The Fundamentals of Git

A Manual by Seth King and Ben Odell

Welcome to the Git manual! The purpose of this document is to describe Git by addressing its purpose, exploring Git terminology, and demonstrating helpful bash commands. After you finish reading this manual, you should have a good idea of how Git will serve as a helpful tool to you and your CS383 project team throughout this semester.

**What is Git?**

Git is a version control system that allows its users to track and manage changes to shared code documents. It is the most commonly used version control, and takes shape through a wide variety of hosts, including Github, Atlassian Stash, and AWS Codecommit.

**Why is version control important?**

Version control is important because it allows teams to collaborate on a shared body of code. For example, in CS383, you can have each of your team members working on their own part of the project at any given time without interfering with each others’ coding process (though, of course, everyone will have to merge their code into a single body eventually).

**How will we use version control in our CS383 projects?**

As mentioned above, Git will likely be used as the primary version control system for each CS383 project team this semester. If you haven’t worked with Git before, it is a good idea to familiarize yourself with the system early on in order to ensure smooth project collaboration going forward.

**Terminology**

There are a wide variety of terms associated with Git. These can seem intimidating at first, but they begin to feel more intuitive as you work with the tool. Some important terms to know include:

* **Version control**: The tracking and management of changes to collections of documents, which typically contain code.
* **Version control host**: The platform that you use for version control. Some popular examples include Github, Stash, and CodeCommit.
* **Repository:** A specific collection of documents that you use version control to manage.
* **Local repository:** A repository stored locally on your machine.
* **Remote repository:** A repository hosted by a remote host (e.g. Github).
* **Clone:** Create a local copy of a repository. Changes to this local repository are tracked and can be merged back into the source repository.
* **Merge:** Integrate two branches into a single branch, including all changes from both branches.
* **Master:** The “main” or default branch for a repository. All branches are either eventually deleted or merged with the master branch.
* **Commit:** Save a “snapshot” of the repository after you have made changes to it.
* **Branch:** An independent line of development for a given repository. For example, if you want to add a feature to an application, you would create a branch of the repository containing the application code, edit the code to add the feature, and then merge the branch back into the master branch.
* **Check in:** Write changes from a local repository into a remote repository (but do not merge the changes).
* **Check out:** Get changes from a remote repository to a local repository (but do not merge the changes).
* **Push:** Write *and merge* changes from a local repository to a remote repository.
* **Pull:** Read *and merge* changes from a remote repository to a local repostory.
* **Pull request:** A forum where collaborators can view and discuss changes that have been made to a branch’s code. If the changes are approved by the collaborators, then they are typically merged with the master branch and become a permanent part of the repository.
* **Ignore (gitignore):** By including a gitignore file in your repository, you can specify files that you do not want to track with Git.

**Windows Subsystem for Linux**

Before we get into command line terminology, it may be helpful to discuss Windows Subsystem for Linux. If you are operating on Linux or MacOS, you can skip this section. WSL is a helpful tool for those of us running Windows (which, when I asked around in class, it seems like most of are) to have access to a Unix terminal without having to install a bulky VM or maintain a separate file management system. Thus, WSL is very helpful for using Git, because many Git tools are available and convenient to use via the Bash command line.

To install WSL, follow the instructions here:

<https://docs.microsoft.com/en-us/windows/wsl/install-win10>

After installing WSL, run the command **sudo apt install git-all** to install Git. Once the installation is complete, you will be able to make use of all the commands listed below.

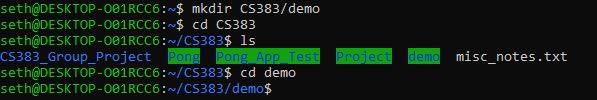
**Command Line Overview**

* git init
* git clone
* git checkout
* git pull
* git push
* git branch
* git commit
* git add
* git rm
* git remote

**Example: Cloning, editing, and pushing a repository with Git and WSL**

Suppose that there is a Github repository that contains an interesting Unity project. You want to clone the repository, edit the code, and create an executable. After this is done, you want to push the updated project to a new branch in the remote repository so that the manager of that repository can merge your code into their master branch if they want. The steps involved in this process would be as follows;

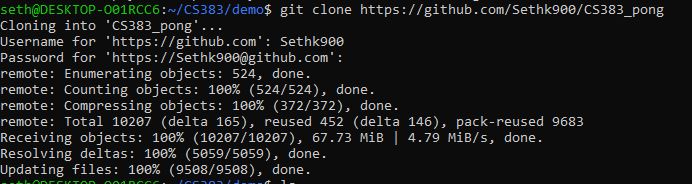
1. Open your Linux distribution, running on WSL (I use Ubuntu).
2. Create a directory that you want to clone the repository into and navigate to that repository.



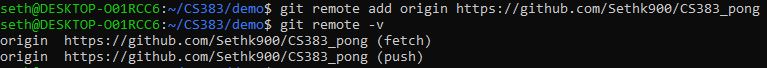
1. Initialize Git within that directory.



1. Clone the remote repository to your machine (for demonstration purposes, I’m using the repo that contains my Pong project). Note that the version control host (in this case, Github) will prompt you for credentials.



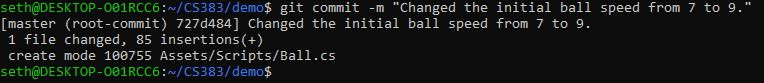
1. Next, we need to track the remote URL so that we can push/pull/checkin/checkout as needed between the remote and local repositories. You can call the remote address anything you want, but the convention is to call it “origin” or “upstream.”



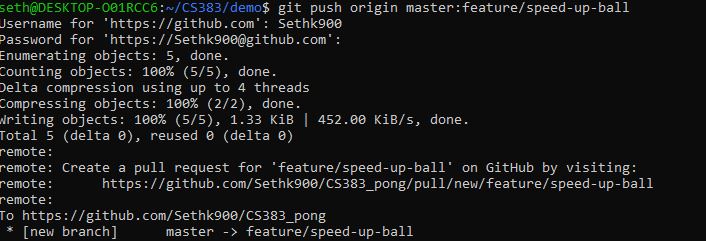
1. **Here’s the one downside of using WSL to manage Unity projects:** Unity doesn’t support case-sensitive file systems, which means that if you want to run the project, you will have to copy it to a place higher up in your directory and then copy any files that you change back into WSL. We won’t get into that too much because it isn’t related to Git.
2. For demo purposes, I just edited one file (Assets/Scripts/Ball.cs) to alter the ball speed. To track this change, we need to tell Git that we altered the file:



1. Next, we want to commit the changes that we made. You can think of this as saving the current version of the local repository.



1. Finally, we can push the updated repository to a new branch of the remote repository. I’m going to call the new branch “feature/speed-up-ball", so that the branch name reflects the change to the code.



1. In the Github console, you and your teammates can look at the code in the new branch. You can also create a pull request to have your teammates review the code and subsequently merge it with the rest of your project.



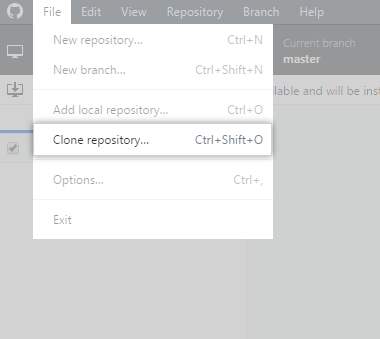
**Github Desktop (Windows and Mac)**

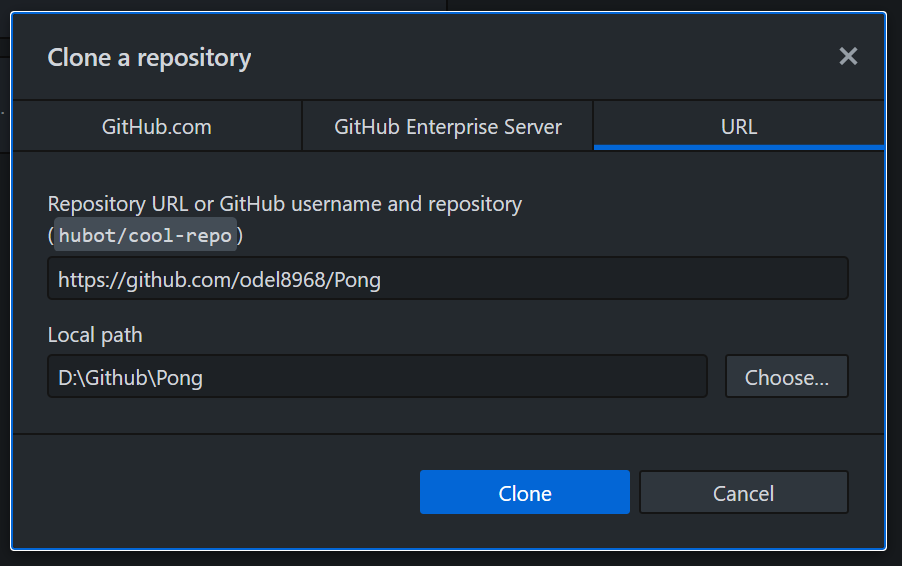
If you wish to avoid the command line interface, Github desktop provides a simple GUI for working with git repositories. The download and instructions can be found at:

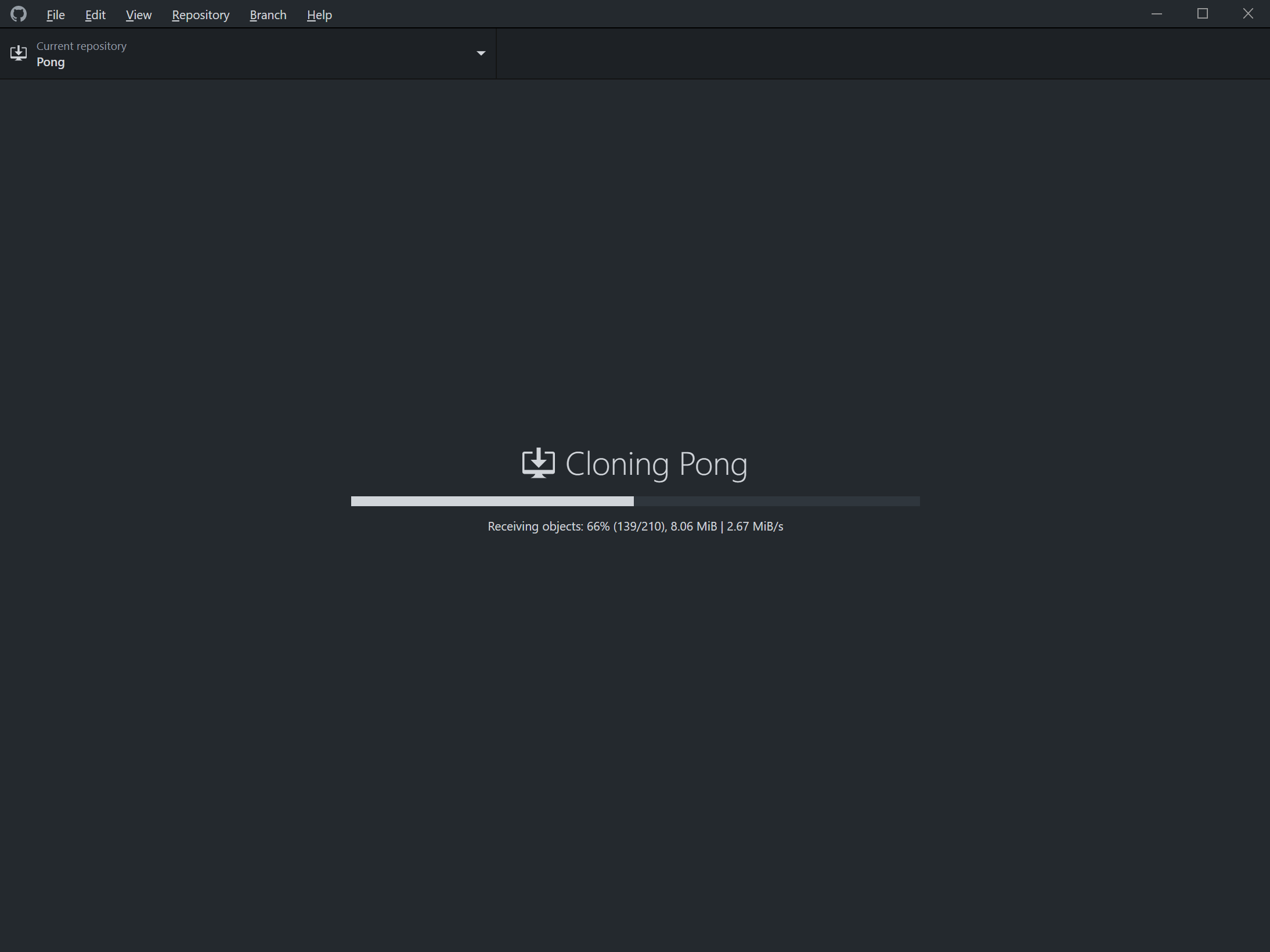
https://desktop.github.com/

**Example: Cloning, editing, and pushing a repository with Github Desktop**

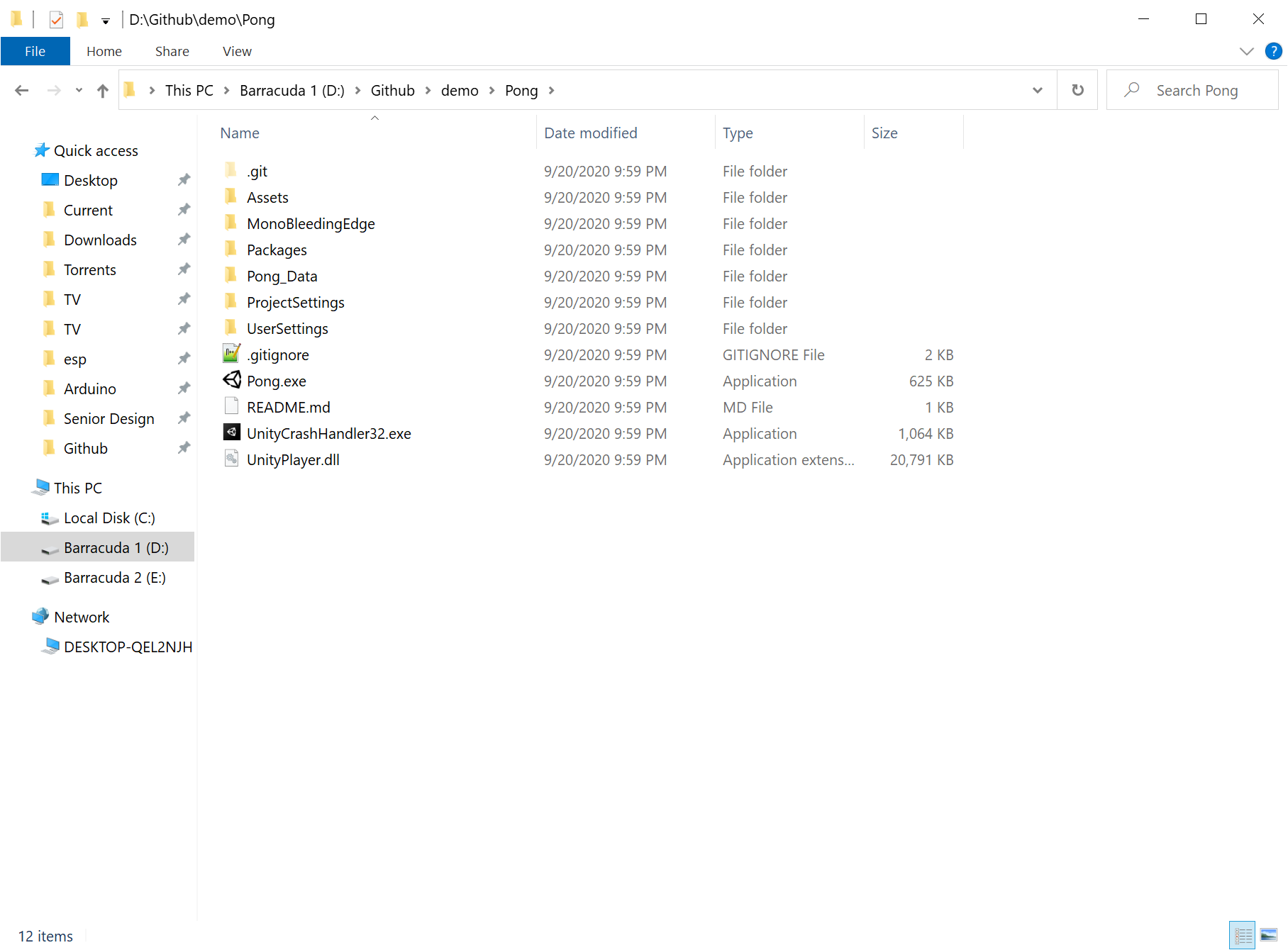
1. From the file menu, select clone repository. From here you will have to identify your repository’s online location as well as your local copy’s location. Once you’ve done both of these press Clone to begin building your local copy.

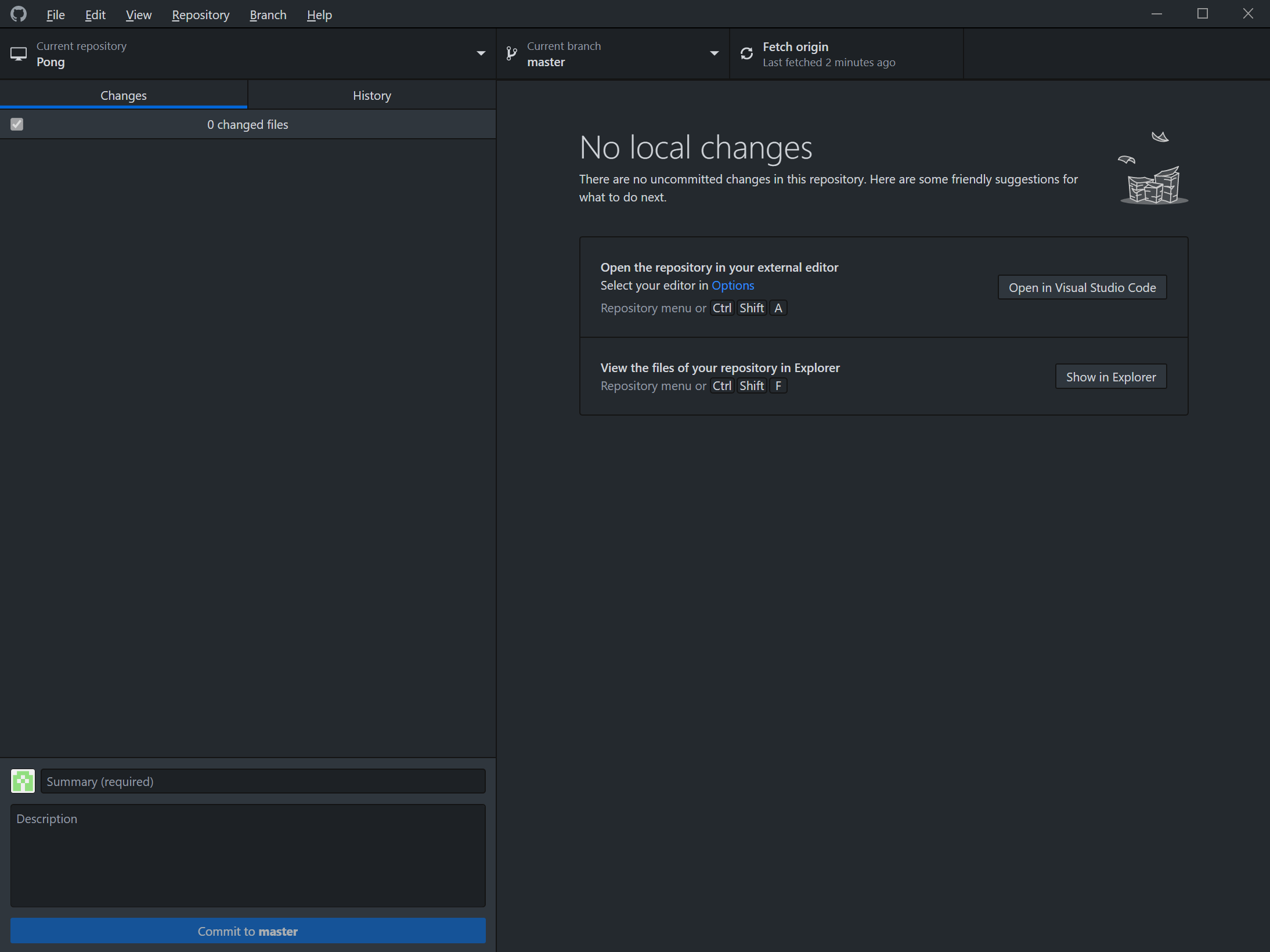




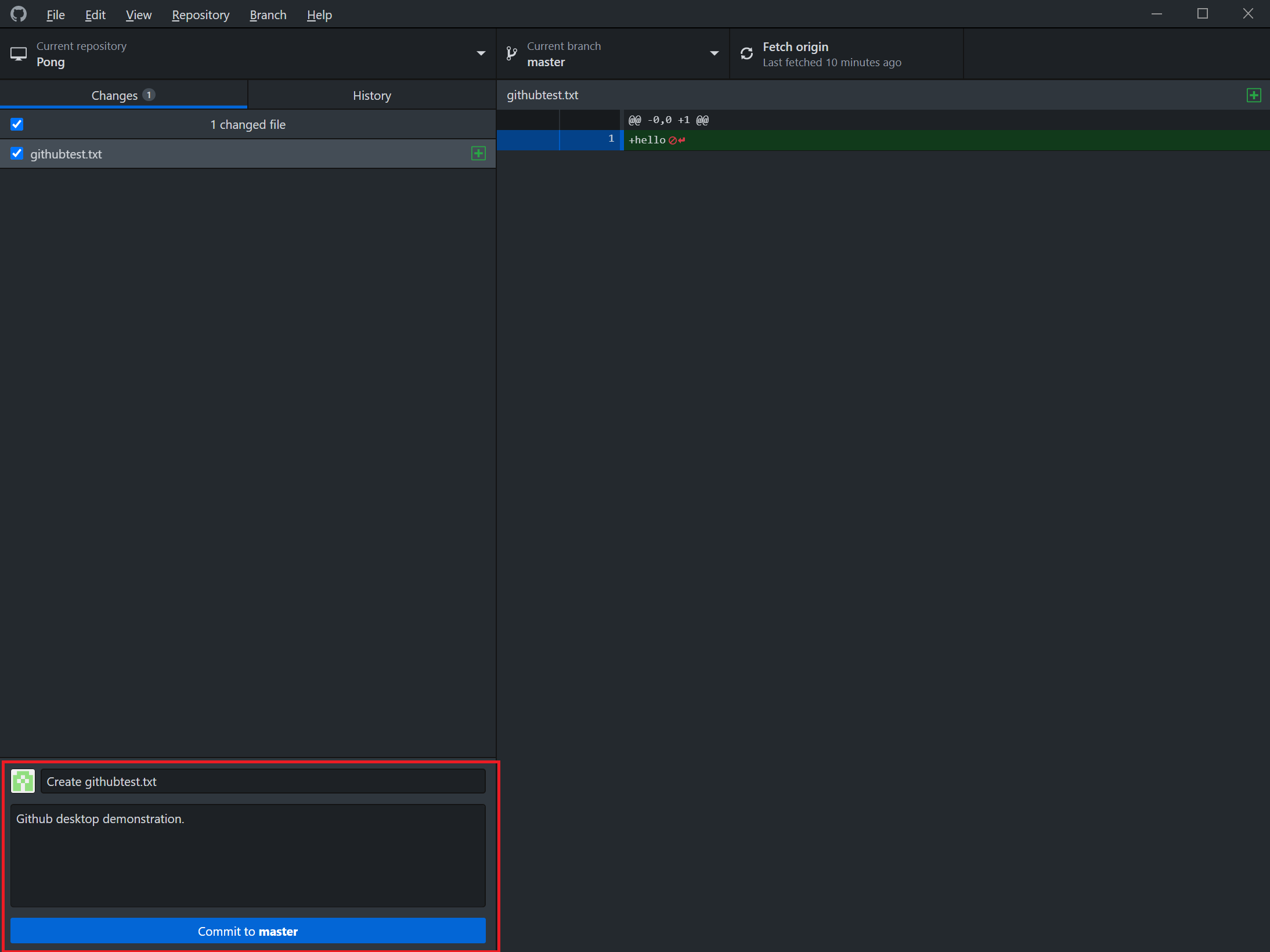


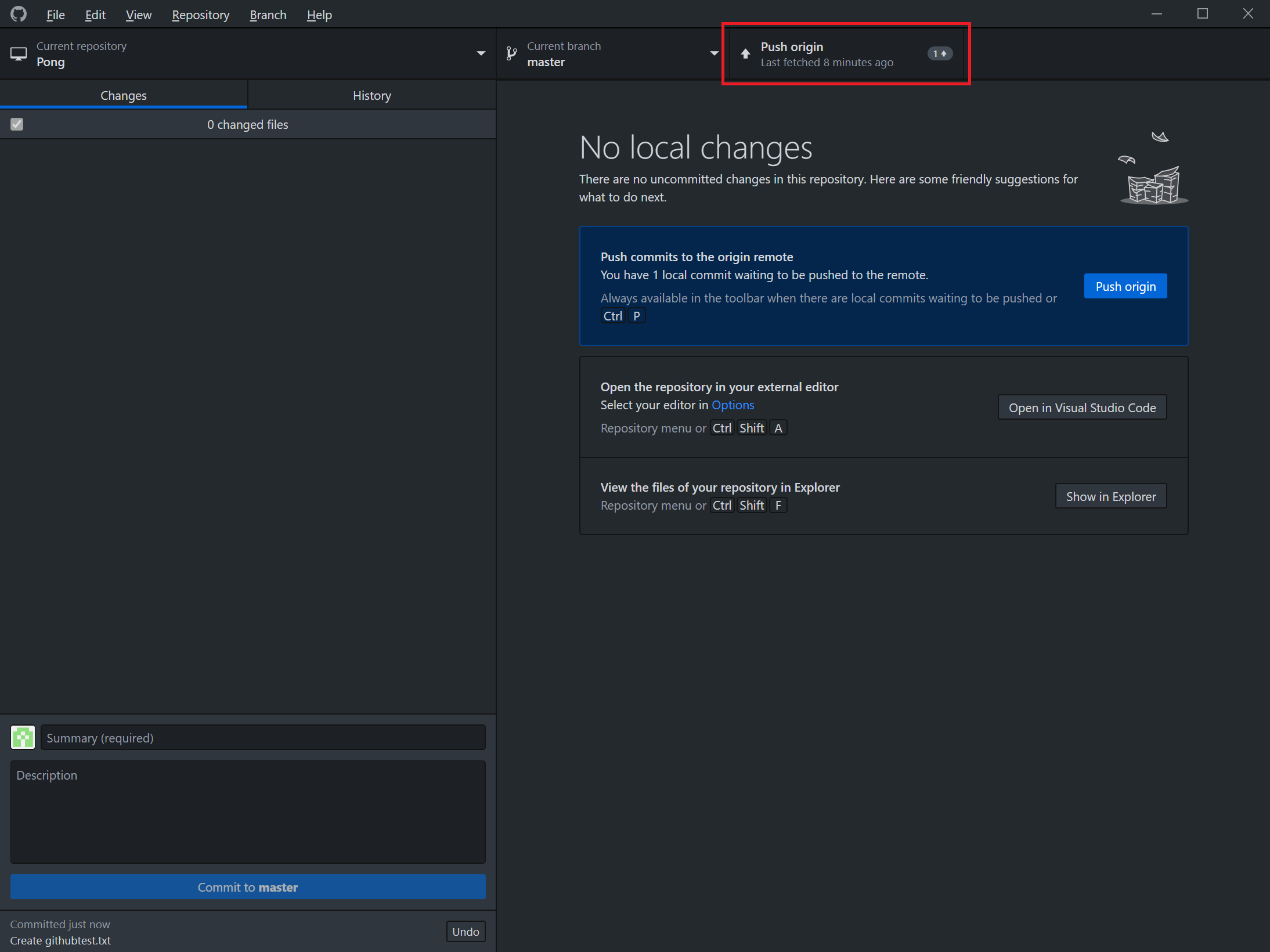
1. You should now have a local copy of the repository in the earlier specified folder. Github desktop will track and display any differences between your selected branch of the repository and your local copy. You may now add, change, or remove files from your local copy of the repository. These changes will remain local until you commit them and then push them back to Github.





1. For this demo I will simply create a new text file in the repository. Github desktop will recognize the change and track it. Provide a name and comment and then commit your changes. This commit is still local and can be undone. Once you are sure your changes are correct, press Push Origin to send your changes to the remote repository so others may access them.



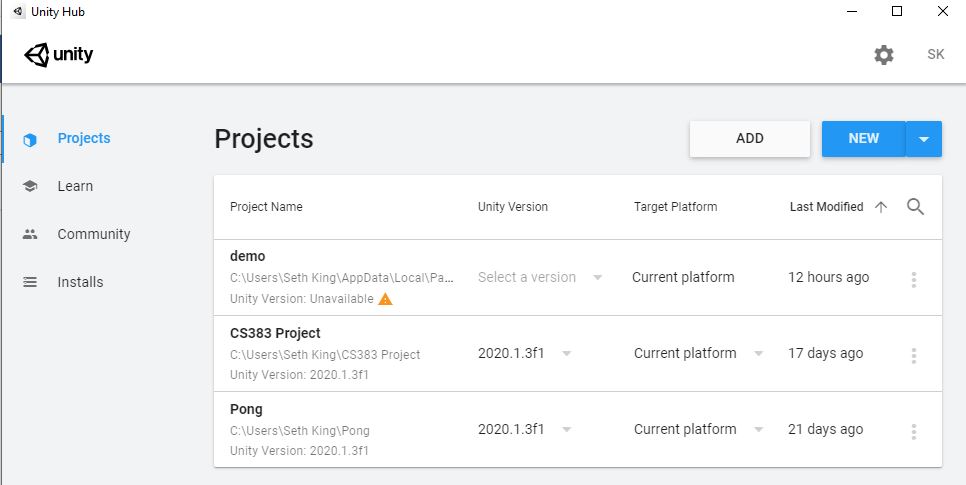


**Creating a Unity Project**

Now that you understand how to use Git for version control, let’s briefly discuss how to create a new Unity project. Many of you have likely already done this for our first assignment, but it is well worth a review—especially if you plan to use WSL as your interface to Github.

To create a new Unity project, do the following:

1. Open the Unity Hub. You should see something like this:



2. In the top right corner, click “new.”

3. Select a name, location, and type (e.g. 2D, 3D) for your project.

**Note:** Unity does not support case-sensitive file systems. If you are using Windows Subsystem for Linux, you will have to place the project somewhere above WSL in your path, so that the file system is not case-sensitive.

**Further Reading**

This manual is a good primer for using Git in CS383, but there is always more to learn, and chances are you will encounter something while using Git that is not featured or discussed here. In that case, you should check out the official Git documentation here:

<https://git-scm.com/doc>

You should also consider making a Github account if you haven’t already. This version control host is very widely used, and is likely the one that you and your team will use for the semester project. Check out Github here:

https://github.com/

Thank you for reading, and good luck with your projects!