside the inner radius, their is no attenuation and for distances past the falloff distance, there is full attenuation (silence). Under Attenuation Distance, set the inner radius to 100 and the falloff distance to 400.

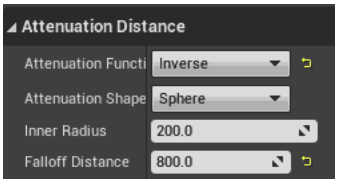


Press play and then move around the scene to see hear how the sound gets quieter the further you are from the audio source. The default attenuation function is Linear which can sound unnatural. The sound comes in too abruptly as you enter the falloff threshold. In reality, sound amplitude is inversely proportional to distance. Try changing the attenuation function to Inverse and listen to the result. You will notice that the sound fades in more gradually, but also that it increases in volume quite suddenly as you near it. Double the values of the inner radius and falloff distance and listen again. Hopefully you agree that Inverse sounds more natural than Linear. Why use Linear? The answer is that Linear allows us to more aggressively cull sounds that cannot be heard since it allows for shorter attenuation distances. This can lead to performance savings. However, the trade-off is that Linear attenuation sounds less natural, especially near the falloff threshold.

6.6 Audio Component: Hide and Speak

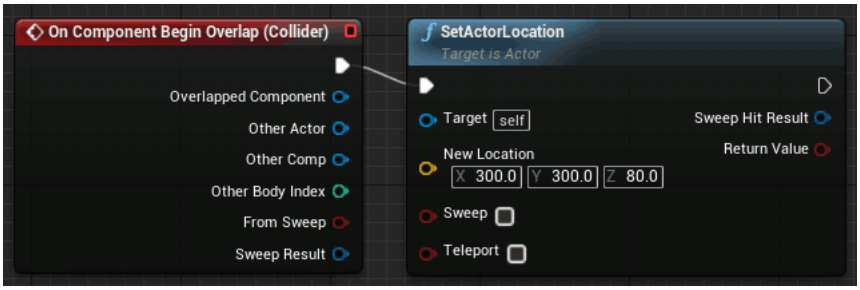
In order to do some audio programming, we will create a simple game where a shape teleports to a random location and then emits a sound when you get near it. This exercise will teach you how to add an `Audio Component` to an existing actor and how to program basic audio events.

We will use a different setup than we had with the `Ambient Sound` actor since that is configured for non-visible entities. Create a new level and place a sphere mesh in the scene. Add a `Sphere Collision` component to the mesh component and set its scale to 3.0. The collider will be used to trigger playback when the player is in proximity. Add an `Audio Component`. Set its sound to something short in duration (under one second) and ensure that the audio sample is non-looping and that auto-activation is disabled (so the sound does not play when we start the level). Enable Override Attenuation and set the Attenuation Distance parameters to the following



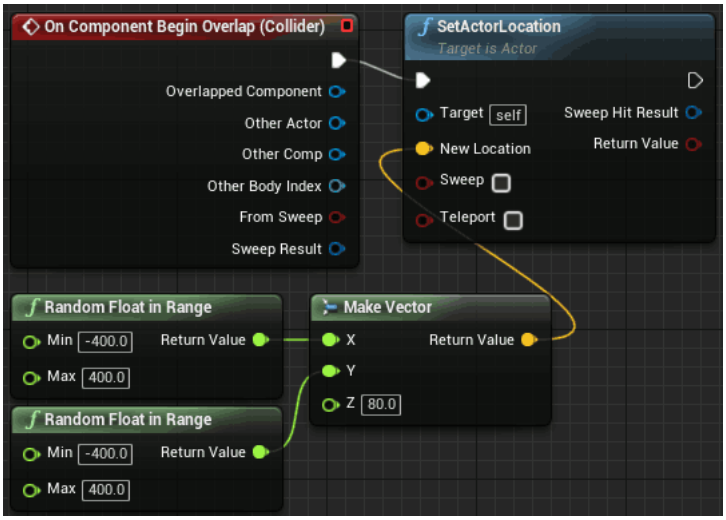
Now create a new Blueprint from this actor called "AudioActor_BP" and go to its Event Graph.

Our first task will be to move the actor whenever the player overlaps with its collider. Add an `On Component Begin Overlap` event for the collider and connect its `Exec` outlet to a `SetActorLocation` node. Set New Location to (300, 300, 80).



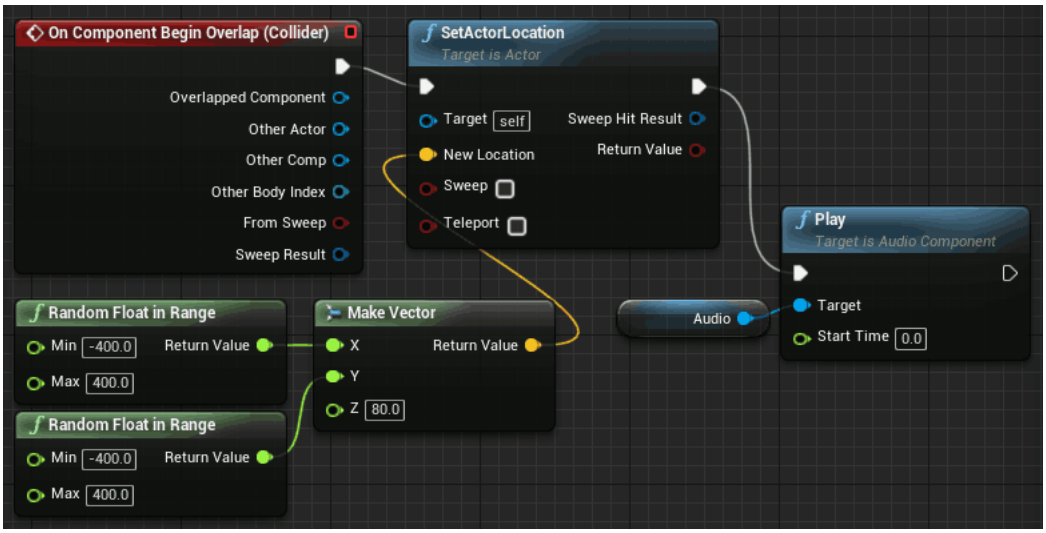
Press play and ensure that the actor moves when you get close to it.

We want to actor to move to a random location, so add two `Random Float in Range` nodes and set their min and max to -400 and 400. Connect these to a `Make Vector` node with `Z` set to 80. Connect the random vector to the `SetActorLocation` node.



Play again to test that the actor moves to a random location when you get near it.

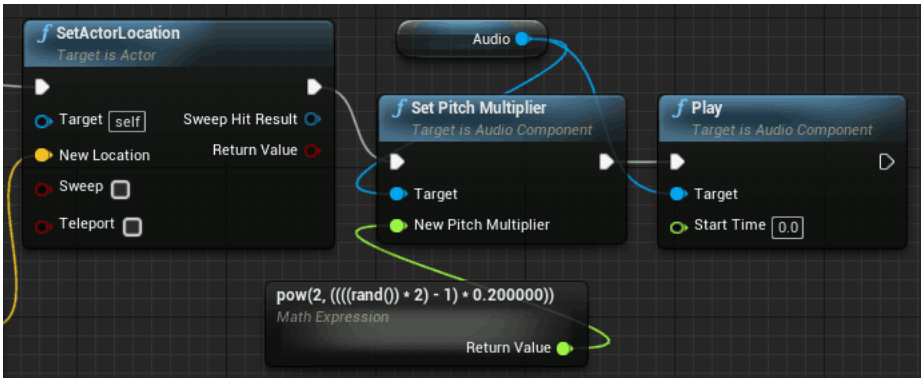
We need to trigger the audio playback whenever the actor moves. Fortunately, this is very easy. Simply add a `Play` node and connect the `Exec` outlet of `SetActorLocation` to its `Exec` inlet. The target of the `Play` node is the `Audio Component`. The final graph should look like this:



To make things more sonically interesting, we will randomize the pitch of the playback. Add a `Set Pitch Multiplier` node and insert it between the `SetActorLocation` and `Play` nodes. Use the following math expression to set the pitch multiplier:

```
pow(2, (rand()*2-1)*0.2)
```

The updated section of the graph should look like this:



The math expression in normal notation is $2^{\delta(2U-1)}$ where U is a uniform random number in $[0, 1]$ and δ is the amount of pitch variation in octaves. We use an exponential function (2^x) since we hear pitch on a logarithmic scale.

7 Particle System

7.1 What Is It?

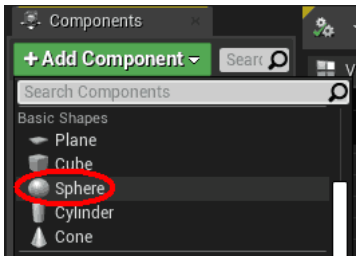
A particle system is a set of particles that collectively produces some kind of emergent phenomenon not present in the individual particles. For example, a particle system may be used to model natural phenomena such as fire, smoke, snow or mist. Karl Sims' particle behavior language describes several composable operations on particle states that can be used to create various effects such as vortices, spirals, and bouncing [37]. The animation system is demonstrated in his work *Particle Dreams* [36]. Particle systems are also useful starting points for modeling groups of interacting agents such as bird flocks, fish schools and insect swarms [33] or for steering more general types of virtual characters [34]. Particle systems may also be used for abstract art as in John Whitney's differential motion [41] and visual music composition *Arabesque* [42] and in Char Davies' *Osmose* for autonomous creature movement [7].

Figure 17: Output from Karl Sims' particle behavior language.

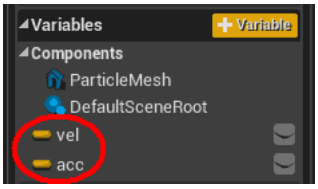
7.2 Making Our Own

Unreal Engine has a dedicated particle system engine called Cascade. While powerful, it is not fully programmable. In this section, we will build our own particle system with fully-programmable particles from the ground up in Blueprints. In the process, we will learn how to spawn actors to create and destroy objects dynamically. We will build our particle system using two separate Blueprint classes: a Particle and a Particle System. Each Particle will be a simple Newtonian particle with velocity and acceleration and the Particle System will be responsible for spawning and initializing new particles at some fixed rate.

Begin by creating a new level called ParticleSystem. Next we will create a Blueprint class for the particle. Create a new Blueprint Actor and name it "Particle_BP". Open the Blueprint and from the Components pane add a Sphere `StaticMeshComponent`.



Rename the component "ParticleMesh". Set the scale of the ParticleMesh to 0.25 in the Details pane and ensure that it is movable. Next, we will add two member variables to store the particle's current velocity and acceleration. In the left pane, click the + symbol to the right of Variables and set the variable type to Vector (yellow) and name it "vel". Repeat the same for a second variable named "acc".



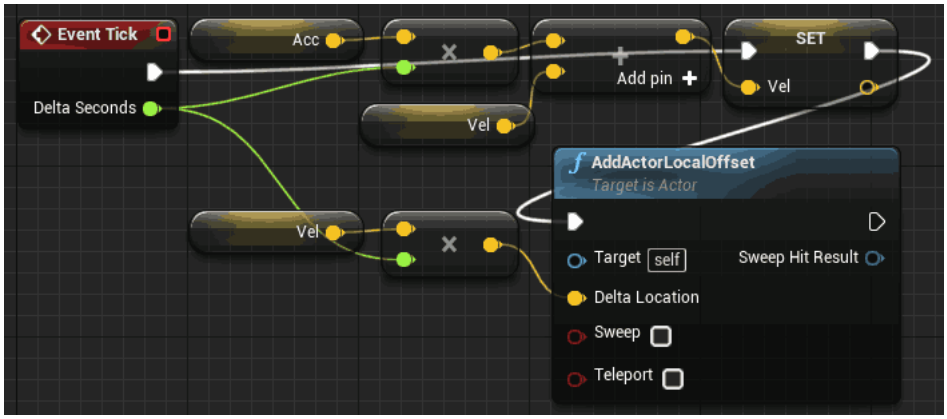
Now we will write the code to update the particle position based on its velocity and acceleration. We will use what is called the Euler method to update the particle. First the velocity gets updated and then the position gets updated. The math goes as follows

$$\begin{aligned} v &\leftarrow v + a\Delta t \\ p &\leftarrow p + v\Delta t \end{aligned}$$

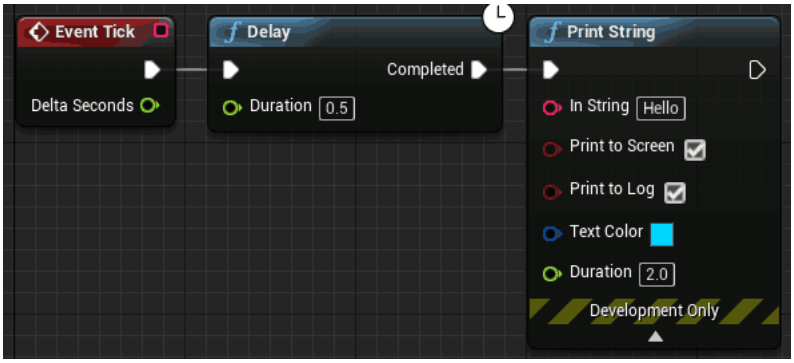
where a is acceleration, v is velocity, p is position and Δt is the time step. We need to translate the formula above into a Blueprint graph. We will start with the velocity update. Drag and drop the `vel` and `acc` variables onto the canvas and select Get to create getters for these variables. Multiply `acc` by the `Delta Seconds` output of `Event Tick` using a `vector * float` node. The result of this operation is a velocity delta that we add to the old velocity value to get its new value. Use `vector + vector` to add this velocity delta to the `vel` variable.

Next, we need to set `vel` with its new value. Drag and drop another `vel` variable onto the canvas and choose Set. Connect to outlet of `vector + vector` to the inlet of the `Set vel` node. Whenever you create a setter, it is good practice to immediately connect something to its Exec inlet, otherwise the variable will not get set! Connect the `Event Tick` `Exec` outlet to the `Set vel` `Exec` inlet.

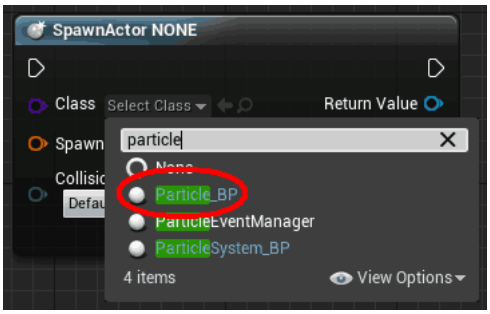
The update logic for the particle velocity is now complete. Let us now update the particle position. The particle position is actually stored in the Actor, so we need to set it using a special function called `AddActorLocalOffset`. Add an `AddActorLocalOffset` node and connect to its `Delta Location` inlet, the result of `vel` times `Delta Seconds`. Connect the `Exec` outlet of the `vel` setter (from the previous step) to the `Exec` inlet of `AddActorLocalOffset`. This ensures that the location (position) of the Actor gets set and that it happens *after* the velocity is updated. You final Particle Blueprint should look something like the following:



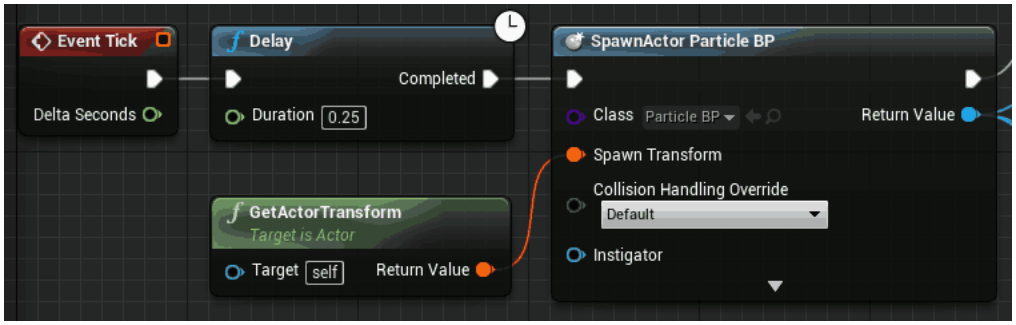
Make sure to compile and save the Blueprint. We will now make a second Blueprint for the particle system. The basic principle for the particle system will be for it to periodically spawn a new Particle Actor having a fixed lifetime. Begin by creating a new Blueprint Actor and name it "ParticleSystem". Open the Blueprint and go to its Event Graph. Before doing anything, we will first learn how to fire off periodic events. The simplest way is through the `Delay` node. `Delay` takes an `Exec` input and then waits a specified duration before sending the event to its outlet. Any events coming in during the waiting period are ignored. Add a `Delay` node and connect `Event Tick Exec` to its `Exec` inlet. Set the duration of the `Delay` to 0.5 (seconds) and connect its `Exec` outlet to a `Print String` to test it out.



Press play. If everything is working right, you should see the message printed to the screen every 0.5 seconds. When done testing, you may remove the `Print String`. Next we will connect the output of the `Delay` to a node that spawns actors. Create a `SpawnActor from Class` node and set its `Class` field to `Particle_BP`.

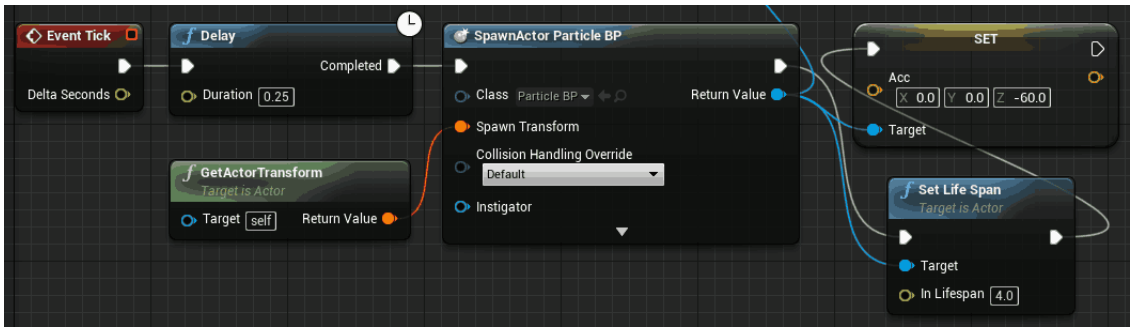


Now connect the `Delay Exec` to the `SpawnActor Exec`. This will cause a new particle to be spawned with a period equal to the duration of the `Delay`. (You may want to lower the `Delay` duration to something like 0.25 seconds.) We want to use the particle system position as the initial position of the particle. To do this, create a `GetActorTransform` node and connect its outlet to the `SpawnActor`'s `Spawn Transform` inlet.

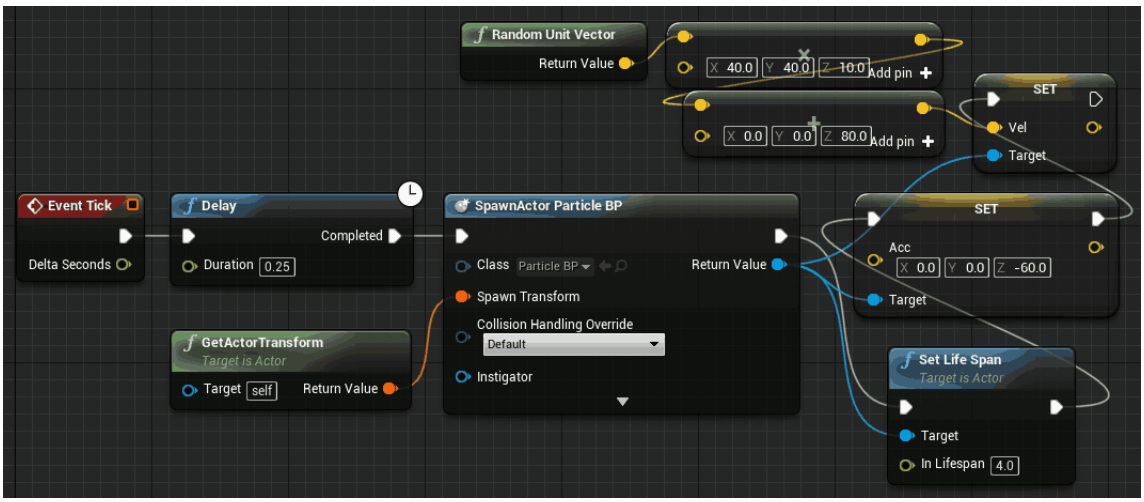


At this point, our particle system will spawn new particles, however, the particles will 1) never get destroyed and 2) have zero velocity and acceleration. The return value of `SpawnActor` is a

reference to a newly created `Particle_BP` object. We use this reference to initialize the particle. Drag off the `SpawnActor` outlet onto an empty spot on the canvas and then create a `Set Life Span` node. Set the lifespan to 4.0 seconds. Create another node off of `SpawnActor` called `Set acc`. This will allow us to set the `acc` member variable of the `Particle_BP` object we just spawned. Set `acc` to (0,0,-60). At this point, your graph should look something like this:



Finally, we will set the velocity of the particle to a random value to create a fountain effect. Create a `Set vel` node off of `SpawnActor`. We want to set the velocity to a random vector so create a `Random Unit Vector` node. This node outputs a unit vector with a random orientation. Scale and offset the random vector and plug it into `Set vel`. Remember to connect all your `Exec` pins so the variables get set. In this situation, the order does not matter. Here is the final graph:



Compile and save the Blueprint, drag a ParticleSystem into the scene and press play. If you see a fountain (Figure 18), congratulations, you have created your own programmable particle system!

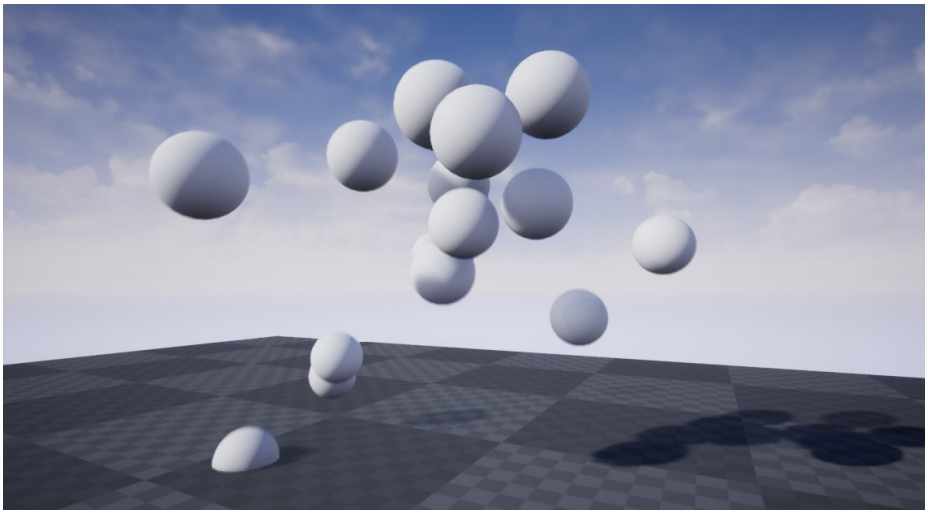


Figure 18: Our custom particle system.

Here are some things you can try next:

- Replace the sphere mesh with a Billboard Component. Billboards are textures on quads that always face the camera. If you plan on having many particles, billboards can be more efficient to render than 3D meshes since they have fewer vertices.
- Try creating different effects like a waterfall, bubbles, radiation or (firework) explosion. This will require changing the initial parameters (velocity and/or acceleration) of the particles.
- Add some jitter to the movement of the particles using a random vector.

7.3 Anti-popping

You likely noticed a visual issue with our particles—they suddenly pop out of existence at the end of their lifetime. This looks very unnatural and we would prefer a more gradual exit from the scene. One strategy that works regardless of the particle appearance is to scale it down to zero by the end of its lifetime. We will now enhance our particles to use this anti-popping strategy.

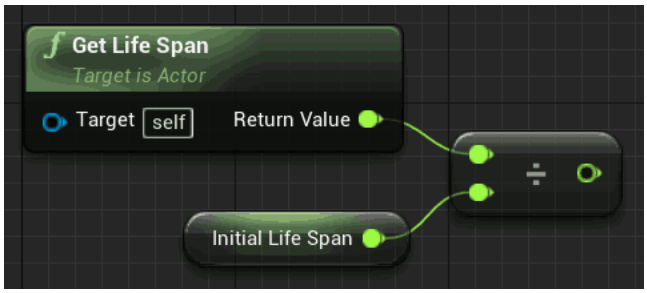
We first need to get the current age of the particle in order to determine how far along in its lifespan it is. Open the Particle Blueprint and add a `Get Life Span` node. One misleading aspect of this node is that it does not return the total lifespan, but rather the *life span remaining* of the actor. Thus, it is not symmetric with `Set Life Span` (as we used in the Particle System Blueprint) which sets the time until the actor gets destroyed! We need to map the particle life span into a value in the range $[0, 1]$ which we can use to scale the particle. We accomplish this with the following formula

$$scale = 1 - \frac{age}{age_{max}}$$

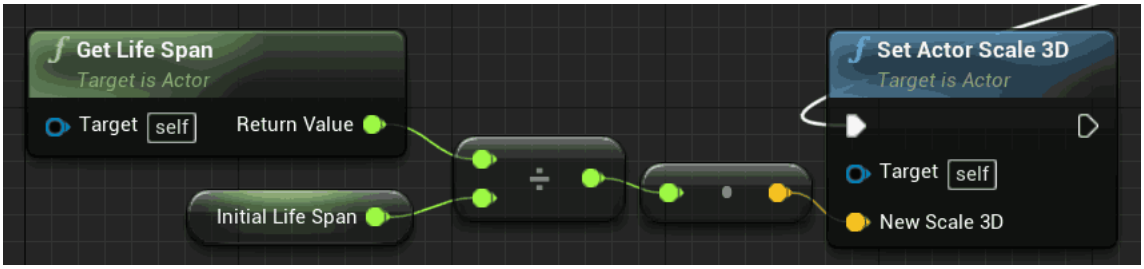
where age is the current age, in seconds, of the particle and age_{max} is the maximum age of the particle. We can get the maximum age of the particle from another node called `Get Initial Life Span`. We do not have a direct way to get the particle age, however, `Get Life Span` gives us $age_{max} - age$. If we substitute $\frac{age_{max}}{age_{max}}$ for 1 in our formula above, we get, after simplification,

$$scale = \frac{age_{max} - age}{age_{max}}.$$

Therefore, we construct the following graph to compute the scale factor:



The last step is to scale the actor by this amount. Create a `Set Actor Scale 3D` node and connect the scale factor to its `New Scale 3D` inlet.



Note that the editor will automatically create a float-to-vector conversion node that simply sets all elements of the vector to the float value. Remember to connect to the Exec inlet to do the actual scaling. If done correctly, you should see the particles scale down to zero before they get destroyed (Figure 19).

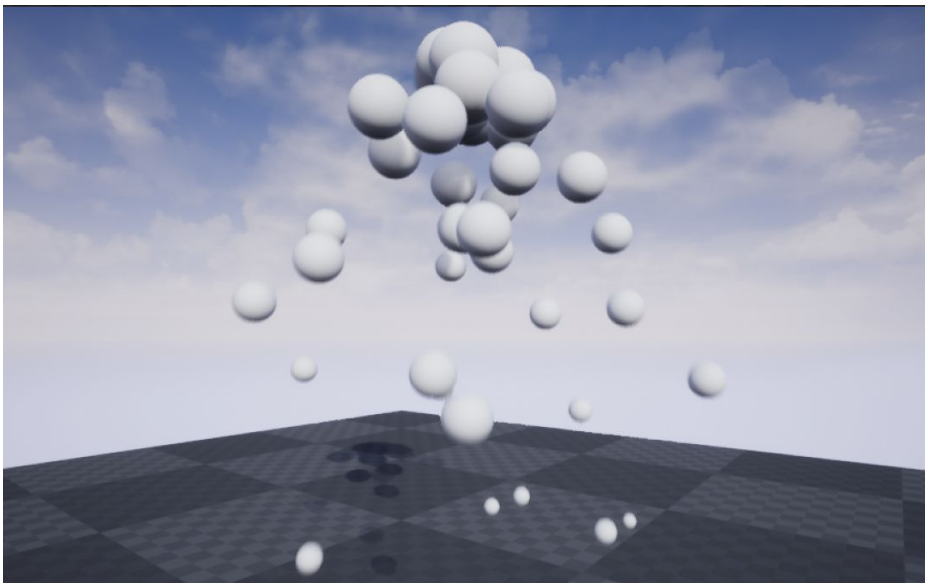


Figure 19: Particle system with particles modified to scale down to zero by the end of their life span.

While this anti-popping measure is certainly an improvement, it has a tendency to make the particles appear smaller since they are constantly be scaled down over their lifetime. We would prefer that they only get scaled down near the end of their lifetime and otherwise remain unscaled (scaled by 1). We can accomplish this using the following mapping

$$scale \leftarrow \min(1, scale/f)$$

where f is a value in $(0, 1]$ that determines at what remaining fraction of the particle's lifetime the scaling down begins. Note that f cannot be a value of 0, otherwise we get division by zero. A good programmer will safe-guard their code against someone entering bad values, so we will modify the above formula to

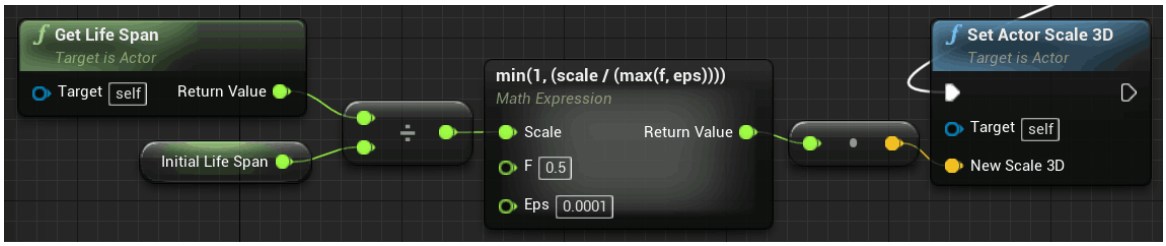
$$scale \leftarrow \min(1, scale / \max(\epsilon, f))$$

where ϵ (Greek "epsilon") is a very small positive value.

Create a `Math Expression` node with the following expression

```
min(1, (scale / (max(f, eps))))
```

and then set `f` to 0.5 and `eps` to 0.0001. Insert the math expression just after the division operation.



Play the level and you should see that the particles remain unscaled for the first half of their lifetime and then get scaled down for the last half (Figure 20).

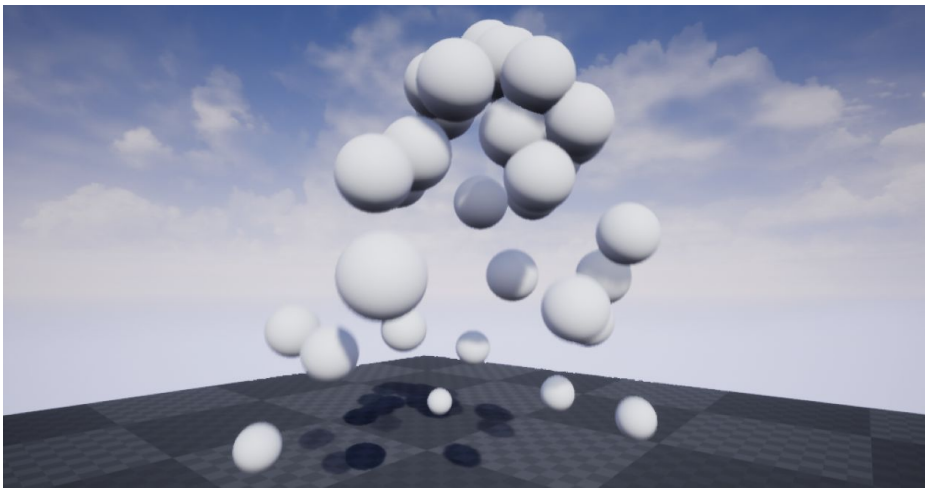


Figure 20: Particle system with particles modified to scale down to zero starting halfway through their lifetime.

8 Materials (Shaders)

The Unreal Editor supports customization of surface attributes through the notion of Materials. A *Material* [20] defines how a surface reacts to light. Additionally, Materials allow dynamic control over mesh positions (vertices). Materials in Unreal are equivalent the (programmable) shaders seen in other graphics interfaces like Maya and OpenGL. In fact, under the hood, Unreal Materials are built directly on top of HLSL [8] shader code. A simplified shader pipeline (Figure 21) consists of a

vertex shader and a pixel (or fragment) shader. The vertex shader modifies vertex attributes (colors, normals, texture coordinates, etc.) of the incoming mesh and the pixel shader colors in each pixel (e.g. for lighting). The job of the rasterizer is to convert vertex geometry into "fragments" of pixels that "fill in" the vertex geometry. The rasterizer interpolates vertex attributes across the connected vertices. In the final stage of the pipeline, fragments are composited with the output frame buffer.

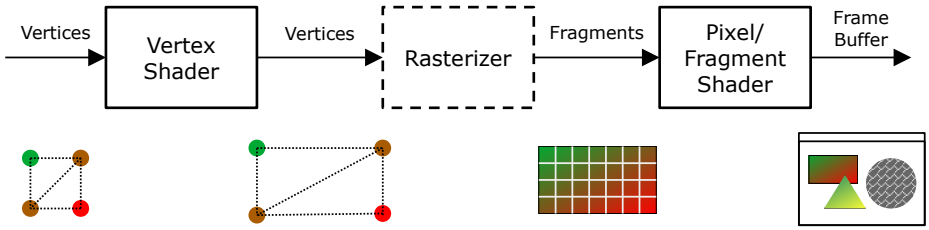


Figure 21: A simplified shader (material) pipeline consisting of programmable vertex and pixel/fragment shaders. The vertex shader transform vertices from the incoming mesh. The rasterizer converts vertex geometry into fragments of pixels. The pixel shader assigns colors to pixels in each fragment. Finally, fragments are composited with the output frame buffer.

Custom Materials may be created through the Material Editor [17] which is discussed next.

8.1 Material Editor

Custom Materials are created just like another other asset—by clicking "Add New" in the Content Browser. This will create a new Material asset. If you double-click the Material, it will open up in the Material Editor. The Material Editor operates very much like Blueprints in that various nodes are connected to define custom procedures. The main difference is that the nodes available in the Material Editor are more oriented towards working with colors, textures, normals, light and other shading constructs. The Material Editor *looks* like Blueprints, but it is in fact much more limited in scope. For example, a Material cannot have its own state (variables) like a Blueprint and we do not have access to all the state of the game.

Figure 22 shows the basic layout of the Material Editor. The Material Editor UI has many features [18], but we will only discuss the most basic ones here. In the center pane is the graph editor, very much like Blueprints. Like Blueprints, right-clicking an empty spot on the canvas brings up a searchable menu of nodes. Nodes are connected to one another just like Blueprints. The upper-left pane shows a live preview of the material. Whenever you change the program, the underlying shader will recompile and there will be a slight wait before the preview gets updated. The left and right mouse buttons rotate and zoom the camera, respectively. Several different preview shapes may be selected from in the bottom-right corner. The Details panel contains parameters that determine how the material interacts with the main renderer, such as blending and lighting modes.

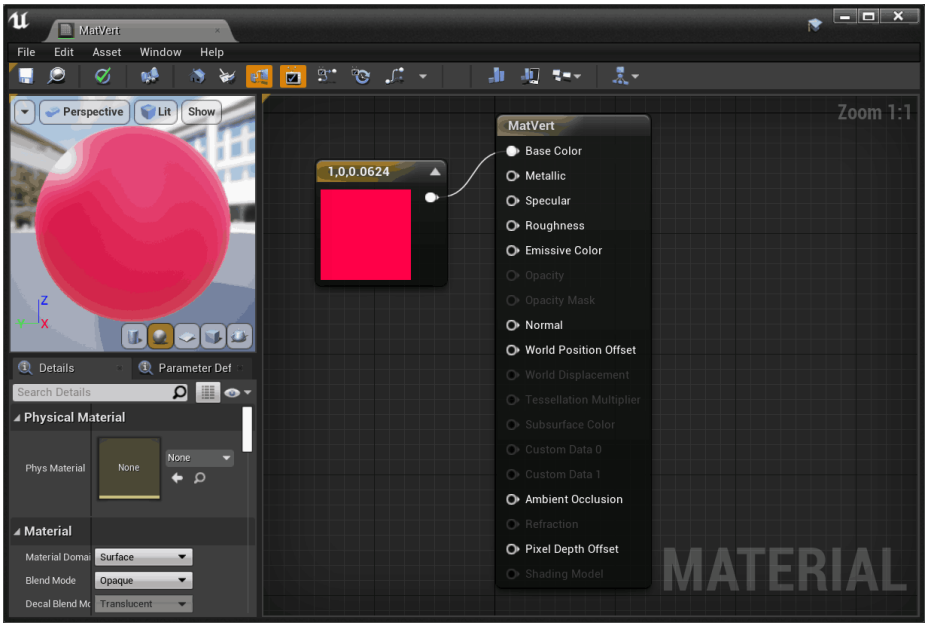


Figure 22: The Material Editor.

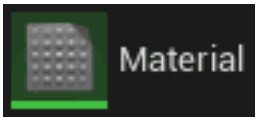
Every Material graph contains a node having various material attributes as inputs. Table 3 describes several of the most commonly used attributes.

Attribute	Description
Base Color	The color the material reflects (albedo)
Metallic	How metal-like the surface is
Specular	Color the material reflects directly back to the eye (shininess)
Roughness	Scattering amount (diffuse)
Normal	Orientation of surface (tangent plane)
World Position Offset	Displacement of vertex position (in world coordinates)

Table 3: Description of various material attributes. Most attributes are concerned with the pixel stage of the shader pipeline, except for the position offset which occurs in the vertex stage.

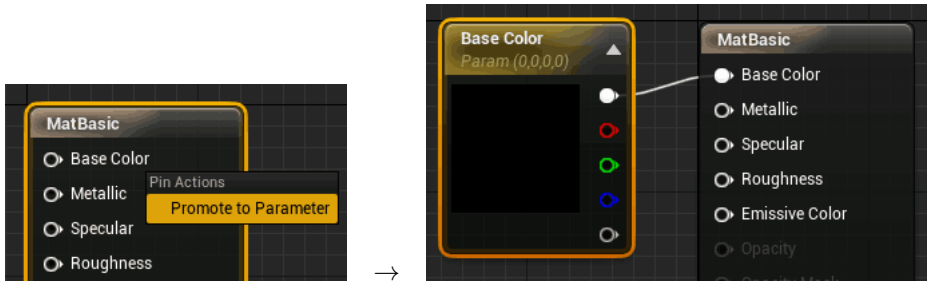
8.2 Our First Material

To get a better understanding of the Material Editor we will create a very basic material with a controllable color. We will also learn about Material Instances and how they differ from Materials. Begin by pressing "Add New" in the Content Browser and then select "Material". Materials are designated by a green icon:

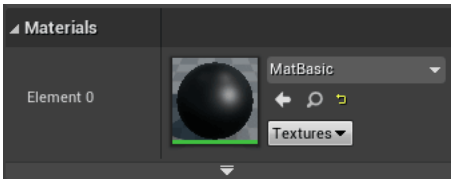


Rename the new Material "MatBasic" and then double-click it to open it in the Material Editor. You should see the graph editor with a single tall node in the center as shown in Figure 22. The inputs derive from Unreal's physically-based rendering which aims to produce more realistic lighting with more intuitive and fewer parameters. More details on each property, including numerical values

for real-world materials, can be found online [24]. Next, we will expose a *parameter* so that we can edit the material's color interactively. A parameter works very much like a public variable would in Blueprints. Right-click the `Base Color` outlet and select `Promote to Parameter`. This should create a new color picker node connected to the `Base Color` inlet.

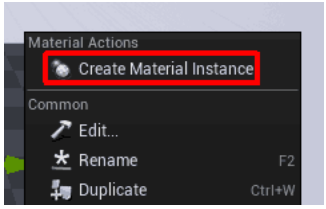


We will now learn how to assign our material to a mesh. "Compile" the material by clicking the green check in the top menu bar and then save it. This will be our typical routine after editing a material. Go back to the level editor. Drag a sphere into the scene and then go over to its Details and find its Materials property. Click on the drop-down box and enter "MatBasic" into the search field. Select MatBasic from the result.



If done correctly, the sphere should turn black since that is currently its default color. Try going back to the MatBasic program and changing its Base Color to blue. Compile and save. You should now see the sphere in the scene turn blue. Drag another sphere into the scene and assign it the MatBasic material. You should now have two blue spheres. What this tells us is that any meshes with a particular Material assigned to them will reflect any changes to the underlying Material. This is useful, however, there are a couple problems: 1) every change to a Material requires a recompile of its underlying shader code and 2) all assignees of the Material share the exact same material. We would like to have instances of a Material with their own state, just like we do with Blueprint classes.

Recall the parameter we created in the MatBasic material. This acts just like a member variable of a class. Unlike Blueprints, we must explicitly create instances of our Material. Things are set up this way as multiple meshes will likely share the same material. To create a Material instance, right-click on MatBasic in the Content Browser and select "Create Material Instance".



Rename the new Material Instance "MatBasicInst". Select one of the spheres in the scene and change its material to MatBasicInst. Double-click the MatBasicInst asset and you will be presented with yet another editor! This editor is a simplified version of the Material Editor with only a preview and Details pane (Figure 23). You might call this a Material Interface since we cannot edit the underlying Material program, only tune its exposed parameters.

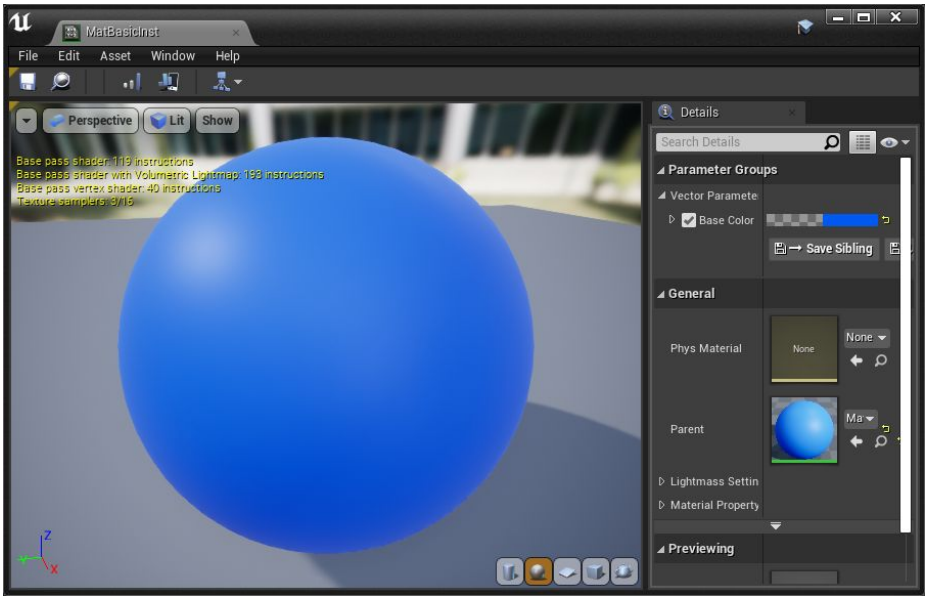


Figure 23: The Material Instance interface that allows modification of material parameters that do not require a recompile of the underlying shaders.

In the Details pane, you will see the Base Color parameter we exposed. Tick the box next to the variable name and change the color to red. Save the material instance and go back to the level editor. You should now see a blue sphere (using MatBasic) and a red sphere (using MatBasicInst) (Figure 24).

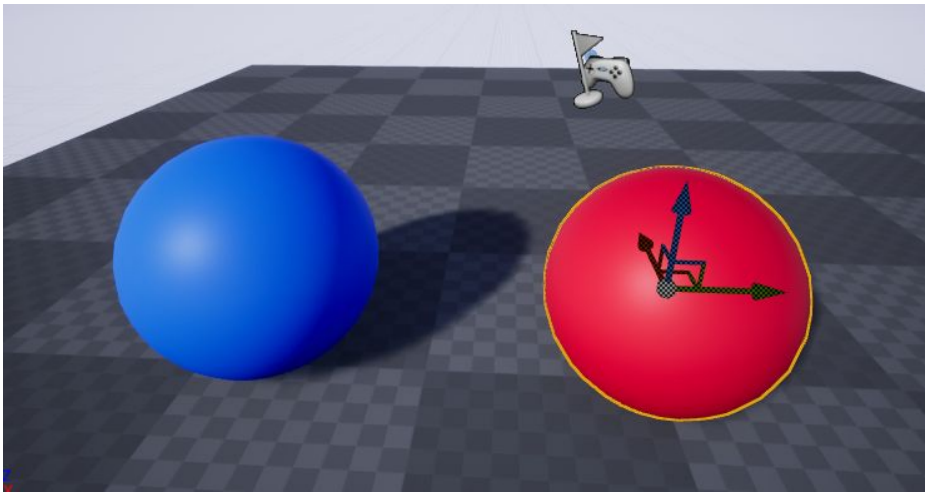


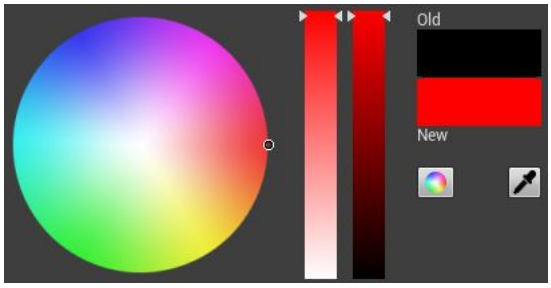
Figure 24: A Material Instance (right) with a parameter modified from its base Material (left).

To summarize, a Material represents a programmable shader and a Material Instance represents an interface to the parameters exposed by the shader. Editing a Material is "deep" in that it must recompile HLSL shader code, which is not interactive. Editing a Material Instance is "shallow" in that no code needs to be recompiled and we can see changes immediately (in real-time). As an analogy, a Material is like a synthesizer or plugin and a Material Instance is like a preset.

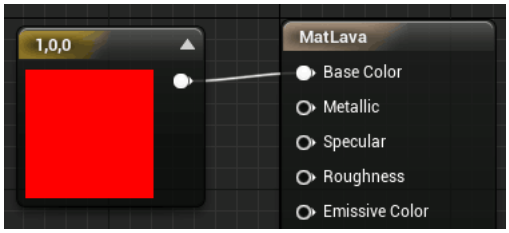
8.3 Pixel Shader: Lava Ball

The job of the pixel shader is to define a color for every pixel that will be seen on screen. A pixel shader is analogous to a "fill" operation you might see in a paint program. The most common use of a pixel shader is to define how a surface (of a mesh) reacts to light. In this walkthrough, we will create a lava ball by programming a pixel processor.

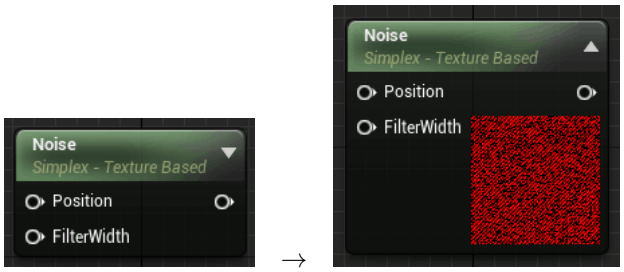
Create a new Material called "MatLava" and then double-click it to open it in the Material Editor. Create a `Constant3Vector` node and connect it to the `Base Color`. Double-click the `Constant3Vector` node to bring up a color picker. Change the color to red and click OK. Make sure to also turn the brightness up all the way as shown here:



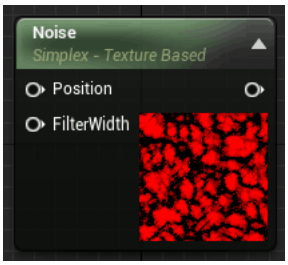
If done correctly, the node should now show red (as well as the preview shape).



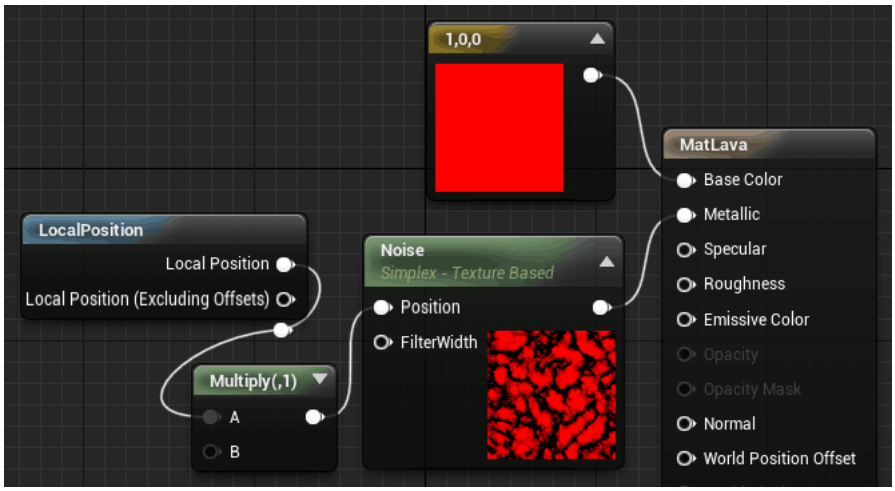
Next, we will use some noise to create a more interesting surface texture. Add a `Noise` node. The `Noise` node takes a position as input and outputs a noise (scalar) value in the range $[-1, 1]$. This type of noise is called *Perlin noise*[32], but also sometimes *value noise* or *scalar noise*. In the upper-right corner of the node, you will see an arrow. If you click it, the node expands to reveal a preview image of the noise.



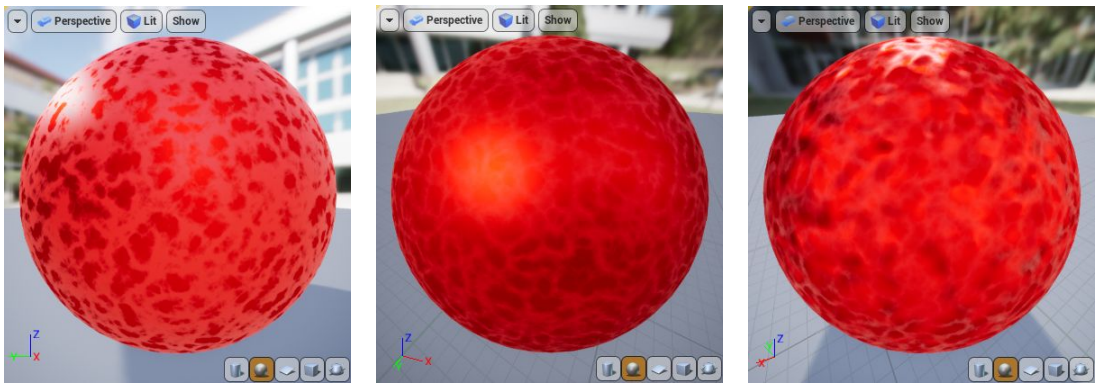
In fact, all nodes have a preview window that can toggled on and off. At the moment, the noise preview looks somewhat garbled since the noise frequency is too high. In the `Noise` node's Details panel, change its Scale to 0.05. Scale is equivalent to spatial frequency. Now you should see the noise more clearly.



Next we need a way to map the noise onto the surface of our mesh. A simple way to do this is to simply use the object-space position of the mesh vertices as input to the noise. Create a `LocalPosition` node and `Multiply` node and create the following graph connected to the Metallic input.



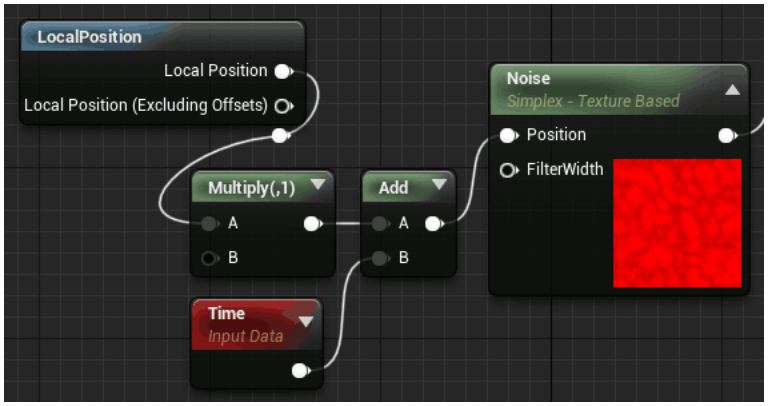
You should see a very dramatic effect in the preview window (Figure 25a). However, there are a couple problems. First, the noise outputs negative values, but Metallic should be in the range $[0, 1]$. Second, when Metallic is 0, we get a plastic appearance, which is not very lava-like. Plastics tend to have a white specular component while for metals the specular matches the base color. If Metallic is too low, our material will have a washed out look due to too much white specular. We can solve both of these problems by making the output range of the noise fall in $[\frac{1}{2}, 1]$. Fortunately, this is trivial to do. Select the `Noise` node and in its Details, set Output Min to 0.5. After this change, the surface will regain its red color and lavaness (Figure 25b).



(a) (b) (c)

Figure 25: Lava ball with (a) value noise mapped to metallic $[0, 1]$, (b) value noise mapped to metallic $[\frac{1}{2}, 1]$ and (c) bump mapping added.

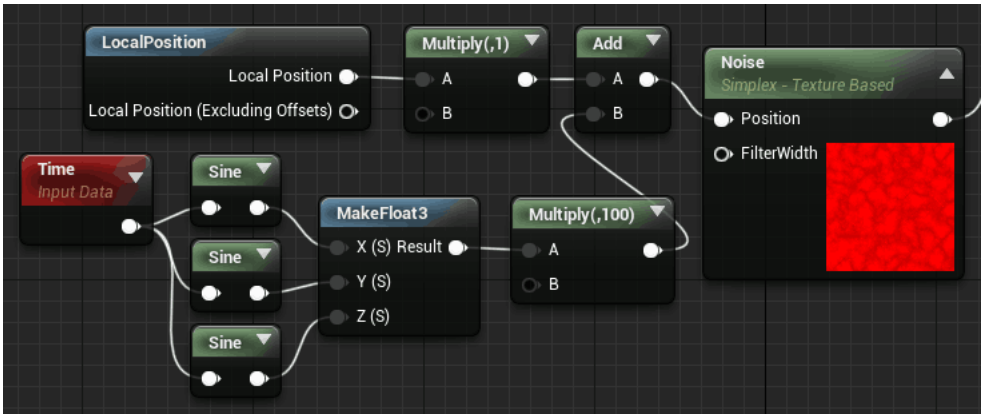
Next we will animate the lava to make it more interesting. A simple way to perform animation is to vary the position being input to the noise function over time. Create a Time and Add node and connect them as follows.



You should see the lava slowly flowing in one direction (along the vector $(1, 1, 1)$). This livens up our lava, but with a bit more work we can make it even more interesting. Instead, we will add an orbiting pattern to the position to get a more complex effect. A simple orbit that is easy to program is a Lissajous curve [47, 30, 6]. We can construct a 3D Lissajous using the parametric equation

$$\begin{aligned} x(t) &= A\sin(f_x t) \\ y(t) &= A\sin(f_y t) \\ z(t) &= A\sin(f_z t) \end{aligned}$$

where A is amplitude, f_x, f_y, f_z are frequencies and t is time. To make the Lissajous orbit, create three `Sine` nodes, a `MakeFloat3` node and a `Multiply` node and connect them into the adder as follows.



Set the periods of the `Sine` nodes (in Details) to 23.9, 25.7, and 27.1. The periods map to $1/f_x$, $1/f_y$ and $1/f_z$ in the Lissajous equation above. Set the `Multiply` value (A) to 100. You should see a far more interesting flow pattern that continually changes direction. *Technical aside*: You can be quite liberal in your use of the sine function in shaders. This is because GPUs (which run your shader) typically have dedicated hardware for trig functions that can evaluate them in a single instruction. This is in contrast to the CPU where trig functions are calculated on a more general-purpose floating-point unit (FPU) and thus, require more instructions.

One problem remains with the lava ball—the surface boundary looks overly smooth and precise. We would like a more irregular looking surface. We could add small offsets to the mesh vertices, but then the granularity of the distortion is tied to the resolution of the mesh. The more detail we want, the more vertices we need. Fortunately, there is a pixel shading trick we can use called

bump mapping that can produce a very convincing and detailed distortion that is independent of the mesh resolution. *Bump mapping* [4] works by adding small random offsets to the normals in the pixel shader (Figure 26). (Bump mapping is a special case of a more general technique called normal mapping.) By perturbing the normals in this way, we can make the surface appear bumpy since light will be reflected in different directions.

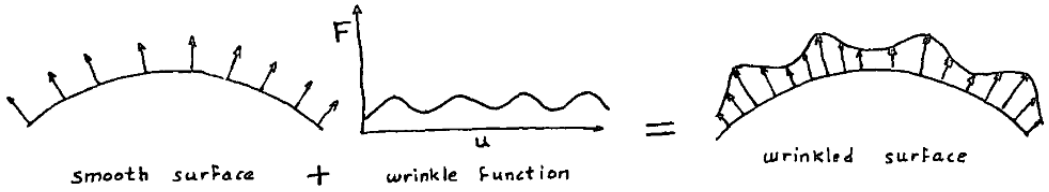
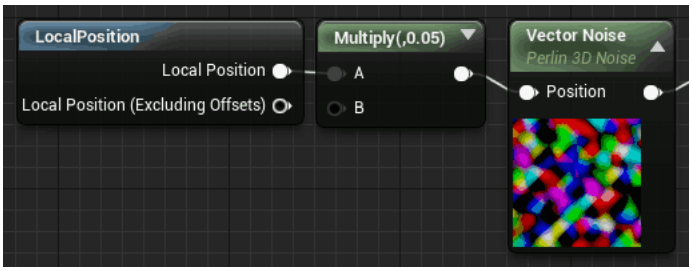
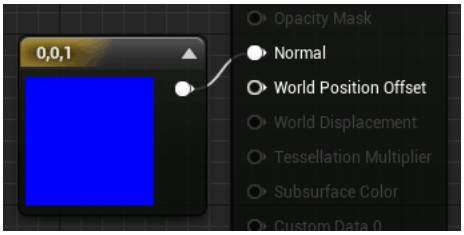


Figure 26: The bump (wrinkle) mapping concept. Positions are displaced along normals and new normals are computed based on the derivatives of the perturbed surface [4].

To implement bump mapping, we need a smooth, random vector field. Create a `VectorNoise` node and then connect a `LocalPosition` multiplied by 0.05 to its position input. `VectorNoise` has several different kinds of noise functions that may be selected from in its Details pane. The default is cell noise, which is not smooth so is not appropriate for bump mapping. Change the noise function to Perlin 3D Noise. This is the same as the value noise we used above, but rather than outputting a single scalar (float) it outputs three independent random values packed into a 3-vector. At this point, you should have a graph like the following:



The last step is to map the vector noise to the surface normal. Before doing so, we need to understand the coordinate space used for normals. In the Material Editor, normals are defined in *tangent space*, rather than, say, object or world space. Tangent space makes it easier to apply normal maps from textures which are in 2D. In tangent space, xy is tangent to the surface and z is perpendicular to the surface. That means an unperturbed normal has the value $(0, 0, 1)$ in tangent space. Test this creating a `Constant3Vector` node, setting its value to $(0, 0, 1)$ and connecting it to the Normal input of the material.



You should not see any change to the lighting. Now we will add our noise to the normal. Multiply the noise by 0.5, add it to the $(0, 0, 1)$ vector and then connect the perturbed normal to the Normal inlet. Note that we do not need to normalize the normal as Unreal does this for us automatically.

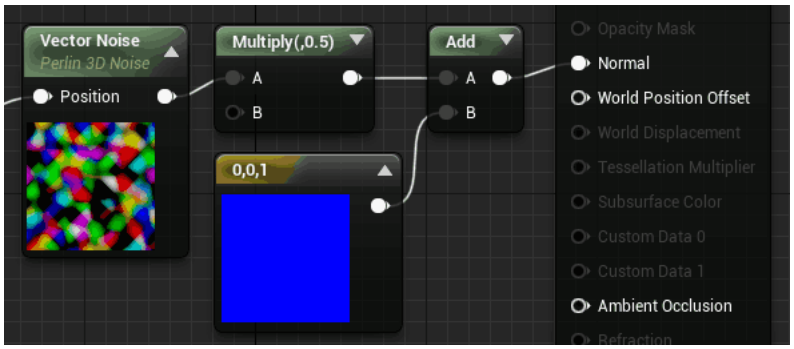


Figure 25c shows the final lava ball composite with value noise mapped to metallic and vector noise used to create bumps.

8.4 Vertex Shader: Wave

To learn how to manipulate vertices of a mesh via a Material, we will create a Material that produces a wave pattern on a flat surface. We will also learn how to create and use a custom Material Function and a little bit of calculus to compute normals.

Before proceeding, we will need an appropriate mesh for our displacement function. For this, we will import a custom tessellated plane made up of 64x64 cells lying on the xy (horizontal) plane (Figure 27).

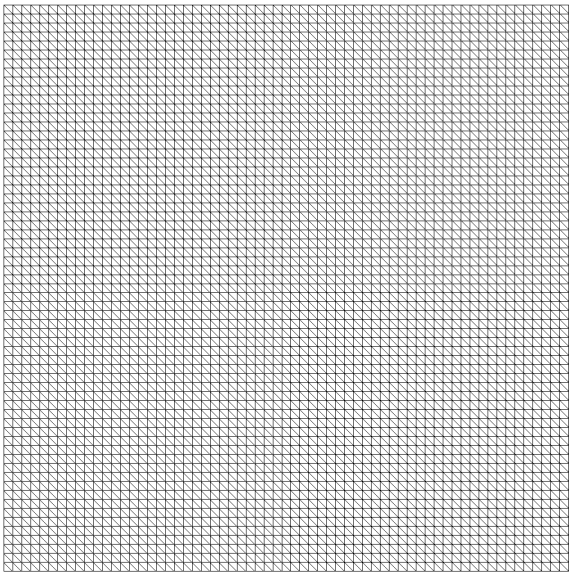
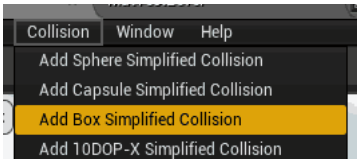


Figure 27: Wireframe view showing triangles of our tessellated plane mesh.

Drag and drop the tessellated plane object file onto the content browser. If done correctly, you should see a new icon in the content browser for the mesh.

Before proceeding, we will create a collider for the mesh to prevent the player from passing through it. Double-click the asset and then in the editor that comes up, go to the Collision menu at the top and select “Add Box Simplified Collision”.



You should then see a green outline around the mesh indicating the collision box. The editor should automatically find the best fit of the collision box around the mesh. Save and then close the mesh editor. You can try placing the mesh into the scene to confirm the collision is working properly.

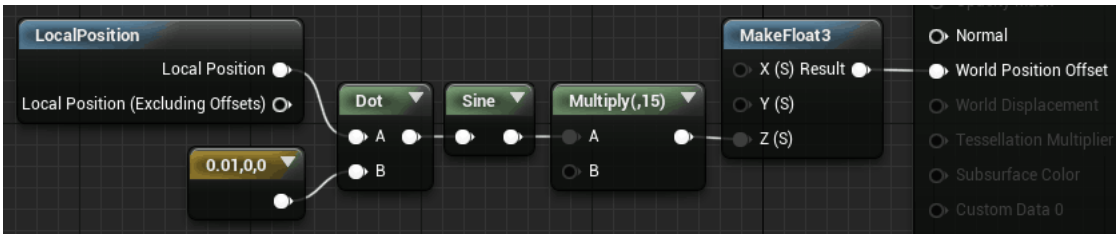
We will now go step-by-step to create a vertex displacement shader. One of the challenges with modifying vertex positions is that the normals also need to be updated in order to get correct lighting. We will first learn how to displace vertices correctly, then we will deal with normals.

Create a new Material called "MatWave" and then double-click it to open it in the Material Editor. If you like, you can use the tesellated plane mesh in the preview by ensuring it is selected in the main editor Content Browser and then by selecting the teapot icon in the Material Editor preview pane.

As our displacement function, we will use a plane wave due to its simplicity. The equation of a plane wave is given by

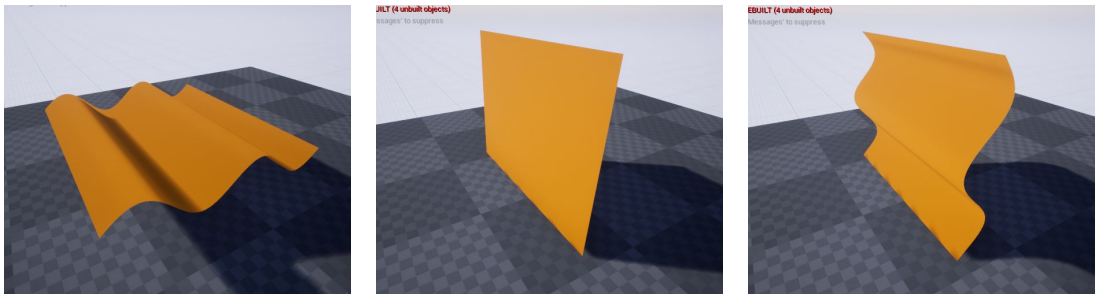
$$w(\vec{r}) = A\sin(\vec{r}\cdot\vec{f})$$

where $\vec{r}$ is position, $\vec{f}$ is frequency (wave number) and A is amplitude. Note that the frequency $\vec{f}$is a vector since it describes both a direction and magnitude. The $\cdot$ operator is a dot product defined as $\vec{r}\cdot\vec{f} = r_x f_x + r_y f_y + r_z f_z$. Construct the following graph to implement the plane wave and the save the material.



We use a `LocalPosition` node since we want the vertex displacement to be local to the object, rather than in world space. That means if we move the mesh in the scene, the vertex displacement will not change. At the end, we use the value of $w(\vec{r})$ to displace along the z (up) axis. The final output goes into `World Position Offset` which does the actual vertex displacement. There are two things to note about `World Position Offset`. First, it is in *world space* (not object or tangent space). Second, it is an offset (delta) rather than absolute position. Offsets are more convenient to work with since most of the time we want to retain the original position of the vertex and just perturb it slightly.

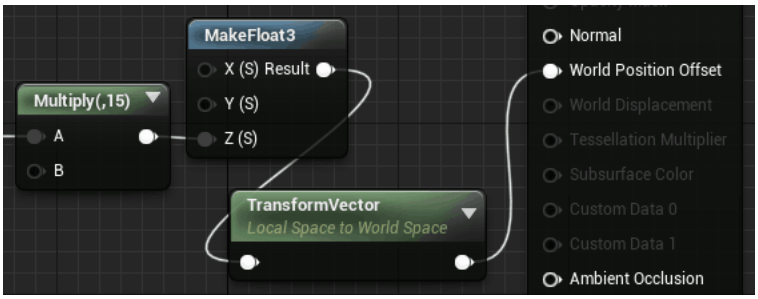
Go back to the main editor and place a tessellated plane into the scene (if you have not already) and then drag a MatWave onto the mesh. If everything was done correctly, you should see a wave pattern on the plane (Figure 28a). Drag the mesh around in the scene to confirm that the displacement pattern does not change. Now try rotating the mesh by 90 degrees along the y axis. The plane has flattened out (Figure 28b). What happened? The problem is that our displacement was in object space but we treated it like it was in world space. We need to transform it into world space.



(a) (b) (c)

Figure 28: Plane wave displacement function applied to the plane mesh: (a) without rotation, (b) with rotation, but using incorrect coordinate space and (c) with rotation using correct coordinate space.

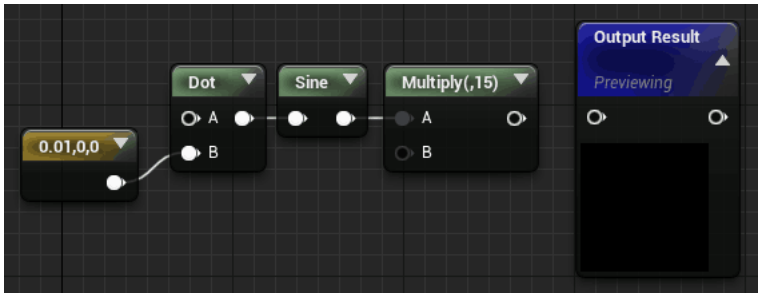
Insert a `Transform` node just before the `World Position Offset` inlet.



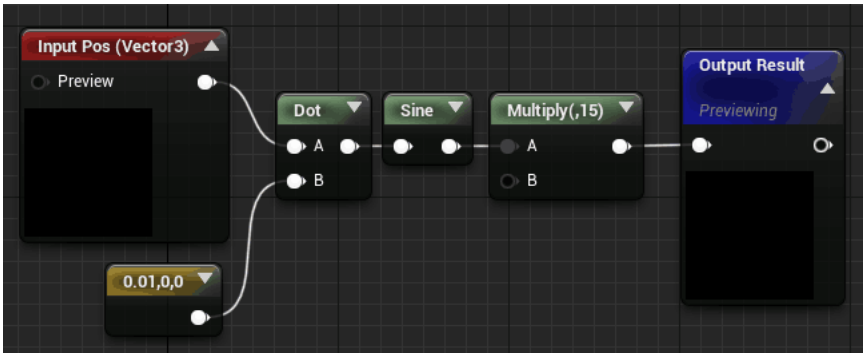
In the Details pane, change the transform to be from Local Space to World Space.

Save the material. Go back to the scene and you should see the correct displacement on the rotated plane (Figure 28c). We now have correct vertex displacement, but the normals are not correct and we can tell because the plane is not lit correctly. Before correcting the normals, we will encapsulate our plane wave displacement function so it can easily be reused. . A *Material Function* is a function that can be reused across Materials [19]. It is very similar to a Blueprints function. In programming, reorganizing code to make it easier to (re)use/maintain/etc. is known as *refactoring*.

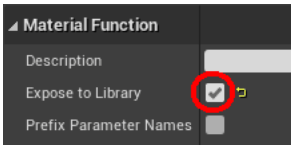
In the Content Browser, create a new Material Function by selecting Add New > Materials & Textures > Material Function. Name the new function "PlaneWave" and double-click it to open the Material Editor. You should see a single node called `Output Result`. A Material Function can have any number of inputs and outputs. We will add more as we go. Copy the following network from MatWave into PlaneWave



Create a `FunctionInput` node and in its Details rename it to "Pos". Ensure its type is Vector 3. For each input we want for our function, we must create a new `FunctionInput` node (this works differently than Blueprints). We will expand our function later. Connect the `FunctionInput` and the `Output Result` nodes to the existing nodes as follows



In a somewhat odd design choice, Material Functions are not visible to other Materials by default and must be manually activated. Click a blank spot on the canvas and then in the Material Function's Details pane, check Expose to Library.



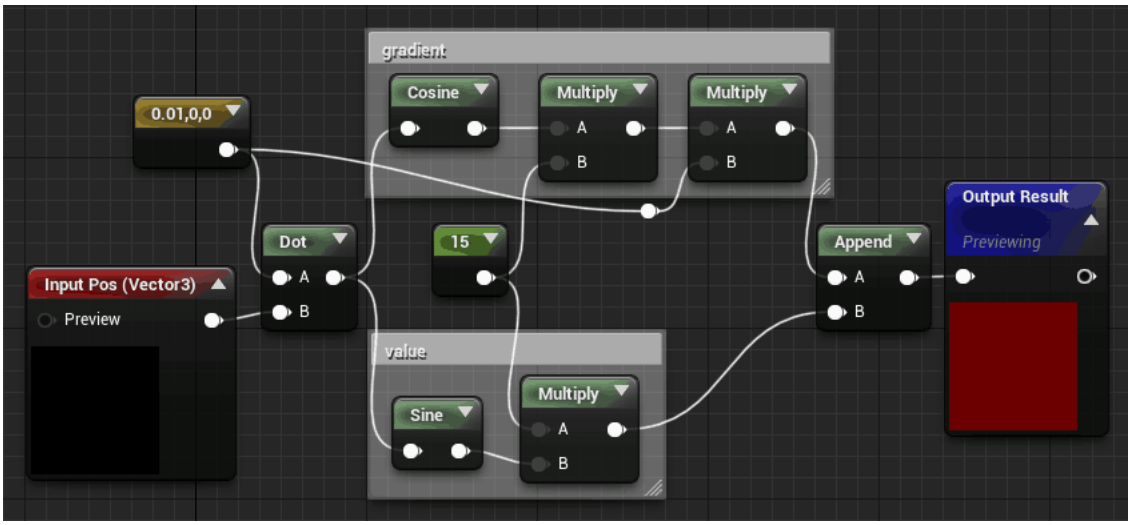
Save the Material Function. Go back to the MatWave editor and add a PlaneWave node. (It should show up in the menu if you followed the previous steps correctly.) Replace the old nodes with the `PlaneWave` node.

Save the material and ensure everything is still working in the scene editor.

Next, we will compute the correct normals needed due to the plane wave displacement. In most scenarios, normals are obtained from the derivative of a displacement function. The derivatives indicate the planes tangent to the surface and from the tangents we can derive normals. The derivatives (one along x, y and z) collectively are called the gradient of the function. A *gradient* is then a vector of partial derivatives of a function. We will modify `PlaneWave` to also return a gradient that we can use later to compute a normal. The equations of the plane wave and its gradient ∇ are

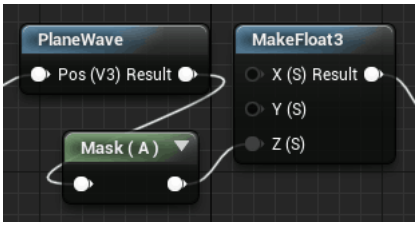
$$\begin{aligned} w(\vec{r}) &= A\sin(\vec{r}\cdot\vec{f}) \\ \nabla w(\vec{r}) &= A\vec{f}\cos(\vec{r}\cdot\vec{f}). \end{aligned}$$

You can see the equation of the gradient equation is very similar to the plane wave, which is one of the reasons we decided to use a plane wave. At this point, we will turn our `PlaneWave` into a more general-purpose unit similar to the `VectorNoise` node that outputs both a value and a gradient. Go back to the editor for `PlaneWave` and construct the following graph



Most of the changes are straightforward with a couple things to note. We pulled the amplitude A out of the `Multiply` (after `Sine`) into a `Constant` since it needs to scale both the value and gradient. The `Append(Vector)` node at the end concatenates the gradient (3-vector) and value (scalar) into a 4-vector.

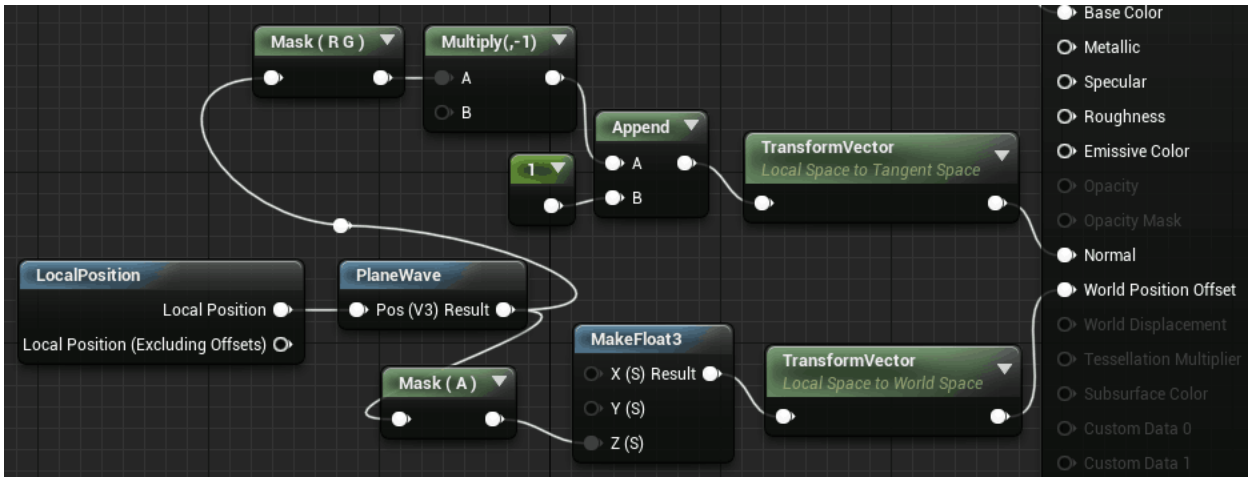
We need to edit `MatWave` to reflects the changes to the output of `PlaneWave`. Save `PlaneWave` and go back to the `MatWave` editor. PlaneWave now outputs a 4-vector, but we only want to grab the last component to do the vertex displacement. To get individual components of a vector, we need to use `Mask(Component)`. Create a `Mask(Component)` node and in its Details uncheck all components except `A`. Insert the `Mask` between the `PlaneWave` and `MakeFloat3` nodes.



Save the material and ensure the wave still looks right in the scene editor. We made somewhat of a detour, but now we are ready to compute the normals. Since we are displacing on a 2D (xy) plane, our normal will be proportional to the vector

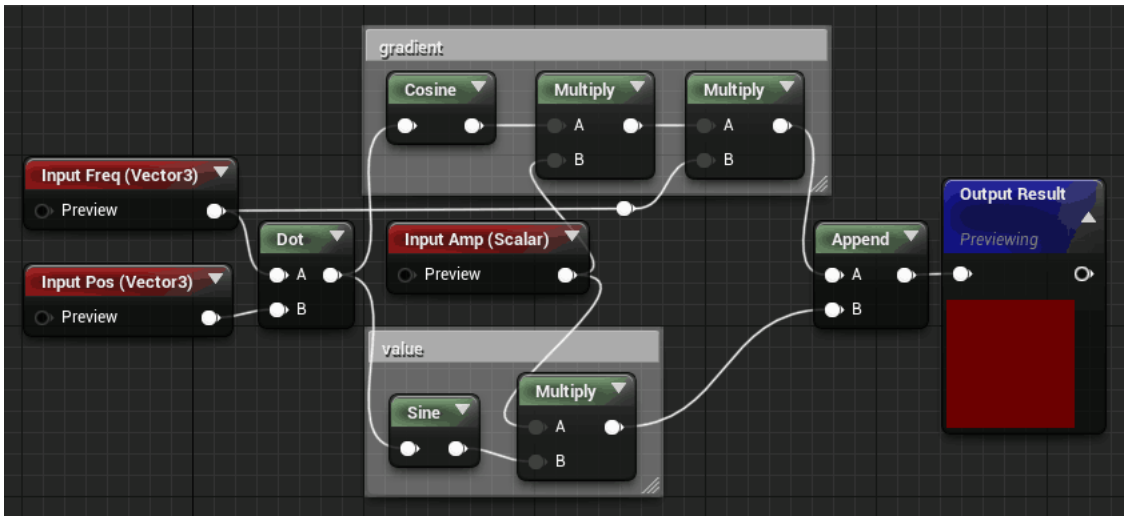
$$n = (-\nabla_x, -\nabla_y, 1)$$

where ∇_x and ∇_y are the x and y components of the gradient. The derivation is $n = t_x \times t_y$ where t_x and t_y are tangent vectors to the surface given by $t_x = (1, 0, \nabla_x)$ and $t_y = (0, 1, \nabla_y)$. The normal is found by taking the cross product $\times$ of the two tangent vectors (a cross product produces a vector perpendicular to its operands). Perform the normal calculation by modifying the graph to the following:



Here, we `Mask` RG since we just want the xy components of the gradient. Before passing the final result to the Normal input, we must transform it from local to tangent space. Note we are passing a scaled normal (n from above) to the output, not a normalized (unit) vector. This is okay since the engine automatically normalizes the normals before it does the lighting calculations.

Now that our `PlaneWave` displacement is working, we will make it a bit more flexible by exposing inlets for the frequency and amplitude. In the `PlaneWave` graph, add a new `FunctionInput` node and change its name to 'Freq' and its input type to Vector 3. Replace the existing frequency constant with this node. Repeat this process for the amplitude. Name the input 'Amp' and ensure its type is Scalar. The final `PlaneWave` function should look like the following:



Lastly, we need to configure a couple things about the inputs to control how they behave in the Material Editor. First, we want the inlets sorted from top to bottom as Pos, Freq and Amp. The ordering can be specified using the Sort Priority attribute of `FunctionInput`. Smaller numbers mean higher priority. The default is 0 which means highest priority. Change the Sort Priority of Freq to 1 and of Amp to 2 so they come after Pos.

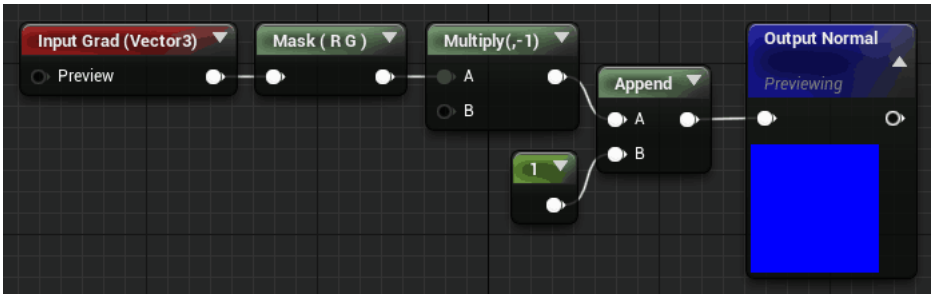
Second, we want the inputs to have default values, otherwise, if nothing is connected to the input,

we will get an error. For Freq and Amp, set their default values (x) to 0.01 and 10, respectively. Then, for each input, set its attribute Use Preview Value as Default to true.

Save the function and then check that the wave displacement is still working in the scene. Here are some things to try next:

- Animate the `PlaneWave` by altering its input position.
- Map the value output of the `PlaneWave` to the base color. Use a `LinearInterpolate` (`Lerp`) node to interpolate between two colors.
- Use a linear combination (sum) of two `PlaneWaves` to create a more complex pattern. Make sure the two `PlaneWaves` have different frequency vectors, for example, (0.01, 0, 0) and (0, 0.01, 0). To combine the plane waves, simply add their outputs and then map the sum to the position offset and normal.

As a final exercise, encapsulate the gradient to normal calculation in a Material Function so it can be reused in other Materials. This will help reduce clutter in the displacement Material graphs. You can use something like the following graph



8.5 HLSL Programming

Shader programming in UE is based on the HLSL shading language. UE defines its own C-like layer on top of HLSL to enable cross-platform compatibility. UE shader source files are located in `/Engine/Shaders/`[1]. The files are divided into `.ush` and `.usf` which stand for, respectively, Unreal Shader Header and Unreal Shader File. UE shaders are typically divided into vertex and pixel (fragment) shaders. UE also has global shaders (`FGlobalShader`) that operate in screen space and are used primarily for post-processing effects.

HLSL is very much like the C programming language and therefore much of the syntax for declarations, conditionals and loops is the same. HLSL defines some basic types as follows

```
float       // 32-bit floating point value (real number)
float2      // 2-vector of floats (UV coords, etc.)
float3      // 3-vector of floats (positions, normals, colors, etc.)
float4      // 4-vector of floats (homogenous points/vectors, quaternions)
float3x3    // 3x3 matrix of floats (rotation matrices, etc.)
float4x4    // 4x4 matrix of floats (poses, model-view, projection, transforms)
bool        // boolean (true or false)
int         // 32-bit signed integer
uint        // 32-bit unsigned integer
half        // 16-bit floating point value
double      // 64-bit floating point value
```

[1]The full path on Windows is typically `C:\Program Files\Epic Games\UE_4.xy\Engine\Shaders\User`.

More on HLSL can be found in the resource list at the end of this section.

Before getting started with shader programming, you will want to enable shader debugging features. Do this by uncommenting the following line `/Engine/Config/ConsoleVariables.ini`

```
r.ShaderDevelopmentMode=1
```

8.5.1 Material Shaders

To create a material shader using custom HLSL code, first create a new material in the editor. Double-click the material to open the material editor. Right-click an empty space on the grid and create a `Custom` node. Click on the `Custom` node and you will notice several fields than can be edited. The most important for now are `Code` and `Output Type`. `Output Type` is the type returned by the expression. By default it is a `CMOT Float 3` (3-vector) that can represent an RGB color. In the `Code` field is where you can place your custom shader code. By default, it has the value 1 which means it returns `float3(1,1,1)`. Figure 29 shows the material editor window with the Custom node details pane.

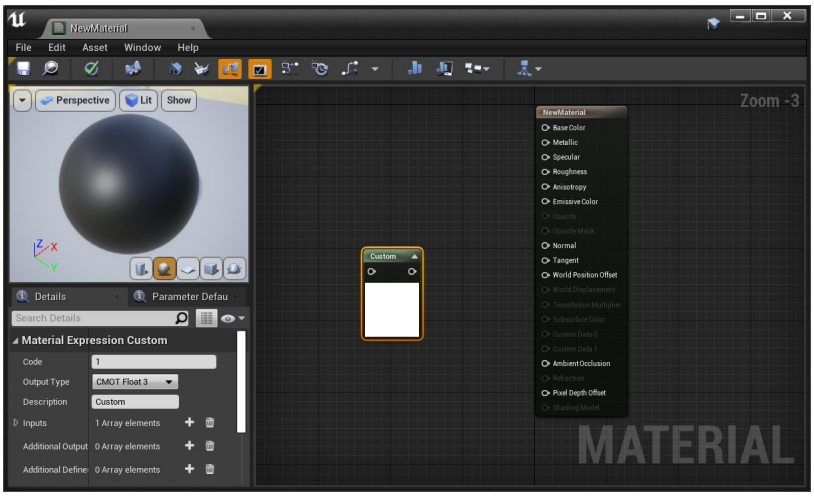


Figure 29: Material editor window with a Custom node. The upper left pane show a preview of the material and the lower left pane shows details of the selected node (here the Custom node).

In the Custom node `Code` field, enter the following code and press enter

```
float3 col = float3(1., 0., 0.);
return col;
```

Note that enter compiles the code. If you want to create a new line press shift-Enter. Connect the outlet of the custom node to the `Base Color` of the material. You should now see a red material applied in the preview (Figure 30a).

To see all of the active HLSL code for the material, select `Window > Shader Code > HLSL Code`. The code view in UE is quite limited, so copy and paste the code into an external editor so you can search it. Your custom expression gets called in the function `CalcPixelMaterialInputs` (abbreviated here for clarity)

```
void CalcPixelMaterialInputs(in out FMaterialPixelParameters Parameters, in out
    FPixelMaterialInputs PixelMaterialInputs){
    // Bunch of code here...
    MaterialFloat3 Local1 = CustomExpression0(Parameters);
    // Bunch of code here...
    PixelMaterialInputs.BaseColor = Local1;
}
```

and `CustomExpression0` looks something like

```
MaterialFloat3 CustomExpression0(FMaterialPixelParameters Parameters){
    float3 col = float3(1., 0., 0.);
    return col;
}
```

Notice that the `Parameters` variable is passed into your custom expression (by value). From this variable you can access a variety of geometric information useful for shading in your custom expression. For example, try changing the expression to the following to color the mesh based on its world space normals

```
float3 col = abs(Parameters.WorldNormal);
return col;
```

If successful, you should see the shape colored according to its normals (Figure 30b). Notice that the color is red along x, green along y and blue along z. This is because colors are specified as 3-vectors (red, green, blue).

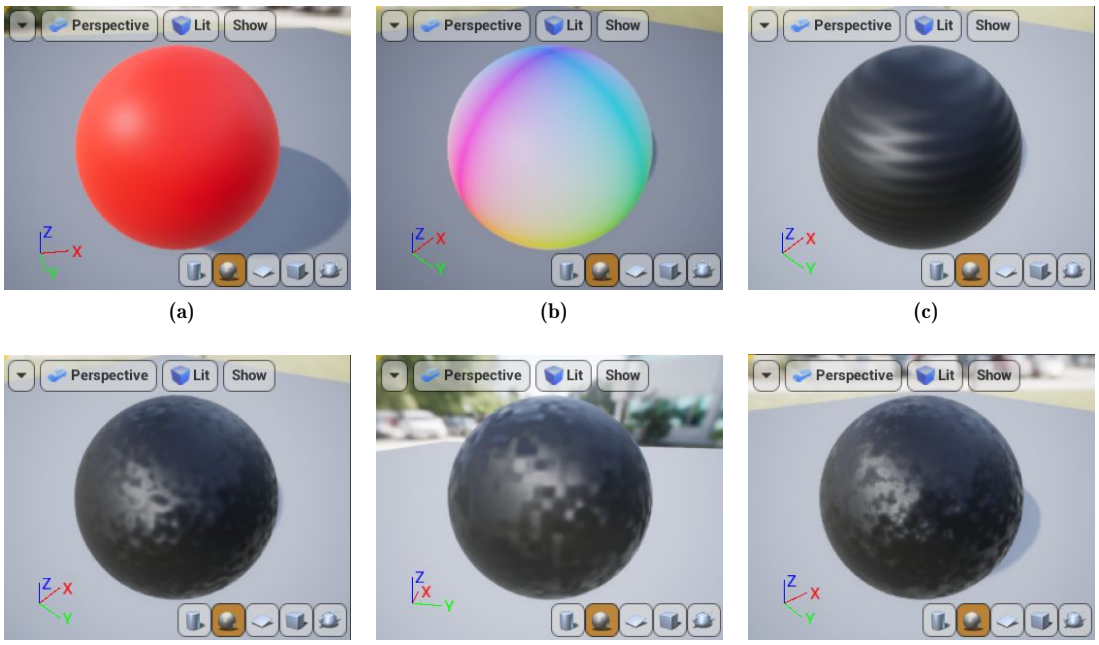


Figure 30: Custom shader code: (a) red base color, (b) world normal mapped to base color, (c) sinusoidal wave mapped to normals, (d) gradient noise mapped to normals, (e) value noise mapped to normals, (f) three octaves of value noise mapped to normals.

The `FMaterialPixelParameters` struct has many members, some of the more useful ones being

```
struct FMaterialPixelParameters{
    half4 VertexColor;              /* Vertex color in mesh */
    half3 WorldNormal;              /* Vertex normal, in world space */
    half3 WorldTangent;             /* Vertex tangent, in world space */
    half3 ReflectionVector;         /* Relection unit vector, in world space */
    half3 CameraVector;             /* Surface to camera unit vector */
    float4 ScreenPosition;          /* Normalized device coordinate */
    half TwoSidedSign;              /* Sign for two-sided materials */
    half3x3 TangentToWorld;         /* Rotation matrix (for normal mapping) */
    float3 AbsoluteWorldPosition;   /* Position of pixel, in world space */
};
```

Next, we will generate a normal map in a shader using mathematical functions. Begin by creating another Custom node. Enter the following into the code field

```
float3 N = float3(0., 0., 1.);
return N;
```

Connect the node output to the material `Normal` inlet. You should not see any change to the lighting as this is a pass-through shader. The returned vector represents the surface normal in tangent space where the x,y coordinates are tangent to the surface and the z coordinate is perpendicular to the surface. (It is also possible to work with normals in world space. To do so, click on the material and then in the details panel click the eye icon in the upper right corner and check the"Show All Advanced Details" option. A "Tangent Space Normal" option will now appear under the "Material" category.) Now, edit the code to

```
float3 N = float3(0.,0.,1.);
N.x += 0.1*sin(0.5*Parameters.AbsoluteWorldPosition.z);
// N = normalize(N); // not necessary as Unreal normalizes automatically
return N;
```

You should see a ripple pattern going from top to bottom (Figure 30c). What we are doing is displacing the normal by a plane wave going along the z axis. The amplitude of the wave is `0.1` and its spatial frequency is `0.5` radians/cm. Note that usually after modifying a normal, you normalize it to ensure it has unit magnitude. We do not normalize it here since Unreal does it for us automatically before the lighting pass. Now change the code to the following

```
float3 N = float3(0.,0.,1.);
N += 0.1*GradientNoise3D_ALU(0.1*Parameters.AbsoluteWorldPosition, false, 1.);
return N;
```

Now you should see a bumpy surface (Figure 30d). *Gradient noise* is a type of 3D noise that varies smoothly across space. You can change the frequency of the noise by scaling the first argument (a float3 position) to `GradientNoise3D_ALU`. ALU stands for arithmetic logic unit and indicates that the noise is computed entirely from arithmetic operations—no lookup tables/textures are used. The second argument specifies whether to tile the noise and the third argument is the period. `ValueNoise3D_ALU` is another type of noise available. *Value noise* is a smoothly-varying noise that interpolates between random values on a cubic lattice. Value noise has obvious gridding artifacts not present with gradient noise (Figure 30e). However, value noise is typically more efficient to compute than gradient noise. For more realistic noise you can add several octaves of noise together (Figure 30f). For good results when adding patterns together, you can set the amplitude of each pattern inversely proportional to its frequency. You can see all the available types of noise in `/Engine/Shaders/Private/Random.ush`.

Lastly, we will animate the sinusoidal normal map using the `View.RealTime` variable. Modify the previous code to

```
float3 N = float3(0.,0.,1.);
float phase = 0.5*Parameters.AbsoluteWorldPosition.z;
phase += 2.*View.RealTime; // propagate wave crests at 2 radians/sec
N.x += 0.1*sin(phase);
return N;
```

You should now see the wave moving across the surface. `View.RealTime` gives the current game time in seconds.

8.5.2 Using External Files

The `Code` field is not very convenient for editing, so in practice you will likely want to `#include` a shader file here and edit the code in an external code editor. In previous versions of UE, it was as simple as adding a `Shaders` directory to your project directory and then `#include "/Project/-MyShader.usf"`. However, in current UE, these "virtual directories" (i.e. search paths) must be

manually configured and unfortunately this turns out to be quite a pain. Alternatively, we can put our shaders in an engine directory that is already searched by the compiler. The upside is that now our shaders can be used across multiple projects. The downside is that these shaders must be in the engine source for our project to compile.

We now describe how to add your own shaders to the engine through a simple example. Go to `/Engine/Shaders/` on your hard drive and add the directory `User`/. Next create a new file `Test.usf` in `User/` with the following contents

```
return float3(1., 0., 0.);
```

Now go back to the UE4 material editor and add the following to the `Code` field of a custom node:

```
#include "/Engine/User/Test.usf"
return 1;
```

Note the extra `return` statement at the end. This is required so that UE correctly interprets the code as a function body. You can follow this same procedure to add any custom shader source to the engine.

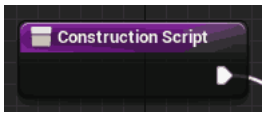
8.5.3 Resources

- UE Shader Development (Epic): https://docs.unrealengine.com/en-US/ProgrammingAndScripting/Rendering/ShaderDevelopment/index.html
- UE Material Expressions (Epic): https://docs.unrealengine.com/en-US/RenderingAndGraphics/Materials/ExpressionReference/Custom/index.html
- UE Post-processing Shader Tutorial (Tran): https://www.raywenderlich.com/57-unreal-engine-4-custom-shaders-tutorial
- Writing HLSL Shaders (Microsoft): https://docs.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl-writing-shaders-9
- HLSL Syntax (Microsoft): https://docs.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl-language-syntax
- HLSL and UE4 Dev Blog: https://shaderbits.com/blog

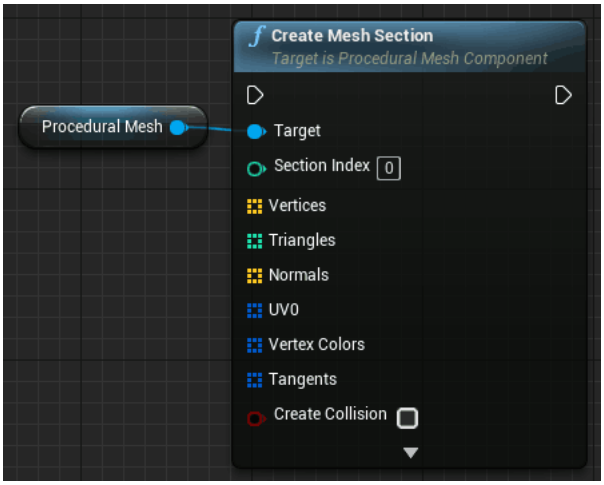
9 Procedural Mesh

In this tutorial, we will learn how to generate a mesh from scratch by calculating its vertex positions and normals and its triangle faces (indices). In addition, we will learn how to use arrays and for loops.

Begin by creating a new `Actor` Blueprint class and name it "ProcMesh_BP". Drag an instance into the scene and raise it up a bit so it is out of the floor. Open the Blueprint and add a `Procedural Mesh` component to it. Since we only want to generate the mesh once when a new instance is created, we will edit the Blueprint Construction Script rather than the usual Event Graph. By default, the Construction Script supplies a single node of the same name that fires when a new instance of the class is created. In this way, a Construction Script is analogous to a *constructor* in OOP.



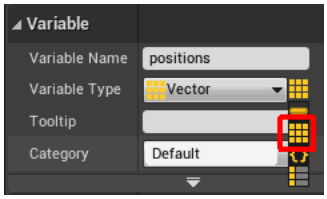
The key component we will use to create mesh vertex data is a node called `Create Mesh Section`. Add a `Create Mesh Section` node and ensure the `Procedural Mesh` component is connected to its target.



Notice three key inlets: Vertices, Triangles and Normals. The square graphic next to these indicate that they are arrays. Vertices is an array of vertex positions, Normals is an array of vertex normals and Triangles is an array of triangle indices. We will need to generate each one of these arrays in order to create a solid, properly lit mesh. Our approach will be to create a tessellated plane mesh and then warp it into a specific shape.

First, we will create some necessary variables. Our tessellated plane will need two variables to define its resolution along the x and y directions. Create two integer variables called 'Nx' and 'Ny' and set their default values to 64. Next, we will need two arrays of 3-vectors to store the positions and normals of the mesh. Create two array variables of type Vector called 'positions' and 'normals'.

To make a variable an array, go to its Details panel, click the drop-down on the right of Variable Type and select the grid icon to make it an Array.



Next, we will populate the array of vertex positions. To do this, we will need to use two loops: one to go along x and another to go along y. Since the array is one-dimensional and the plane is two-dimensional, we need to determine an order to place the vertices in the array. A common way to do this is to first move along columns (x) and then along rows (y). Figure 31 shows an example 3x3 tessellated plane showing the 1D and 2D indices of each vertex.

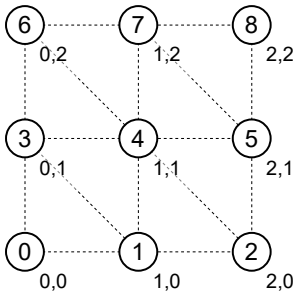
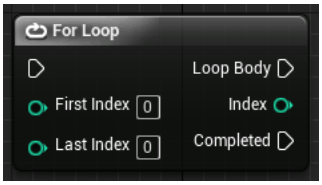
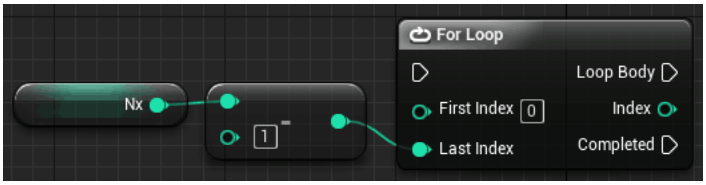


Figure 31: Coordinates of vertices of a 3x3 tessellated plane. Circles represents vertices. The number in each circle is the vertex's one-dimensional array index. The number under the vertex is its two-dimensional index.

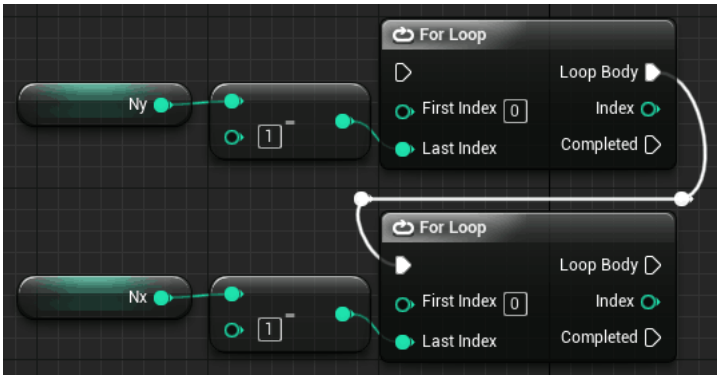
Create a `For Loop` node and examine its pins. We will need to understand all of them.



The loop will output indices going from the First Index to the Last Index, inclusive. For example if the first and last indices are 0 and 3, respectively, the output will be 0, 1, 2, 3. The Loop Body outlet fires for each new output index (or loop iteration) and the Completed outlet fires when the loop is finished. Connect the Nx and Ny variables to the loop as follows



This loop will output indices from 0 to Nx-1. Now we need a second (outer) loop to move along y. Create a second `For Loop` and connect it as follows



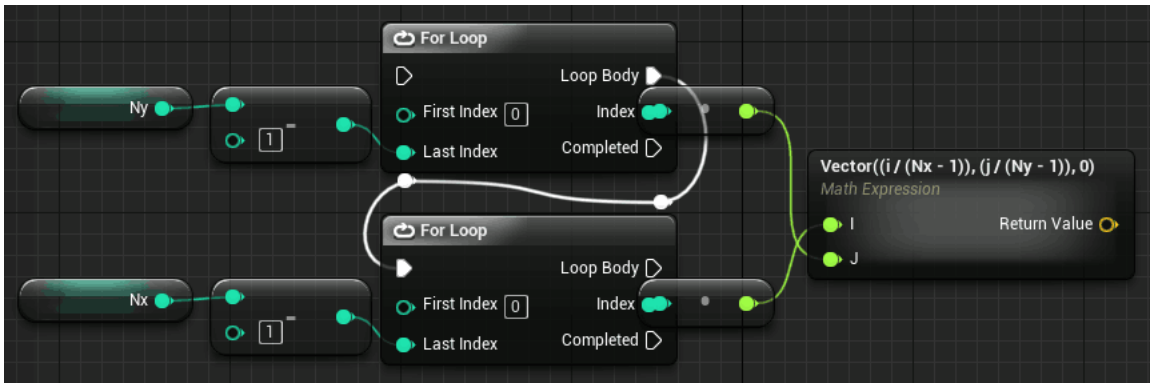
The first (inner) loop gets triggered on each iteration of the second (outer) loop. This will produce the two-dimensional indices of the plane vertices in the order illustrated in Figure 31. Now, we simply need to map the vertex indices into positions and add the position to the positions array. To map the 2D index (i, j) to a position, we use the mapping

$$(i, j) \to \left(\frac{i}{N_x - 1}, \frac{j}{N_y - 1}, 0 \right)$$

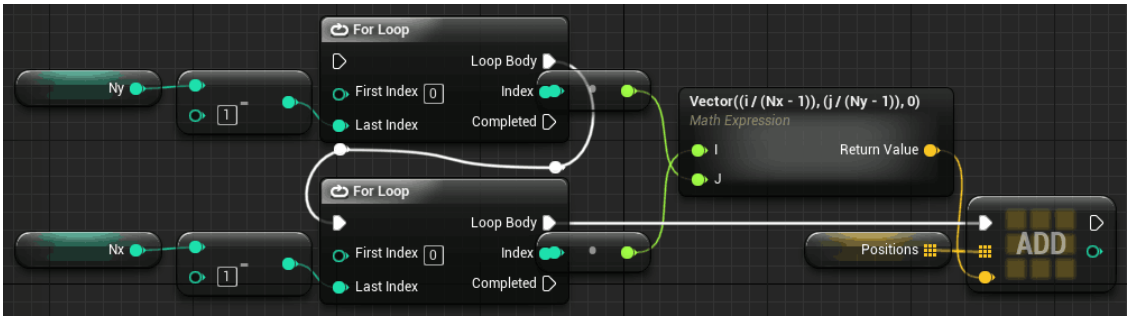
To implement the mapping create a Math Expression with the following expression

```
Vector(i/(Nx-1), j/(Ny-1), 0)
```

Connect the two loop index outputs to the Math Expression ensuring that x indices go to i and y indices to j.

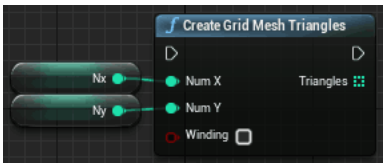


Drag off the output of the math expression and create the `Add` node you find under Utilities > Array. This node is used to add new elements to an array. Connect the positions array to its input. We also need to connect the Loop Body output of the inner loop to `Add`'s Exec input.

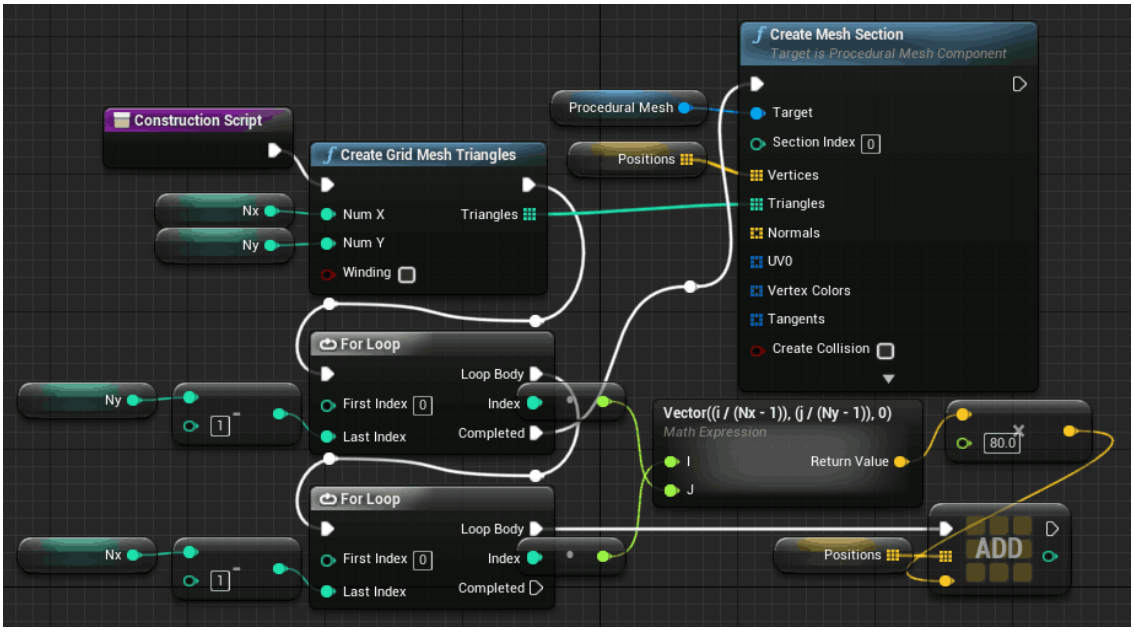


One thing about Add Mesh Section is that we must hand it both Vertices (positions) and Triangles (indices) in order for it to work. We will next create the triangle indices and then connect everything up.

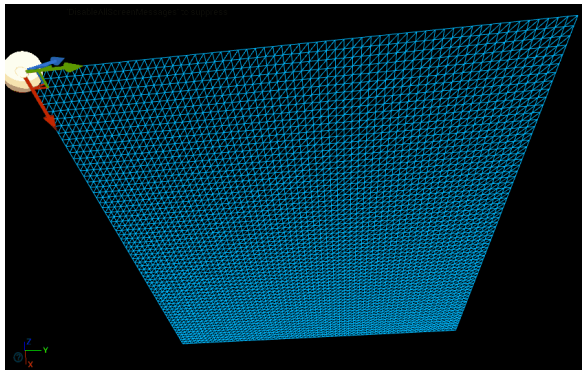
Creating triangle indices would normally be somewhat tricky in Blueprints, but fortunately there is built-in node that will do it for us. Add a `Create Grid Mesh Triangles` node and then connect the Nx and Ny variables to its inlets.



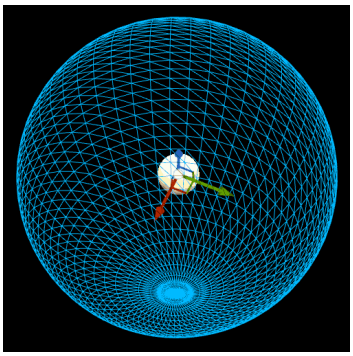
The `Create Grid Mesh Triangles` node will create an array of indices where every three indices define a triangle. For the 3x3 tessellated plane in Figure 31, the first few triangle indices would be something like 0,1,3, 3,1,4, 1,2,4, 4,2,5. Finally, connect all the loose pieces together to get something like the following graph



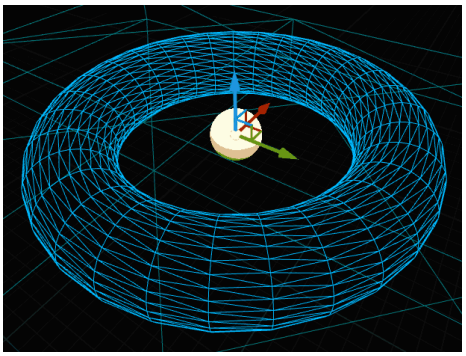
Note that there is an extra multiplier before adding the vector to the positions array. This is only here temporarily for testing. Play the level and you should see a flat plane. You may have to move the actor up since the front face is on the bottom. Change to wireframe view and ensure the triangles look good (Figure 32a). You may want to hide the Sky Sphere to see the mesh more clearly.



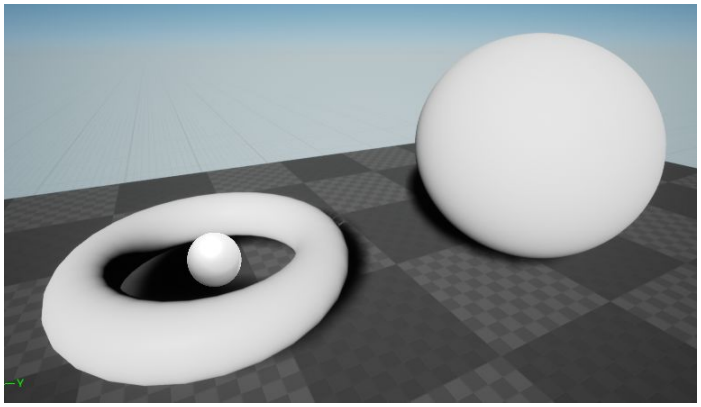
(a)



(b)



(c)



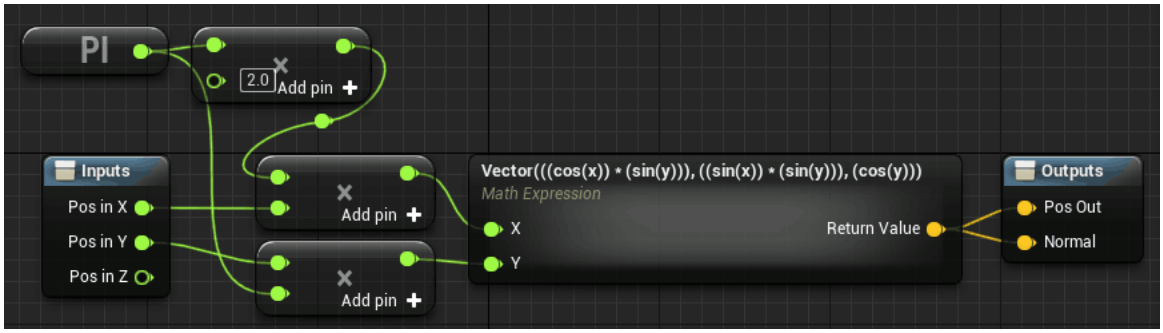
(d)

Figure 32: Wireframe views of our procedurally generated (a) tessellated plane (b) sphere and (c) torus and (d) solid sphere and torus shapes.

The next step is to bend the plane into different shapes by mapping the 2D positions of the plane into 3D positions of the shape's surface. We will create a sphere and a torus. For a sphere, we will employ the spherical to Cartesian mapping

$$\begin{aligned} x &= \sin(\phi)\cos(\theta) \\ y &= \sin(\phi)\sin(\theta) \\ z &= \cos(\phi) \end{aligned}$$

where θ is in $[0, 2\pi]$ and ϕ is in $[0, \pi]$. Create a new macro in the Blueprint called "mapSphere" and construct the following graph

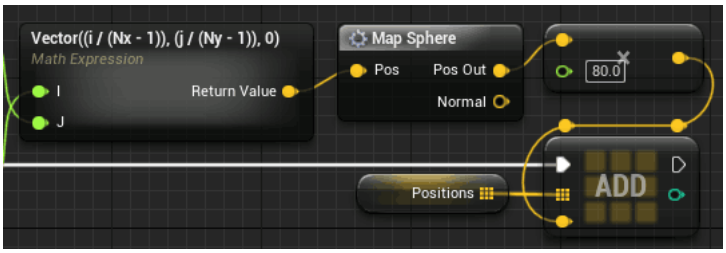


The macro has one input and two outputs. The 'Pos' input is a Vector that should have its x and y components moving in $[0, 1]$. The 'Pos Out' and 'Normal' outputs are also Vectors and together provide the vertex data of the sphere.

The math expression is

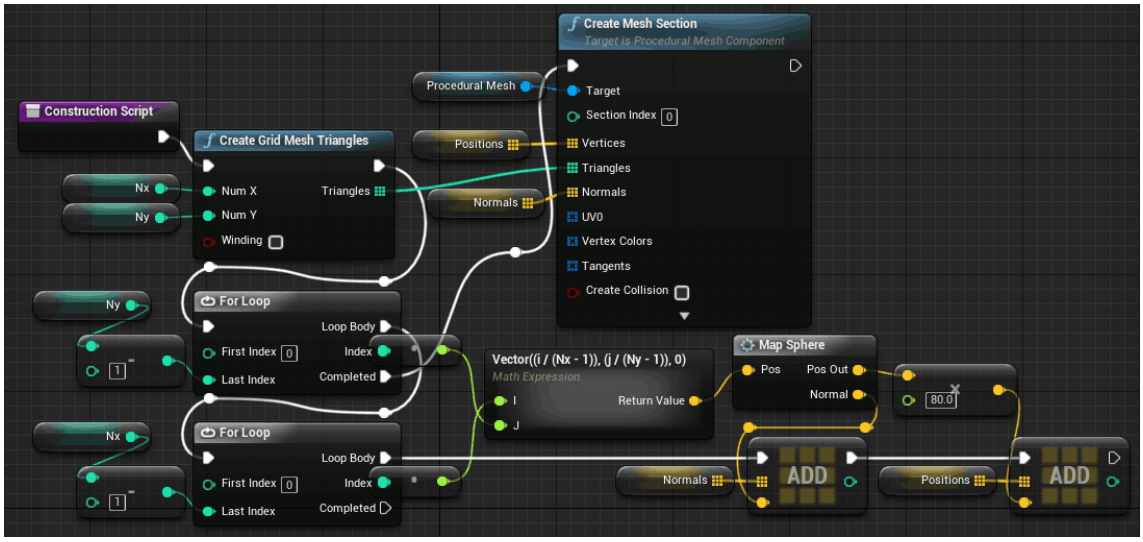
```
Vector(cos(x)*sin(y), sin(x)*sin(y), cos(y))
```

When done, save the macro and go back to the Construction Script. Insert the mapSphere macro just before the multiplication by 80.



Play the level and now instead of a plane you should see a sphere (Figure 32b and 32d).

Our mesh does not have normals, so we will fix that next. Create another variable called "normals" and make it an array of Vectors (just like the positions variable). Connect the normals variable to the Normals input of `Create Mesh Section`. We need to populate the normals array. Create a getter for the normals variable and connect it to a new `Add` node. Connect the Normal output of the mapSphere macro to the `Add` node. Altogether, the final graph should look something like the following



Our last task will be to make a torus shape. A torus is created using the mapping

$$\begin{aligned} x &= (t\sin(\phi) + 1 - t)\cos(\theta) \\ y &= (t\sin(\phi) + 1 - t)\sin(\theta) \\ z &= \cos(\phi) \end{aligned}$$

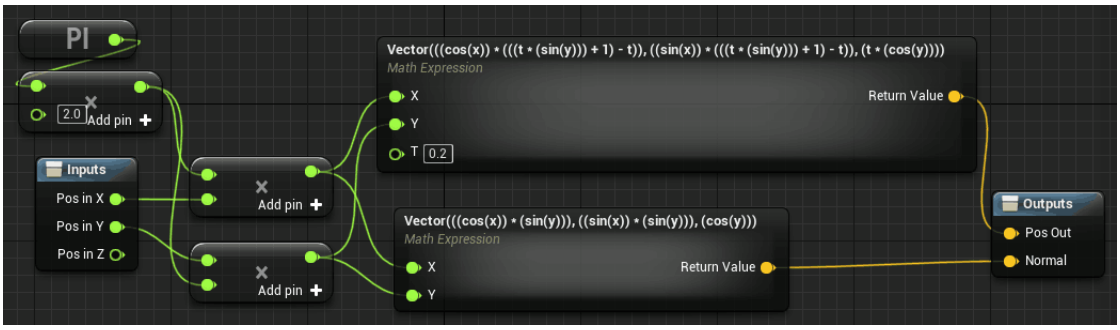
where θ is in $[0, 2\pi]$ and ϕ is in $[0, 2\pi]$ and t is the relative thickness in $[0, 1]$. Just like we did for the sphere, we will create a macro to generate the torus shape. Create another macro called "mapTorus". Set it up in the same way as mapSphere, however, with the math expression for the position set to

```
Vector(cos(x)*(t*sin(y) + 1-t), sin(x)*(t*sin(y) + 1-t), t*cos(y))
```

and the normal set to

```
Vector(cos(x)*sin(y), sin(x)*sin(y), cos(y))
```

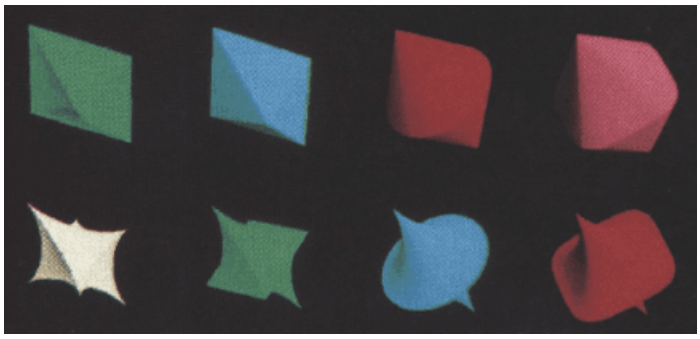
The mapTorus graph should look like the following



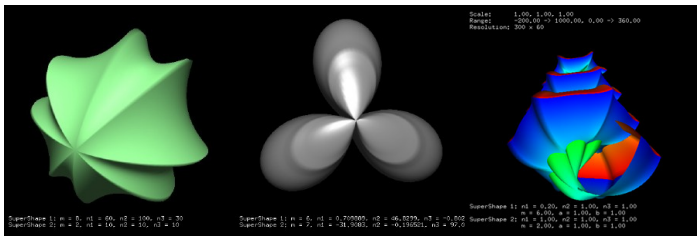
After you are done, save the macro and replace the mapSphere macro with the mapTorus macro. Play the level and you should see a torus instead of a sphere (Figures 32c and 32d).

Going Further

As a next step, you can try creating other shapes out of the tessellated plane. A good place to start are the family of spherical product shapes which include *superquadrics* [2] (Figure 33a) and *3D supershapes* [5] (Figure 33b).



(a)



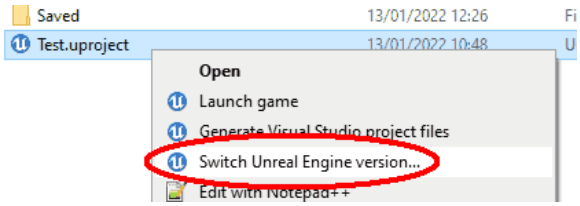
(b)

Figure 33: Spherical product shapes which include (a) superquadrics and (b) 3D supershapes.

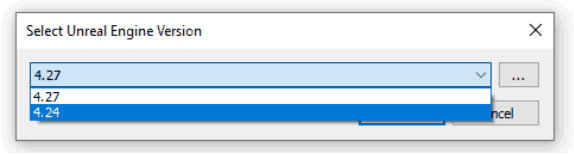
10 Troubleshooting

10.1 Project Will Not Open in Old Version of Editor

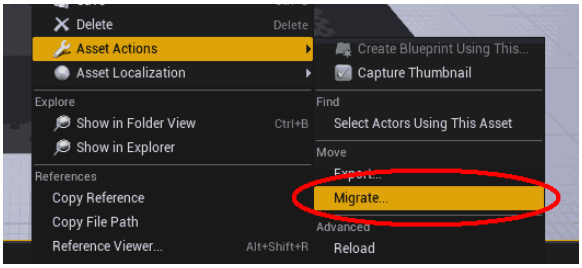
Technically speaking, Unreal Editor is not backwards compatible. A project created in a newer version of the editor will not open in an older version of the editor. Still, you may be able to open some parts of the project by changing its version. Before attempting anything, make sure to backup your project in case anything goes wrong. Right-click the `.uproject` project file in your file explorer and select `Switch Unreal Engine version...`



and then choose the version you want to switch the project to



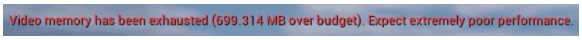
Be warned that assets created in a newer version will likely not show up. In order to get the assets to show up, they must be migrated to the old version *from* the new version. Open the original project (in the newer editor version) and then for each asset, right-click and select `Asset Actions > Migrate...`



When prompted select the Content directory of the *old* project you want to migrate the asset to.

10.2 Sluggish Response in Editor

In the editor, you experience a very sluggish response. You may also see a warning message of this kind



This is a known problem with version 5 of the editor and effects Windows machines. There appears to be a bug or major inefficiency with the editor's use of DX12[2]. To fix, go to Edit > Project Settings > Platforms > Windows. Find Targeted RHIs and change Default RHI to DirectX 11.

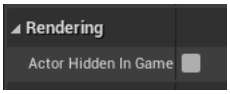
[2] https://forums.unrealengine.com/t/video-memory-has-been-exhausted/247164/23

10.3 Player Passes Through Mesh

This is typically caused by there being no collider defined for the mesh. Most of the StaticMeshComponents provided in the editor (cube, sphere, etc.) already have colliders, however, if you import your own mesh from an external object file, then it is possible no collider is defined. To make a collider, double-click the mesh asset to open the Mesh Editor. Then create a collider from the Collision menu item. For more detailed instructions, see https://docs.unrealengine.com/4.27/en-US/WorkingWithContent/Types/StaticMeshes/HowTo/SettingCollision/.

10.4 Actor Shown In Editor, But Not In Game

All Actors have an "Actor Hidden In Game" attribute that may be set to true by default. This can be the case with a lot of user-interface-related components, like billboards. In the Actor's Details under Rendering, check the following is not checked



In C++, SceneComponents have a "Hidden In Game" attribute that may be set to true by default. Fix via

```
MyComponent.bHiddenInGame = false;
```

10.5 Project Size Too Big or Just Keeps Growing

You have been working for some time on a project, but its size on disk is quite large or just keeps growing. This may be caused by the editor not actually removing assets from disk when you delete them in the Content Browser. It may also be caused by simply having large-sized assets. If you want to reduce your project's footprint on disk, try the following:

1. Right-click on one of your assets in the Content Browser and select Show in Explorer. It should take you to the Content directory for the project which holds all your assets.
2. Close the Unreal Editor.
3. Backup your project.
4. See if there are any assets in the project Content directory there that are no longer used and delete them. Also check for any large-sized assets and see if you can reduce their size. For example, convert 3D object files from ASCII format to binary, convert PNG to JPG, use mono instead of stereo sound files, reduce resolutions, etc. If you can reduce the size of an asset, then do not overwrite what is in the Content directory, but rather re-import it through the editor.
5. Open your project and make sure everything still works okay.

10.6 Blueprints: Nothing Works!

You have connected a bunch of nodes together and then press play and nothing happens.

A very common cause of this is forgetting to connect the `Exec` pins. Whenever you add a new node to your graph, always check if it requires an `Exec` to do its task.

Another issue that can arise, especially with `Math Expression` node, is that you enter something invalid into a field and then the editor disconnects that node from the nodes it was previously connected to.

If you are working on a Blueprint Actor, make sure there is an instance in your scene.

If you are expecting some geometry to be rendered (e.g. from a `Procedural Mesh`), ensure that your actor is not hidden inside another piece of geometry such as a wall or floor. Additionally, ensure that the winding order of triangles is correct so that front faces do not get culled.

10.7 Blueprints: `Print String` Prints Nothing to the Screen

You connect a variable in a `Print String`, but see nothing on the screen when you press play. Possible causes are:

- you forgot to connect to the `Exec` inlet of `Print String`
- you are trying to use `Print String` in a Construction Script (these will only show up in the Output Log)

10.8 Blueprints: Keyboard Events Not Working

You have wired up a keyboard key event to activate something, but nothing happens when playing the level. There can be several causes of this:

- you forget to enable input on the player (connect a `Get Player Controller` to an `Enable Input` node)
- you have not clicked on the window

10.9 Blueprints: Cannot Find Key Event

You are searching for a keyboard key event by typing something like "k" or "key k", but it does not jump to the correct result. This is because you are shown only the first match, not necessarily the best match. For common letters, it is typical to not get the best match. Try instead "keyb ev k" to search only within the Keyboard Event category.

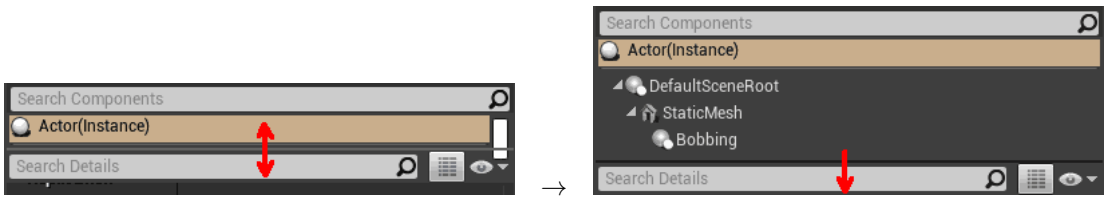
10.10 Blueprints: My Struct/Macro/etc. Does Not Show Up In Another Blueprint

You created a custom structure, macro, function, etc. but it does not appear in the search from within another Blueprint. Potential causes of this include

- you did not save the asset (its icon in the Content Browser has a star on the bottom-left)
- your code is not visible from within the Blueprint

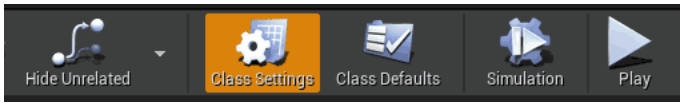
10.11 Blueprints: Item Lists Not Showing Up In Editor

You created a variable, component, etc. in the editor, but you cannot see it where it is supposed to be. A common cause of this is that the panel you are looking in is sized too small to see all items. If you hover the mouse cursor near the border of panels, the cursor will change to a double arrow and the border will become highlighted. You can then drag and expand the border to see all the items in the panel.

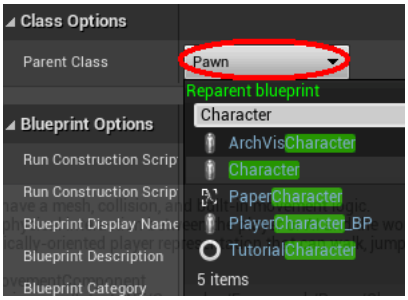


10.12 Blueprints: Wrong Parent Class Chosen For Blueprint

You created a new Blueprint class and spent several minutes programming it only to realize you chose the wrong parent class since some functionality is not available. A common scenario is creating an Actor Component when you meant Scene Component. What do you do? Panic? Start over? No! Fortunately, you can "reparent" any Blueprint class very easily. Open the Blueprint and then press the Class Settings button on the top bar.



On the right, you should see a Parent Class drop-down. Choose the new parent from here and your class will get updated.



10.13 C++: Editor Crashes After Compilation

Even the best programmers will write code that crashes the editor. Unreal Editor is courteous in that it presents a window when a crash occurs and provides some information that can be used for troubleshooting(Figure 34). The most useful information is the crash report in the middle of the window. At the top of the crash report is the type of exception that occurred which can give you a good hint as to why your code is crashing. Underneath the exception is a call stack trace with the top item being the last entry point before the crash. At the very top of the crash report there is a faint link you can click to navigate to the generated reports on your disk.

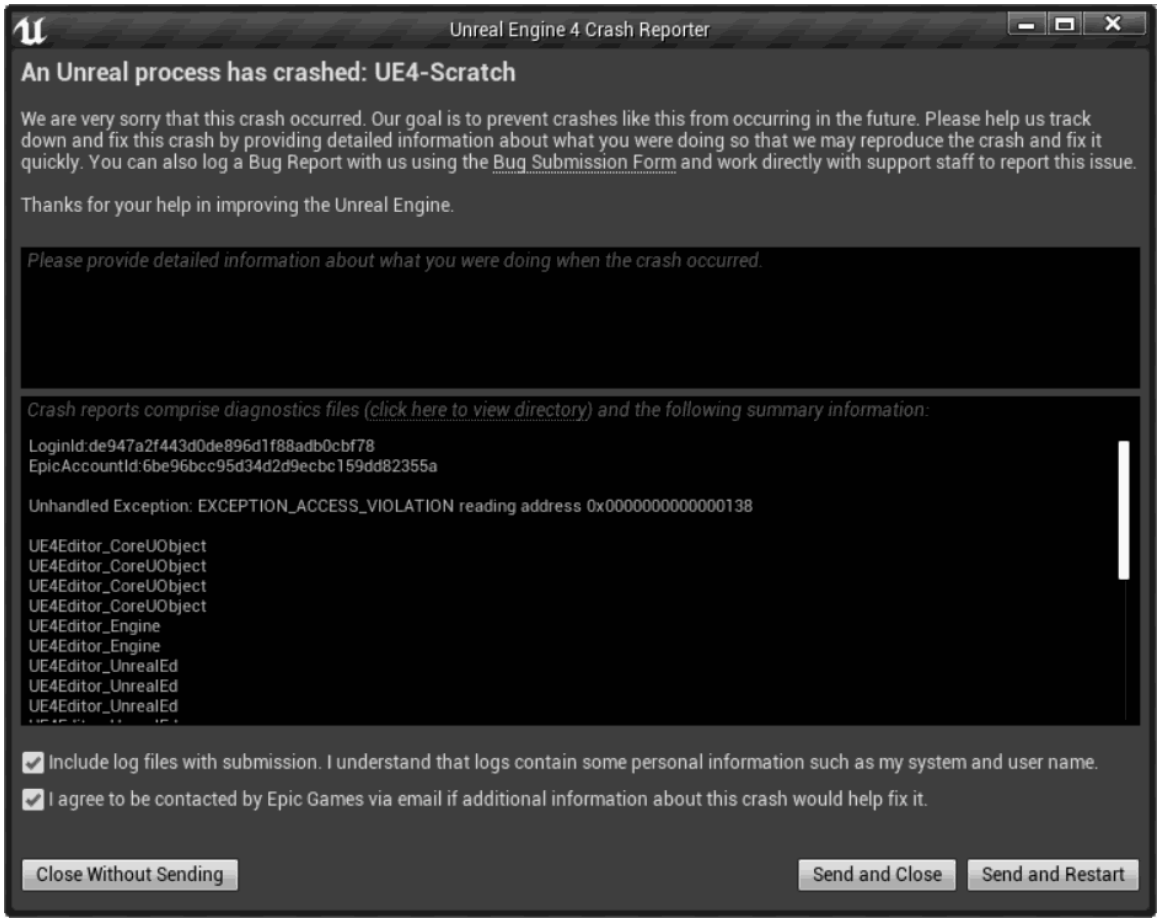


Figure 34: The Unreal Editor "gray screen of death." This dialog typically appears when your C++ code causes a run-time error.

Crashes typically occur because your code has produced a run-time error (e.g., array access out of bounds, dereferencing an invalid/null pointer, etc.). To prevent further crashes, you must remove the error in your code. Try to remember the last thing you did before the crash occurred, comment it out and recompile. If the editor crashes every time you open your project, then delete all files or directories (e.g. Win64) in your project's Binaries directory and then re-open the project. A dialog will appear mentioning missing modules and will ask if you want to rebuild the modules. Click yes and wait for the build to finish. Once you have fixed the run-time error in your code, then the project will open without crashing the editor. If the editor still crashes, it means you have not found the problematic code and may need to iterate this process multiple times.

References

[1] Scratchapixel 2.0. Value noise and procedural patterns: Creating a simple 1d noise. https://www.scratchapixel.com/lessons/procedural-generation-virtual-worlds/procedural-patterns-noise-part-1/creating-simple-1D-noise, 2016. Last accessed March 2022.

[2] Alan H. Barr. Superquadrics and angle-preserving transformations. *IEEE Computer Graphics and Applications*, 1(1):11–23, 1981.

[3] Jonathan Beard. A short intro to stream processing. http://www.jonathanbeard.io/blog/2015/09/19/streaming-and-dataflow.html, 2021. Last accessed December 2021.

[4] James F. Blinn. Simulation of wrinkled surfaces. In *Proceedings of the 5th Annual Conference on Computer Graphics and Interactive Techniques*, pages 286–292, New York, 1978. Association for Computing Machinery.

[5] Paul Bourke. Supershape in 3d. http://paulbourke.net/geometry/supershape/, 2003. Last accessed March 2022.

[6] Nathaniel Bowditch. On the motion of a pendulum suspended from two points. *Memoirs of the American Academy of Arts and Sciences*, 3(2):413–436, 1815.

[7] Char Davies and John Harrison. *Osmose*: Towards broadening the aesthetics of virtual reality. *Computer Graphics (ACM SIGGRAPH)*, 30(4):25–28, 1996.

[8] Microsoft Technical Documentation. High-level shader language (HLSL). https://docs.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl, 2021. Last accessed February 2022.

[9] Unreal Engine 4.27 Documentation. Actor mobility. https://docs.unrealengine.com/4.27/en-US/Basics/Actors/Mobility/, 2021. Last accessed December 2021.

[10] Unreal Engine 4.27 Documentation. Actors and geometry. https://docs.unrealengine.com/4.27/en-US/Basics/Actors/, 2021. Last accessed December 2021.

[11] Unreal Engine 4.27 Documentation. Audio files. https://docs.unrealengine.com/4.27/en-US/WorkingWithAudio/WAV/, 2021. Last accessed January 2022.

[12] Unreal Engine 4.27 Documentation. Blueprint macro library. https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/Blueprints/UserGuide/Types/MacroLibrary/, 2021. Last accessed January 2022.

[13] Unreal Engine 4.27 Documentation. Blueprint visual scripting. https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/Blueprints/, 2021. Last accessed December 2021.

[14] Unreal Engine 4.27 Documentation. Connecting nodes. https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/Blueprints/BP_HowTo/ConnectingNodes/, 2021. Last accessed January 2022.

[15] Unreal Engine 4.27 Documentation. Creating blueprint classes. https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/Blueprints/UserGuide/Types/ClassBlueprint/Creation/, 2021. Last accessed January 2022.

[16] Unreal Engine 4.27 Documentation. Macros. https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/Blueprints/UserGuide/Macros/, 2021. Last accessed January 2022.

[17] Unreal Engine 4.27 Documentation. Material editor. https://docs.unrealengine.com/4.27/en-US/RenderingAndGraphics/Materials/Editor/, 2021. Last accessed February 2022.

[18] Unreal Engine 4.27 Documentation. Material editor ui. https://docs.unrealengine.com/4.27/en-US/RenderingAndGraphics/Materials/Editor/Interface/, 2021. Last accessed February 2022.

[19] Unreal Engine 4.27 Documentation. Material functions overview. https://docs.unrealengine.com/4.27/en-US/RenderingAndGraphics/Materials/Functions/Overview/, 2021. Last accessed March 2022.

[20] Unreal Engine 4.27 Documentation. Materials. https://docs.unrealengine.com/4.27/en-US/RenderingAndGraphics/Materials/, 2021. Last accessed February 2022.

[21] Unreal Engine 4.27 Documentation. Math expression node. https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/Blueprints/UserGuide/MathNode/, 2021. Last accessed January 2022.

[22] Unreal Engine 4.27 Documentation. Nodes. https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/Blueprints/UserGuide/Nodes/, 2021. Last accessed January 2022.

[23] Unreal Engine 4.27 Documentation. Pawn. https://docs.unrealengine.com/4.27/en-US/InteractiveExperiences/Framework/Pawn/, 2021. Last accessed February 2022.

[24] Unreal Engine 4.27 Documentation. Physically based materials. https://docs.unrealengine.com/4.26/en-US/RenderingAndGraphics/Materials/PhysicallyBased/, 2021. Last accessed March 2022.

[25] Unreal Engine 4.27 Documentation. Struct variables in blueprints. https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/Blueprints/UserGuide/Variables/Structs/, 2021. Last accessed January 2022.

[26] Unreal Engine 4.27 Documentation. Unreal editor interface. https://docs.unrealengine.com/4.27/en-US/Basics/UI/, 2021. Last accessed December 2021.

[27] George Eckel, Ken Jones, and Tammy Domeier. *OpenGL Performer Getting Started Guide*, chapter 6. Creating Scene Graphs. Silicon Graphics, Inc., 1997. https://web.archive.org/web/20120830204551/http://techpubs.sgi.com/library/tpl/cgi-bin/getdoc.cgi/0650/bks/SGI_Developer/books/Perf_GetStarted/sgi_html/ch06.html.

[28] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991.

[29] Jr. John L. Kelly, Carol Lochbaum, and V. A. Vyssotsky. A block diagram compiler. *Bell System Technical Journal*, 40(3):669–678, 1961.

[30] Jules Antoine Lissajous. Mémoire sur l'étude optique des mouvements vibratoires. *Annales de Chimie et de Physique*, 51:147–231, 1857.

[31] David R. Myers. Interactive computer graphics and PHIGS. In *Proceedings of the 7th Summer School on Computing Techniques in Physics*, Bechyne, Czechoslovakia, 1987.

[32] Ken Perlin. An image synthesizer. In *Proceedings of Siggraph '85*, pages 287–296, San Francisco, 1985.

[33] Craig W. Reynolds. Flocks, herds, and schools: A distributed behavioral model. *Computer Graphics*, 21(4):25–34, 1987.

[34] Craig W. Reynolds. Steering behaviors for autonomous characters. In *Proceedings of the Game Developers Conference 1999*, pages 763–782, San Jose, CA, 1999.

[35] Curtis Roads. *The Computer Music Tutorial*. MIT Press, Massachusetts, 1996.

[36] Karl Sims. *Particle Dreams*. [Digital Film], 1988.

[37] Karl Sims. Particle animation and rendering using data parallel computation. *Computer Graphics*, 24(4):405–413, 1990.

[38] Sweetwater. Varispeed. https://www.sweetwater.com/insync/varispeed/, 2000. Last accessed January 2022.

[39] Christopher West. White boxing? gray boxing? what? https://gamedevchris.medium.com/white-boxing-grey-boxing-what-6aa9cfa3b0e4, 2021. Last accessed January 2023.

[40] what-when how. Scene graphs (advanced methods in computer graphics) part 1. http://what-when-how.com/advanced-methods-in-computer-graphics/scene-graphs-advanced-methods-in-computer-graphics-part-1/, n.d. Last accessed January 2022.

[41] John Whitney. *Digital Harmony: On the Complementarity of Music and Visual Art*. Byte Books, Peterborough, NH, 1980.

[42] John Whitney and Larry Cuba. *Arabesque*. [16mm film], 1975. USA.

[43] Wikipedia. Dataflow programming. https://en.wikipedia.org/wiki/Dataflow_programming, 2021. Last accessed December 2021.

[44] Wikipedia. Object-oriented programming. https://en.wikipedia.org/wiki/Object-oriented_programming, 2021. Last accessed January 2022.

[45] Wikipedia. Scene graph. https://en.wikipedia.org/wiki/Scene_graph, 2021. Last accessed December 2021.

[46] Wikipedia. Ieee 754. https://en.wikipedia.org/wiki/IEEE_754, 2022. Last accessed January 2022.

[47] Wikipedia. Lissajous curve. https://en.wikipedia.org/wiki/Lissajous_curve, 2022. Last accessed February 2022.

[48] Wikipedia. Unreal engine. https://en.wikipedia.org/wiki/Unreal_Engine, 2022. Last accessed January 2022.

[49] Simon Zolin. Audio formats comparison. https://stsaz.github.io/fmedia/audio-formats/, 2016. Last accessed January 2022.