CS354p Lab 2: Blueprint and Git

This lab is to help familiarize you and your group members with using Blueprint event graphs and integrating UE5 and Git via Gitlab. If you have previous experience with Git, a lot of this should be familiar, but what we are setting up here will be foundational for all your remaining assignments and labs, so please make sure you fully understand all the steps of this lab. Also, even though you will ultimately share one repository for group assignments, please make sure every teammate is fully completing all of these steps and individually submitting the lab.

Getting Started

We are going to build upon Lab 1 for this assignment, so to get started, please copy and paste the entire Lab 1 directory. We will not do a full rename, since that is fairly involved. Instead, we will rename the folder containing all of Lab 2 from "Lab1" to "Lab2." Delete all the intermediate folders (Binaries, DerivedDataCache, Intermediate, and Saved), and rename the .uproject to Lab2.uproject. You may notice the module name inside Lab2.uproject is still Lab1, but we're going to leave that for now.

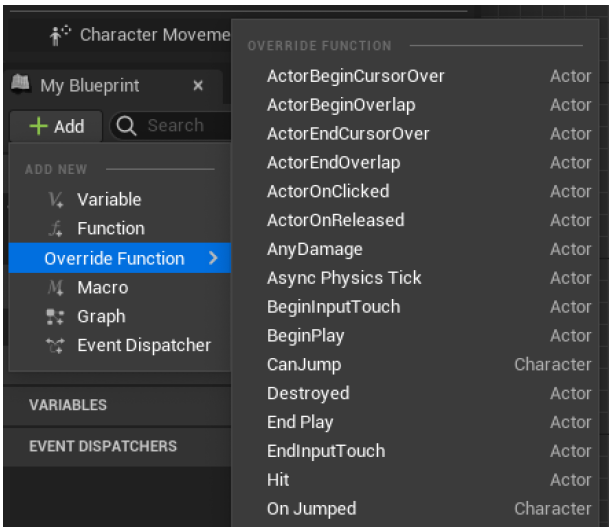
Double click on .uproject. It will ask you if you want to rebuild the project. Once you say yes, the deleted files should be regenerated and the project should open in UE5. You may want to click Tools -> Refresh Project to regenerate/update the .sln/.xcworkspace etc.

Using Blueprints and Event Graphs

We are now going to move functionality out of the Controller and to the Character. This may be helpful depending on how many characters you have, how many character-specific controls you have, but in this case, we're doing this to see how to pull functionality out of the C++ and into the BP. First, we are going to create Blueprint Implementable Events, which allow us to call events in the BP Event Graph from our C++. To get started, create three functions in Lab1Character.h (notice I haven't bothered to rename the Character class to Lab2Character): DashActivatedEvent, TouchHeldEvent, and TouchTappedEvent. These events will map to the events callbacks in Lab1PlayerController. They should look something like this:

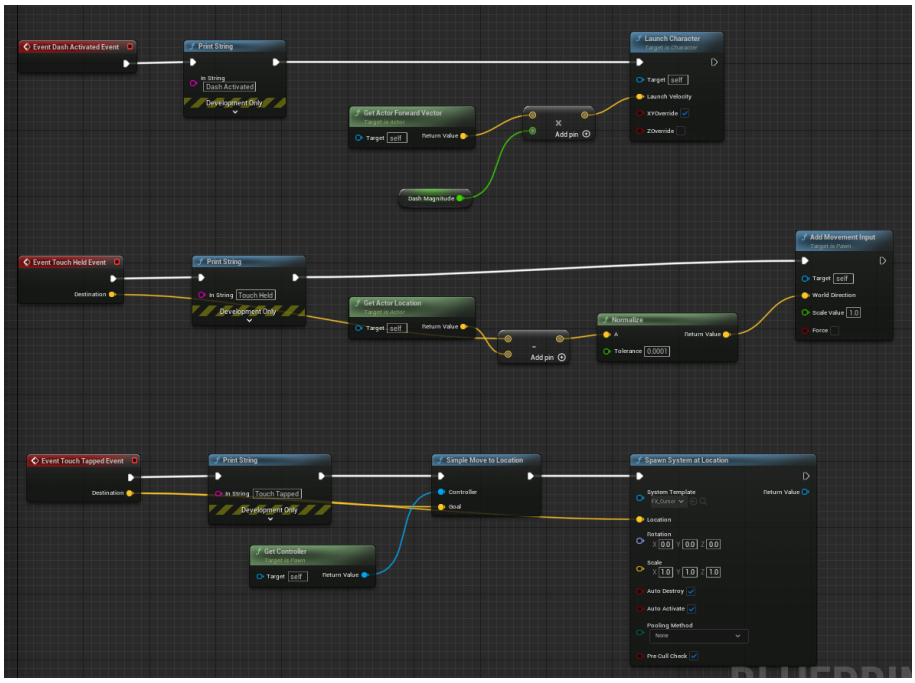
```
UFUNCTION(BlueprintImplementableEvent)
void DashActivatedEvent();
```

(But as we will see, we may want to include arguments on some of them later). Compile your C++ changes via the UE editor. When you open your BP_TopDownCharacter file, you should now be able to add these three events to your Event Graph by clicking Add -> Override Function as shown below:



Once they are added, to the event graph, add a Print node to their exec pin so we can confirm they connected successfully. Before we can test this is working, though, we need to add the calls to our new functions in the C++. To do this, we want to connect them to the Lab1PlayerController so we can handle the Character movements via Blueprint rather than in Lab1PlayerController. We will need to drop in our functions in `OnDashStarted`, `OnSetDestinationTriggered`, and `OnSetDestinationReleased`. Once these functions are connected, recompile and confirm the print statements are appearing as expected.

Finally we need to move the movement functionality over from the existing C++ to our Blueprint. There is some amount of system and math knowledge we need to understand to make this work, but the final output will look like this:



Make sure you comment out the equivalent code in the C++ and confirm this code runs as expected. If it does, you're now ready to get this code uploaded to a git repository!

Using Git

If you do not have a GitLab, you should go ahead and create an account. You will need to add an SSH key to your account, so depending on if you've never done this before, you may first need to generate an ssh key. The process for this depends on your setup, and Gitlab has all the details you need here: `https://gitlab.com/help/ssh/README`. If you want to use Access Tokens and HTTP access, you can do so, but you'll need to go through those steps on your own.

Once you have the necessary keys uploaded, create a new, empty repository called CS354p Lab 2. You will now need to initialize git for your local project then connect it to the remote project. Gitlab will have instructions for this, but it is generally done as follows:

```
cd CS354p_Lab2
git init --initial-branch=main
git remote add origin git@gitlab.com:username/cs354p-lab-2.git
```

Before you do the initial add/commit/push, though, we'll create a `.gitignore` file, and set up git LFS for the project.

Git Ignore and Git LFS

A `.gitignore` includes a list of all files you do not want to push to the remote repository. You can find numerous examples of how to create your `.gitignore` file to work with UE5, but the basic idea is to ignore build-related directories, and files that can be generated by UE5. For example, you do not need to include the `CS354p_Lab2.sln` because you can regenerate it by right-clicking the `CS354p_Lab2.uproject` and selecting "Generate Visual Studio Project Files" (similarly you do not need to include `CS354p_Lab2.xcodeproj` if you are working on OSX). Other generated files including .sdf, .opensdf, .suo, .ipch, .db, and .opendb (or their OSX/Linux equivalents) can be ignored as well. For an example of ignorable files, you can look at this: `https://github.com/samsheff/UE4-Gitignore/blob/master/UE4.gitignore`.

In terms of directories, you only need to include /Source, /Config, /Plugins (if you have any), /Content, and any directory you want to include "source" binaries such as .psd, .png, .fbx, etc. All other directories can safely be included in your `.gitignore` file.

For Git LFS, download the Git extension here: `https://git-lfs.github.com/`. Once this is installed, all you need to do is type:

```
git lfs install
```

And you're ready to go. You should not need to install git lfs on the same machine again, but remember that if you start working from a new machine, you'll want to run that command on that machine.

From the local git repository directory, you can now add binary files that you do not want to check into your repo to be tracked by lfs using `git lfs track`. The basic UE5 binaries you should track are .uasset and .umap. You can confirm they are being tracked correctly in the `.gitattributes` files, which should have been generated upon tracking a file type.

Connecting to Source Control

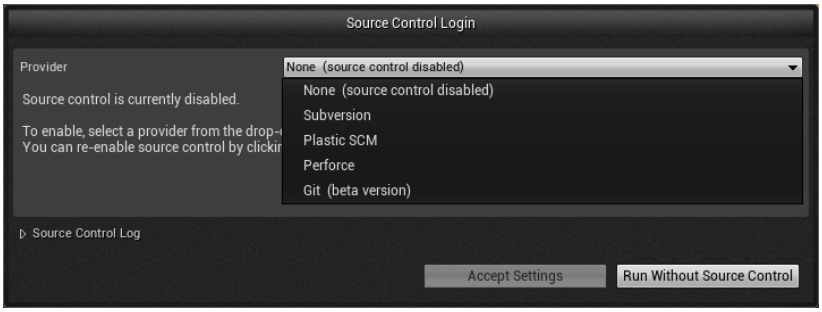
At this point `git add .` should successfully add all necessary files to staging, but I always check using `git status` just to confirm. You can now create your first commit message and push to the repository. If you never get around to changing your git editor settings to Emacs but also hate VIM (or just hate both VIM and Emacs), you can do what I do and add a `-m` to avoid the issue entirely:

```
git commit -m "Initial commit.  No VIM for me, thanks."
git push -u origin main
```

This initial push will take some time if you included the starter content, but it will be a good way to test that LFS is working!

Let's now make sure we can commit assets directly from UE5. Within the UE5 editor, go to Content-> TopDown-> Maps and add a Level file by right-clicking and selecting "Level." Give it a name or leave the default name — we're just making sure source control is working directly from the UE5 Editor.

Right-click this level file and select "Connect to Revision Control." You will see something like this pop up:



We will be using Git, so select Git then make sure the Git path is correct as well as your username and e-mail. Once you confirm, you should be able to right-click the level file again and select `Source Control -> Mark for Add`. The source control icon will change from a question mark to a plus sign, and if you run `git status` from the command line, you'll see the file has been added.

Once you've completed these steps, double check that others can successfully access your work. Add at least one team mate to each of your repositories. You will now clone at least one team mate's repository and open it in UE5 Editor to double check that everything is working. Remember that you will need to right-click the .uproject and build the Visual Studio files before running if the .gitignore is working correctly.

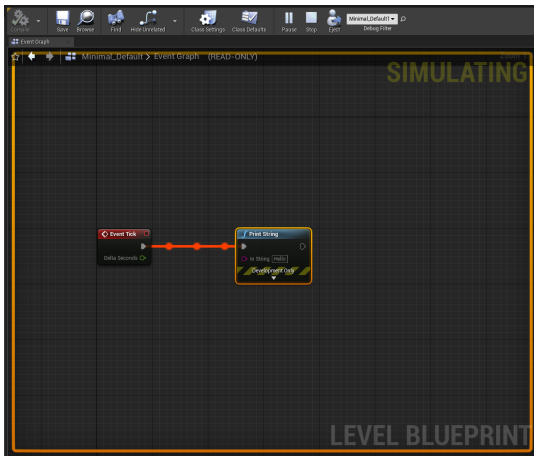
Finally, decide on a branching scheme that will work best for your team to coordinate. It can be as fine-grained as creating a new branch for every ticket, or as coarse-grained as each team member having a branch that is then merged into the main development branch. Have at least two of the team members make some small change using this branching schema to each of your repos and then push it to the repo. Once you have confirmed it is working, include documentation for how team members are expected to contribute in the README.

Debugging

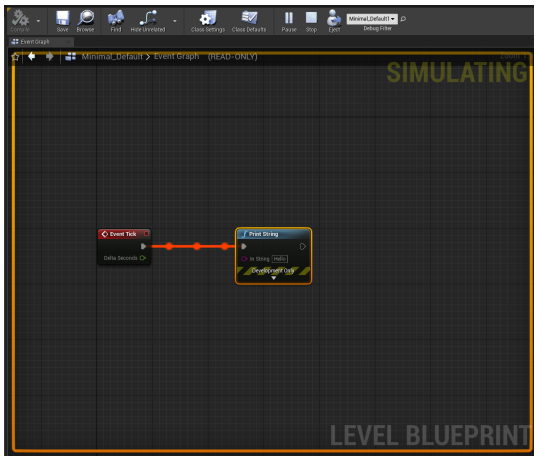
Debugging can be especially challenging when working in a large scale system like UE5. There is a lot of class complexity that can be confusing the work with, and beyond that many things can go wrong outside of the code, such as Blueprints needing variables reset or reattached, bad intermediate data in the build system, etc. We have many tools for investigating beyond printing debug messages to the screen. We'll now try out several ways listed below. There is no official turn-in for these, but please take some time to go through them, as they will be extremely helpful as you begin work on Assignment 1.

Blueprint Debugger

Blueprint offers high-level debugging tools to see the flow of execution from events and add some breakpoints during. In your main level, click on the Blueprints tab along the top. Select "Open Level Blueprint." Once you have opened the Level's event graph, click `Add New -> Override Function -> Tick`. This will create an event node that is called every frame. Attach a Print String node to the exec pin. When you compile and hit Play, you will notice a yellow box appears around the event graph view. If you select a value in the "Debug filter" drop down you will see an animated arrow flowing along the execution path. This is useful for figuring out if code in Blueprint is being executing and what the logical flow is.



Right-click the Print String node and add a Breakpoint. When you click compile and hit play, you will see a red arrow above this function indicating the execution has stopped here. You will also notice additional functionality along the top for debugging support. This is a subset of the tools you have access to using breakpoints in a traditional debugger.



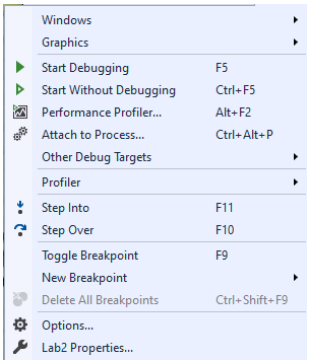
Console Debugger

To access the console, hit tlide (~) while in the Unreal Editor. This will open a console command, which you can use to issue commands during runtime. This is a very powerful tool that allows you to debug and test features while running the game, as well as switching between features and modes. To see a full list of commands type `help` into the console, which will open up a .html document with a nicely formatted list of commands and arguments.

Visual Studio Breakpoints

To use breakpoints in VS, you must first ensure you are in DeveloperMode running the 64-bit version. There should be similar settings for XCode, VSCode, etc, but you will need to familiarize yourself with your particular IDE to find out where those settings are if you are currently unsure.

If you click on the green arrow, it should launch a new instance of the UE5 Editor, but now if you press "Play" in the Editor, the executing code will hit any breakpoints you've set in VS. Put at least one breakpoint in to check that this is working (I'd recommend creating a constructor in `ALab2GameMode` similar to what we did in Lab 1. Put in some sort of code just to have something to look at). Try out the commands listed under "Debug" such as Step Into, Step Over, etc.



Notice you can also inspect variables by hovering over them when the breakpoint has stopped execution. You can also use the Autos and Locals windows to see variables and their current values. Autos shows variables used on the current and preceding lines, while Locals shows in-scope variables. You can also see the Call Stack.

There are many, many other things you can do with breakpoints and debugging in IDEs like Visual Studio, XCode, and VSCode, but these are the basics. Breakpoints are a great resource when you know roughly where something is going wrong, but you're unsure what or how.

Note: Visual Studio Building and Cleaning

When you are convinced it's Unreal's fault and not your own, it may be time to clean and rebuild the project. Cleaning removes intermediate files, which can be extremely helpful – especially if you're changing class headers a lot, fixing seg faults, bad memory accesses, etc, and find your project in a broken state where the code seems to be updated but you're not seeing the changes and/or it's still crashing repeatedly on trying to open the project for very vague reasons in the logs. Building in Visual Studio is the same as hitting Compile in UE5's Editor. It will not rebuild the lighting/geo/etc, but it may fix issues if you simply can't open UE5 Editor or the hot reloading is being janky.

Under the Build tab in your IDE, you will find options for building/cleaning the solution as well as building/cleaning the project. Building/cleaning the solution will rebuild everything including the UE5 engine project. Building your Lab2 project will only affect your code's object files. Try to use build and clean on your project before building and cleaning the solution which will take more time, but if you need to "nuke from orbit – it's the only way to be sure," clean and build the entire solution.

Submission

Please submit a link to your repository with your Lab2 project. Include a report that goes through what we did during the lab by including screenshots of the code and the Blueprints and explaining what this code does. If you were unable

to complete a feature, include an explanation of what you did and where you got stuck.