Week 2 *Recap*$_1$

Taraash

Contents

Introduction

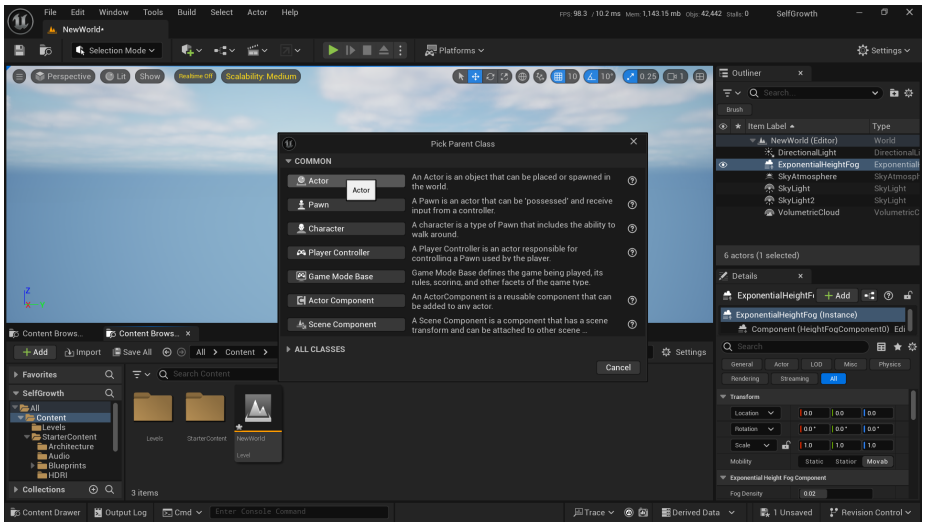
The class was taught by Utkarsh, but I'll be writing the recap cause his phone was stolen, so he's dealing with that. But at least you get LaTeX'd notes! We went over Unreal's Blueprint system, and I'll be explaining the basics of it here. (Find the week 1 notes here). We really did cover a lot, these will be part 1 of the recap, I'll upload part 2 later with the rest of the stuff we covered.

What are Blueprints?

From Epic's documentation (which you can find here), Blueprints are a node-based programming system that allows you to add functionality or interactivity to your game, without having to directly write code (which will be C++ in Unreal's case). They are a visual scripting language that is built on top of Unreal Engine's C++ codebase.

Getting started

To create one, right-click in the content browser and select *Blueprint Class*. Doing so will open a new window with a bunch of options that look like this:



I'll briefly go over some of the most common types:

Actor Everything in Unreal is referred to as an Actor, hence it is the most basic class you could create. This is something you could just drag and drop into the world, without much interaction to it.

Pawn This is a type of Actor that has more functionality. In this case, a Pawn can receive input and can be interacted with.

Character This is a type of Pawn which is bi-pedal (two arms, two legs) and has a built-in movement system (walk, run, jump, etc.). This is what you would use if you've got a character in your game.

There are more (many more, expand the *All classes* section). Many have a documentation page which you could easily look up.

Lights in Blueprints

Remember the environment light mixer window? Say we customized it heavily for a game and would like to save it, to be easily reused in a different level or project. We can create a Blueprint that contains all these settings, and then drag and drop it anywhere. Let's first customize our lights.

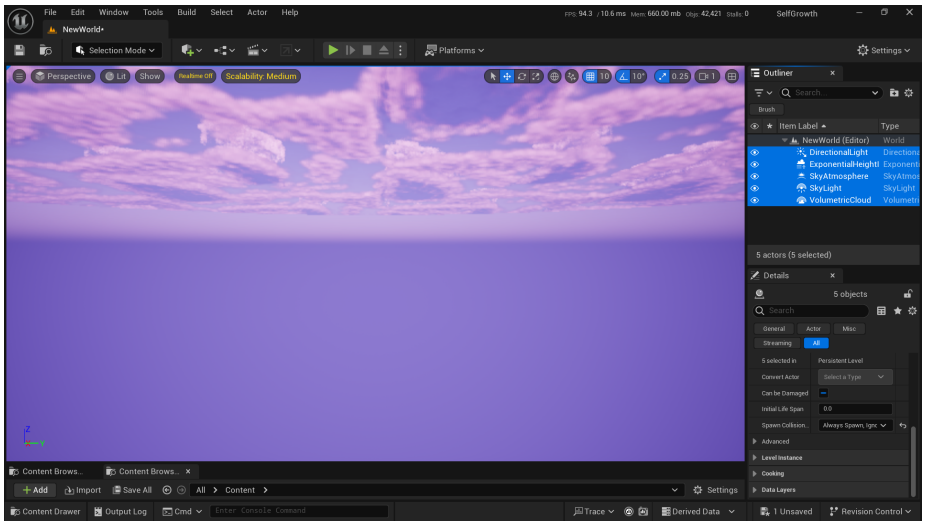
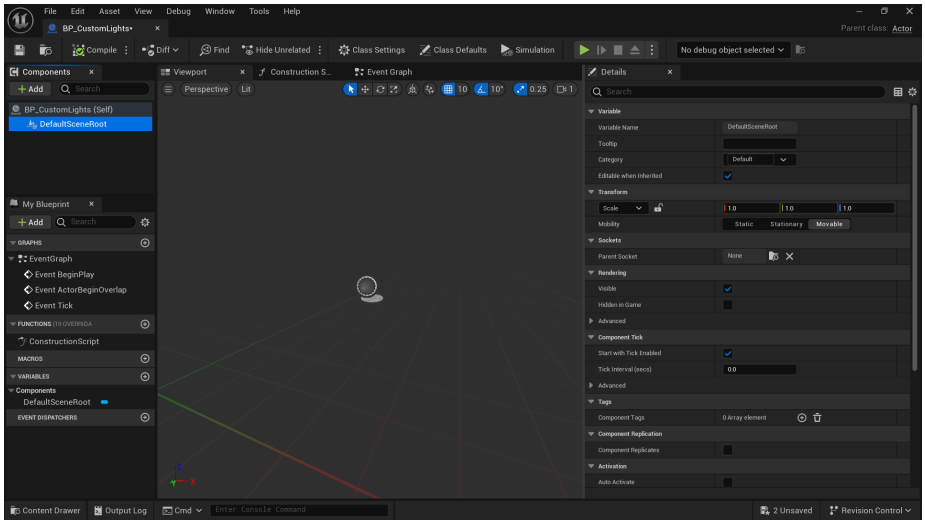


Figure 1: Tweaked lights

I'll now select all the lights from the outliner and press `Control+C` to copy them.

In the content browser, I'll create a new Blueprint class and select *Actor*. Name it `BP_CustomLights` (the `BP_` is a common naming convention, you should follow it). Double click the new blueprint to open it up.

We will be greeted with something that looks like



1. On the top left, we have the components panel. Very similar to the outliner, but here we can add components to our Blueprint (via the Add button).

2. On the bottom left we have the My Blueprint panel. This is where we can see all the variables, functions, and events that we have in our Blueprint. (more on this later).

3. On the right, we have the details panel, which, as the name suggests, shows details of the currently selected component.

4. The window in the middle is the familiar viewport, this is what we will see when we drag and drop our Blueprint into the world.

5. Take a look at the settings icon next to most of the panels, it's got a bunch of useful options.

6. The `Class Defaults` section is where you can set parameters related to multiplayer, replication and other defaults.

7. The `Class settings` section is mainly only used to set up `Interfaces`, we'll talk more about these later.

Now we paste all the lights we copied earlier. Select the **Default Scene Root** component in the components panel, and press `Control+V` to paste the lights. (The usual hierarchy for "parents and children" is followed, if something is below something, and indented, the top actor is the parent, and all it's transformations will also be applied to the children.) You should see them appear in the

viewport, and in the components panel as well. (You could manually add them in using the Add button as well, but this is easier.)

You'll notice the components panel has a little more stuff in it. Particularly, the Billboards. These are just visual indicators that help you see where the lights are in the viewport to help you grab/rotate them. (These are 2D planes with a texture on them that always face the camera.)

Select any of the light and the details panel will show the various properties you can change. The real fun part is you can change these values dynamically, outside this details panel, maybe when the player is doing something, or when a certain event happens.

Construction Script

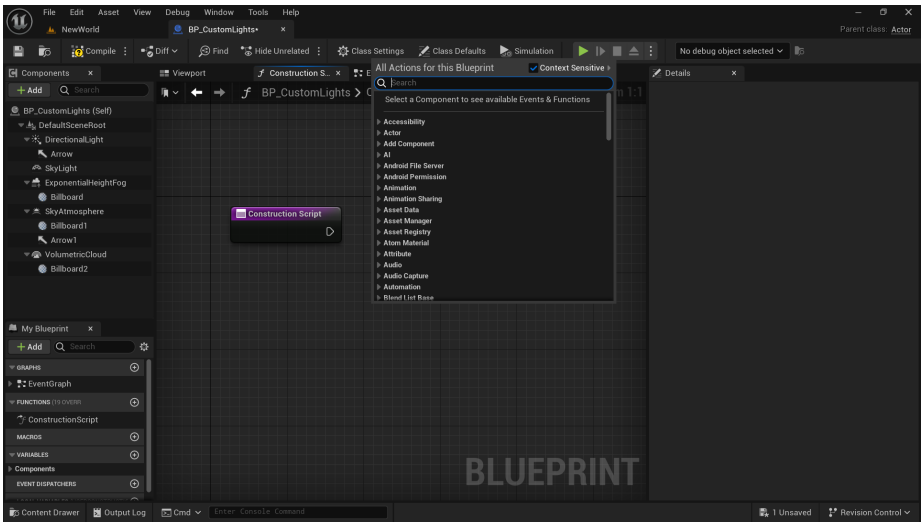
Right next to where is says *viewport*, there is a tab that says *Construction Script* (If you don't find it, you bring it back by clicking on it in the `My Blueprints` tab, it will **not** be in the Window panel).

The construction script is run when the Blueprint is placed in the world, or when any of its properties are changed. It's useful for setting up the Blueprint, tweaking the variables and positions of the components.

The event graph is run every frame, and is used for more dynamic things, like responding to player input, or changing the state of the game. We'll talk about this later.

Basic Blueprint stuff

1. Right-click anywhere on the event graph to open the add menu.
2. Alternatively, you can drag and drop a component from the components panel into the event graph, this will create a node for that component.
3. Even more alternatively, you can drag a pin from any existing node to bring up the same menu. This is useful for performing context-sensitive searches (see the `Context sensitive` switch on the top left).



Manipulating lights

Remember the lights we pasted in the components panel? And how clicking on them brings up it's details (the parameters we can set) in the details panel? We can access these parameters in the Blueprint as well. Here, I'll just drag a few and hook them up, you should experiment with various different nodes here.

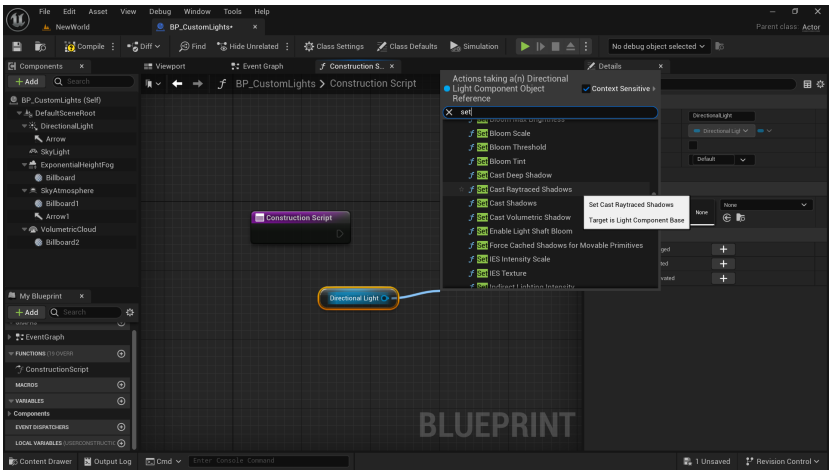


Figure 2: A list of pretty much all you could change about this specific type of light.

Variables and Navigation

A few navigation tips:

- Scrolling the mouse wheel will zoom in and out.
- Left-click + drag will pan.
- Right-click + drag will select.
- Pressing `C` over a select will create a comment box, useful for organization.
- `Alt+click` on a connection will break it.
- `Ctrl+drag` will allow you to move a connection.
- Double-clicking on a connection will bring up a *reroute* or *dot* node, which is useful for organizing your graph.
- Dragging a variable directly on top of a pin will create a connection (if the types are compatible).

A little talk about variables:

- The simplest way to create a variable is right-clicking on a parameter and selecting `Promote to variable`.
- Alternatively, in the my blueprints panel, you can click the + button next to the `Variables` section, and create a new variable. Rename it and change its type in the details panel or right next to the name. Take a look at Epic's documentation on variables for more information.
- You'll be asked to compile the Blueprint after creating a variable, this checks for any errors in the Blueprint and makes sure everything is up to date. You do it by clicking the big `compile` button the top left.
- It's also recommended you save the Blueprint after any change, press the disk icon next to the compile button to do so.

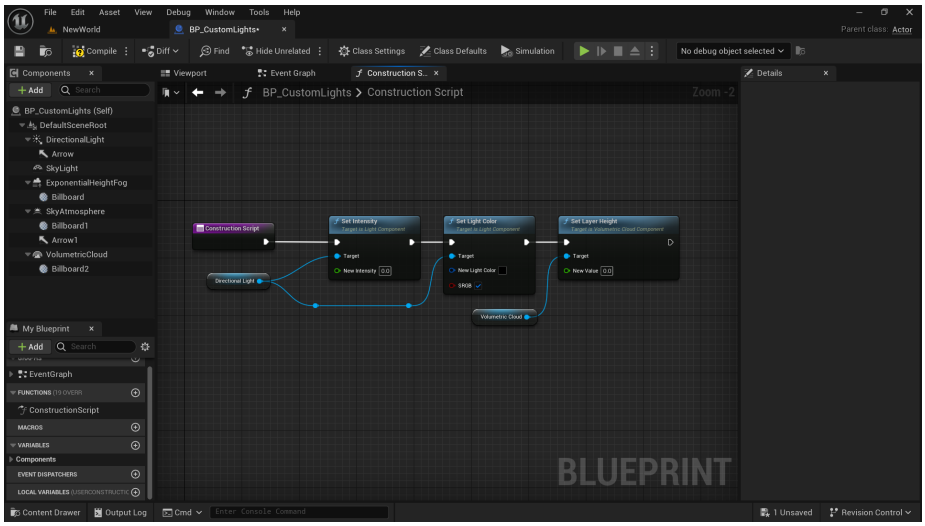
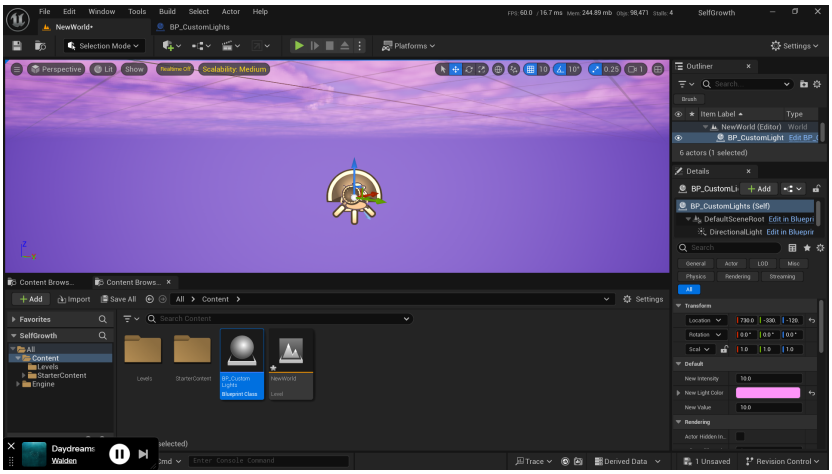


Figure 3: A simple construction script.

Let's break the above screenshot down a bit:

- I've added variables by right-clicking on the parameters and selecting `Promote to variable`. This automatically creates a variable with the same name and type as the parameter and connects it.
- I then compiled the blueprint, this allowed me to set the defaults.
- After selecting the variable, I can see its details in the details panel. I can add a description, make it editable in the 3d viewport, set a default value and more.
- If I check `Instance editable` (same as clicking on the eye icon next to the variable name), I can change this value without opening the Blueprint, directly in the 3d viewport.

We now have a Blueprint that we can drag and drop into the world, and it will have all the lights we set up in it. We can also change the values of the lights directly in the 3d viewport, without having to open the Blueprint.

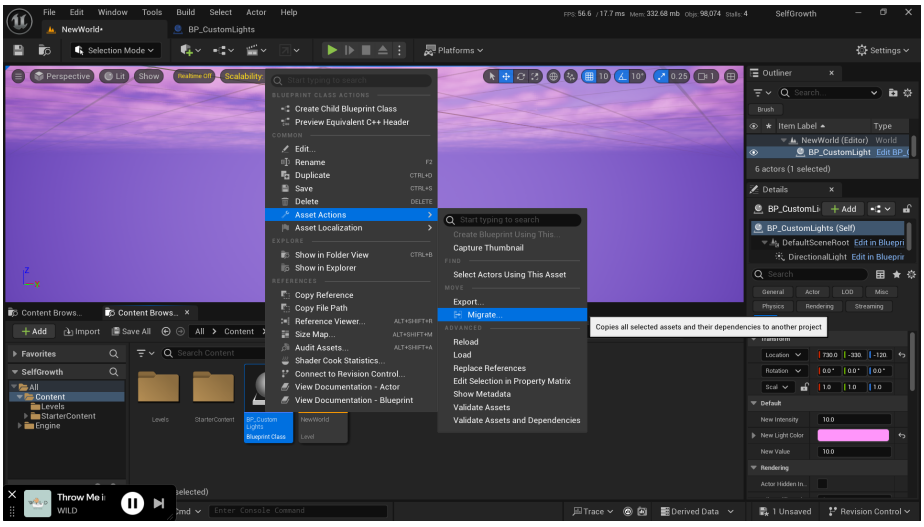


(I've got some music going on with Spotify's miniplayer on, you'll find some good songs too!)

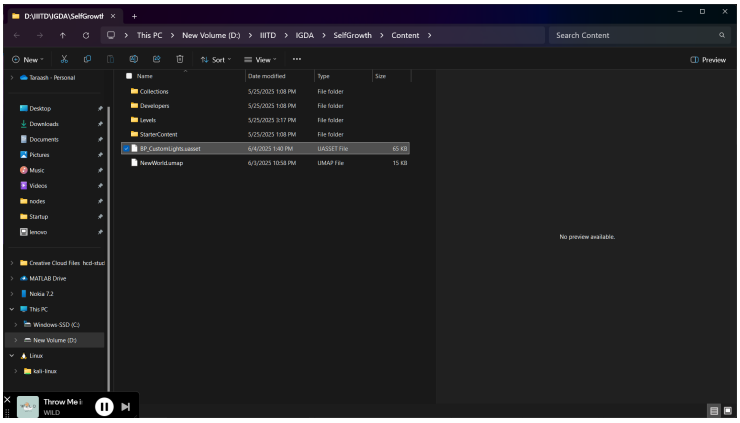
Take a look at the details panel with our blueprint selected, we can change the values here!

You can use this blueprint in any other project as well. There are two ways to do so:

Migration Right-click on the Blueprint in the content browser and select `Asset Actions > Migrate`. This will copy the Blueprint and all its dependencies to a new project.



Via File Explorer If you'd like only this asset somewhere, you can find the actual asset on your computer (look at the `Show in Explorer` button), copy this asset and paste it anywhere inside your second project.



Unreal will ask for confirmation before importing this asset in your new project.

Making a wall

We didn't cover this in the class, but I thought it would be a good idea if I covered this, it'll introduce you to some basic nodes and get you comfortable with the Blueprint system.

We will make a simple blueprint that has a `Sections` variable that defines the length of our wall, and a `Material` parameter, which specifies the material.

We will be doing this in the construction script as we don't want anything to change while playing (this is what the Event Graph is for).

1. Add an InstanceStaticMesh component to the Blueprint and set it to the Wall_400x400 mesh (you can find it in the Engine Content, or search for it in the content browser, if you don't find it, press the little settings icon next to the search bar in the drop down and check `Show Engine Content`).

2. I created a variable called `Sections`, changed its type to `Integer`, and made it instance editable. I defaulted it to 1.

3. The for-loop should be familiar. It starts from 0 and runs till $sections - 1$ (inclusive). The `Loop Body` is executed for each iteration of the loop. To be more explicit, when the for-loop node is executed (by the execution signal given off by the `Construction Script` node), it will run the loop from 0 to $sections - 1$.

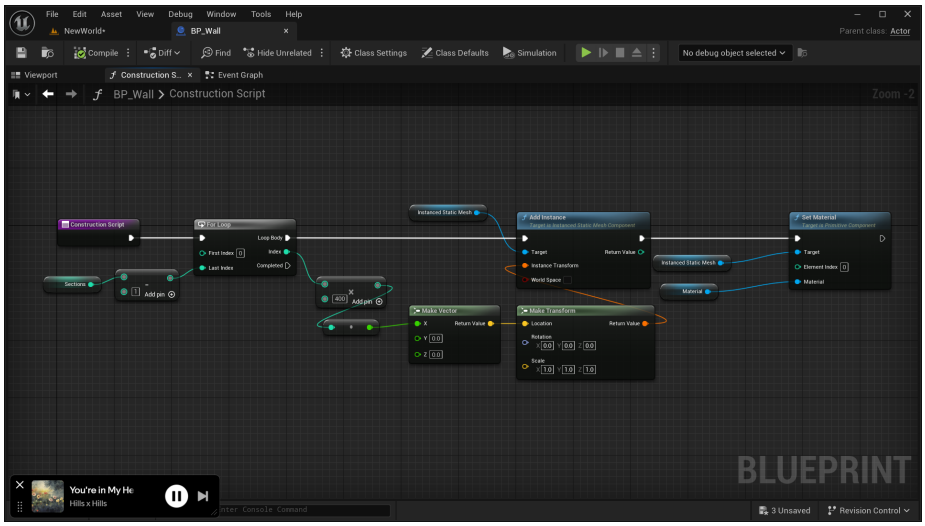


Figure 4: The final construction script, I'll explain it below.

4. I'm then adding a new instance of our Wall_400x400 mesh, and transforming it to where I need it.

5. A `Transform` is a combination of a location, rotation, and scale. The `Make Transform` node allows us to create a transform from these three values.

6. We only need to modify the location, which is a vector, the `Make Vector` allows us to create this vector.

7. I only need to move it in the x direction, so I'm doing that.

8. We then set a material. Simply drag the `InstanceStaticMesh` node and search for material, you'll find the node.

9. The tiny cyan/aqua (int) color to the green (float) color is a conversion node, which is automatically created when you try to plug *compatible* (but not the same) types together. In this case, the `make vector` takes floats and we were giving it an integer. Hence, the conversion node.

Instances

A tiny detour to talk about Instances, these are copies of a single object, which share the same data. This allows them to render much faster, modify in bulk and save memory. If you've got the same mesh being duplicated multiple times (without changing the geometry), you should use instances.

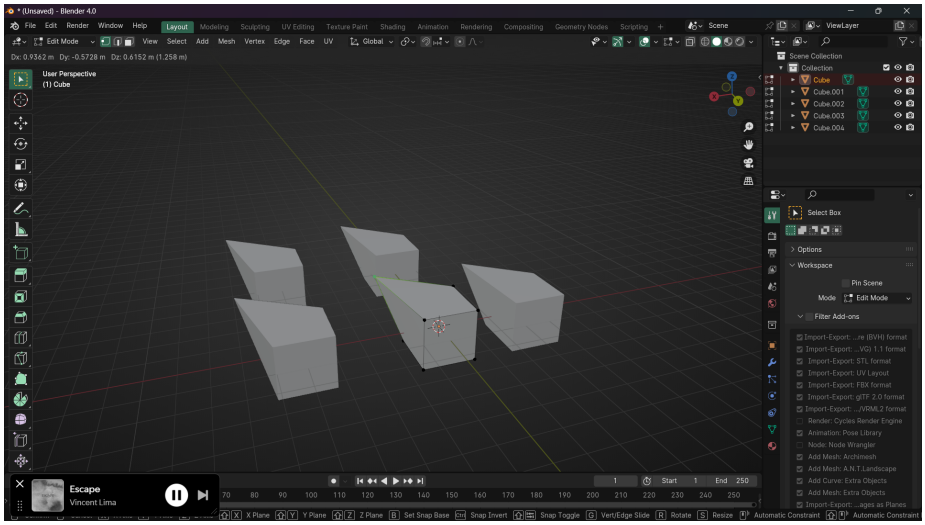


Figure 5: Instances in Blender.

Here, I'm modifying the geometry of a single cube, and since the rest are instances, they are modified as well.

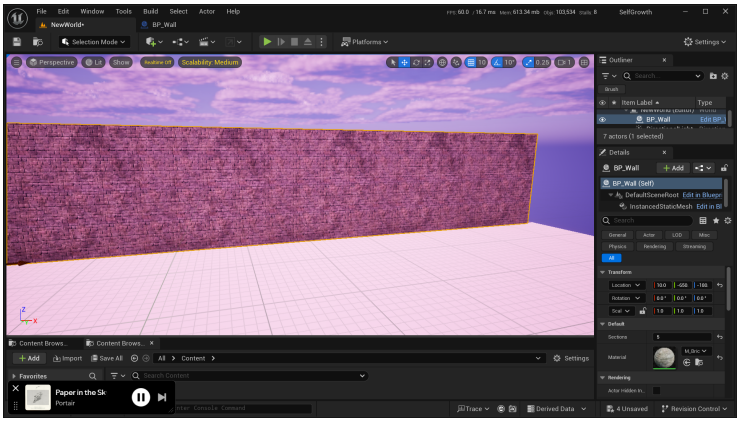
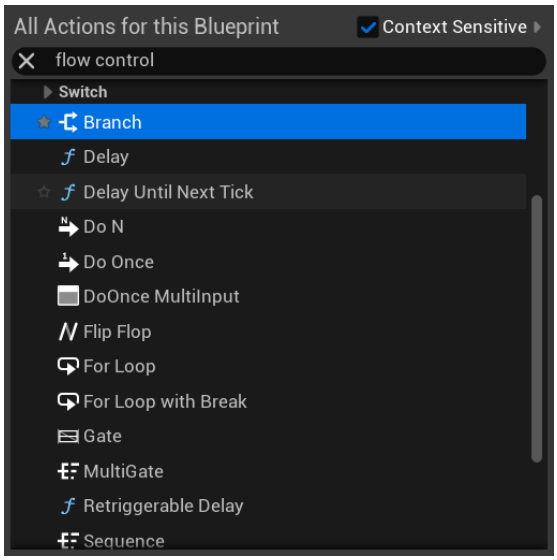


Figure 6: The final result.

Flow Control nodes

These are a collection of nodes that allow you to control the flow of execution in your graph.



I'll briefly describe a few:

Branch This is an if-else statement. It takes a boolean input and executes one of the two outputs based on the value of the boolean.

Sequence This is a node that allows you to execute multiple nodes in sequence. It has multiple outputs, and each output will be executed in order. useful for organization and readability.

Do once As the name suggests, say you've got a node that you only want to execute once, this is the node for you. It will execute the first time it is called, and then ignore all subsequent calls until it is reset (which is done by another execution pin, once this reset signal is received, it will allow for one more execution).

For loop We took a look at this above.

Timeline Extremely similar to a for-loop, but this is used for animations, it updates a value every event tick (when a frame changes). You can change interpolation (that is how the graph is drawn between the keyframes). More about this later.

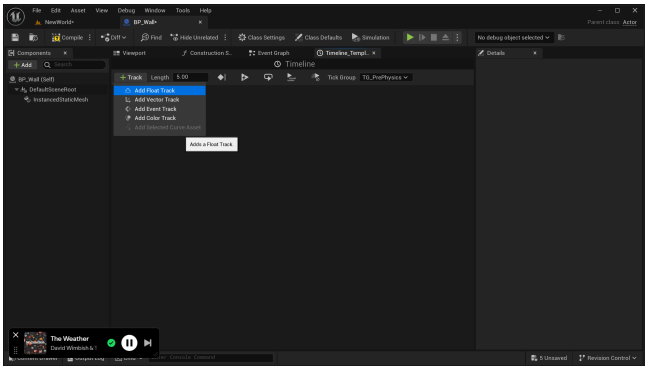
The Timeline node

The timeline node is a very powerful node that lets you update something over time and needs to be only executed once, as I mentioned, very similar to a for-loop. I'll list a few differences:

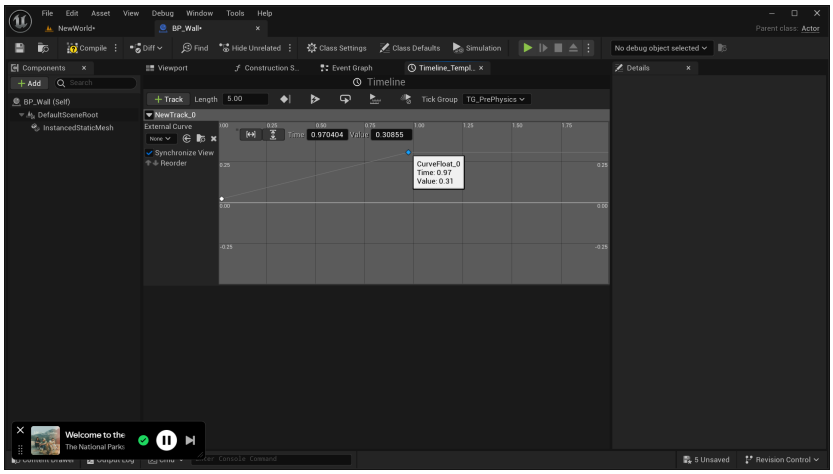
1. For-loops are executed per iteration, no matter how much time it takes, while the timeline node is executed once per frame, and it will update the value every frame.

2. For-loops are linear (or proportional) in nature, while the timeline node can be set to any interpolation type (linear, cubic, etc.) and has keyframes which is incredibly powerful.

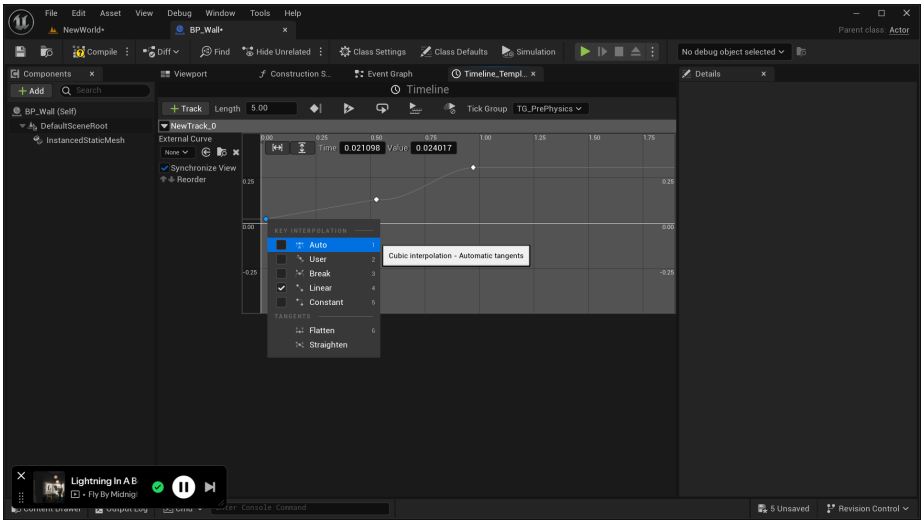
3. Double click on the timeline node, which opens up a separate window. We'll talk about this now.



I'll select a float track, since that's the simplest one, with one keyframe being described by two items — the `value` (the value of the keyframe) and the `time` (the time at which the keyframe is set). You can add more keyframes by right-clicking on the timeline and selecting `Add Key` or `Shift+left click` on the graph. You can also change the interpolation type by right-clicking on a keyframe and choosing an interpolation mode.



A selected keyframe turns blue. Take a look at the top to find the Time and Value pairs.



Right-clicking a keyframe will let you change the interpolation type, setting Auto is recommended, this gives you a smooth curve with two additional points to change the curve. I highly recommend you play around with these settings.

Also take a look at the `length` setting at the top, this is the duration of the timeline, which is the time it takes to go from the first keyframe to the last keyframe.

Conclusion

I'll end now, very tired I am. The second part should follow soon. I haven't talked about the Event nodes, which are fairly important. Search for event in the add-menu and see if you use the timeline node to create a simple animation.

A task: Create a door that opens two seconds after the game is started. (You can do it many ways, a do-once next to the event tick, or a delay node after the event begin play.)

I'll cover the following topics in the next part:

- Event nodes
- Collision and overlap events
- Custom events
- Casting