# Git Team Lab (Points given based on completing all steps with team)

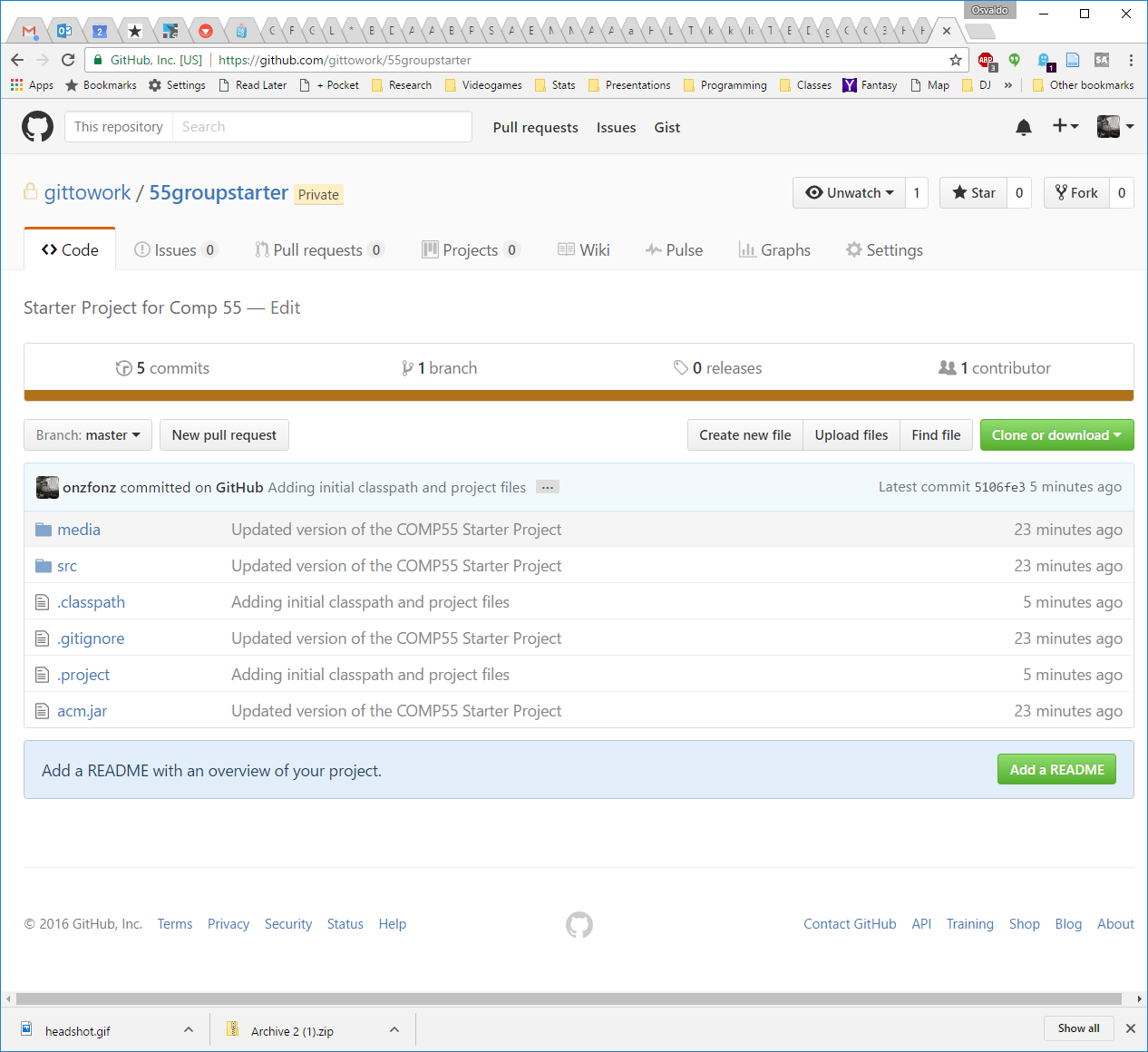
While this is a team lab, each of you will need to do this on your own computer. Your points for this lab will be based off of both your own and your team’s ability to finish this lab, so it’s important to work together. **Make sure that you sit in a way where you can see what everyone else is doing, so that you can provide each other help.** It’s also important that all of you finished the Git Prep Mini-Lab, which was necessary to finish before starting here. **You’ll be using the git bash tool I had you download from the prep lab to issue the commands for this lab.**

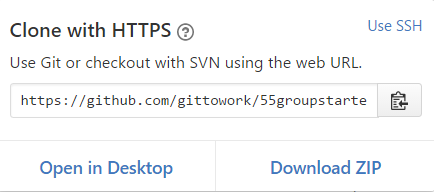
## Step-0

*Setup your computers so you can all see each other’s screens, and decide on labelling the people on your team: 1) git team admin 2) hackslash 3) procrastinator 4) martyr 5) sleepy*

Choose one git team admin – this person should feel comfortable with git, or have a fully working laptop that hasn’t had any issues in the past couple of weeks, the git team admin should also feel comfortable searching for things in google. Once you decide on an admin notify me. The others don’t matter, if you only have 4 people on your team, have the git team admin also server the sleepy role.

## Step-1 (For team admins only – others review the mini-lab)

Team Admins, first sign into github.com. Then separately, sign into classroom.github.com. Once you’ve done both, go to this site: http://j.mp/159teamlab. There you’ll accept the assignment, and it will ask you to create a team. Make the team with your teams’ original name. Once you’ve done that, tell everyone else to do the same steps, sign into github, sign into classroom.github.com and then click on the link. They will then be given to the option to join your team. Once you click to finish the lab, you’ll get the default files that I have provided for all of you for this lab. To get to the URL once you are on your github’s repository, you’ll see a green button called clone or download, once you click on that, it will give you a URL and what looks like a clipboard button, use that or select the URL to copy the entire URL.



Finally, place that URL in your Team’s google doc from last class so they can access it easily.

## Step-2 (for everyone)

Cloning the new project

Once your team admin has posted the URL into slack, you’ll want to clone the new project. To clone the project, you should use the same guidelines that we posted previously about the project, open up the command line, go to the directory where you’d like to have your project stored and clone the project. Use the new URL given to clone the new repository. If you’ve forgotten the steps here, it’s best if you go back and look at the git prep lab. Before moving to step 3 make sure everyone on your team has the group project.

Also you can rename the directory git-team-lab-test on your own computer to something else if you’d like. You can use the mv command to rename something within the bash command line.

mv currentdirname/ newdirname/

## Step-3 (for everyone)

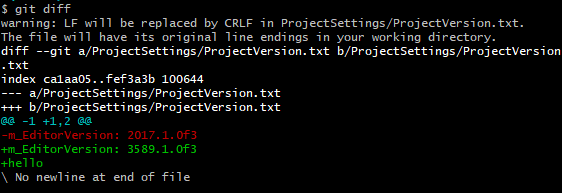
Let’s all make a change and reset

This may seem silly, but part of this will be just to make sure that all of you are used to making small changes and that all of you get practice pulling and making changes to the project.

What I’d like you to do is simply add a line or change the single line that is in the *ProjectVersion.txt* file (inside of the *ProjectSettings* folder). It can be any gibberish you want, and you can make the change in the text editor of your choice. Make sure you don’t spend too much time coming up with the change. The change is not important, what’s important is getting used to using git.

Once all of you have made that change and saved, I want you all to go back to git and do a **git status** command. From there, git will report that the ProjectVersion.txt file has changed.

If you ever want to see what’s different you can simply type **git diff**, if you have the default git color scheme, you’ll see the red changes prefixed with the minus sign as being what was there before and the ones with the + sign as what is there now.



Anytime that you have a git status message that shows files have been modified, you have two choices to make, you can commit those changes or you can either reset the changes. Resetting the changes will revert your files back to the last version committed, thereby losing any of the changes you just made. Since we didn’t make any good changes, let’s all do this right now.

Type **git reset --hard** which will replace your current files with the previous files that were committed. If you check the **git status**, it should say that nothing is different. Doing a reset is useful in situations where you don’t remember what changes you made and you are getting into situations where you are getting errors and don’t feel like you’ve done much with the code. Once all your team members do this and they all have a clean git status, you can move on to step 4.

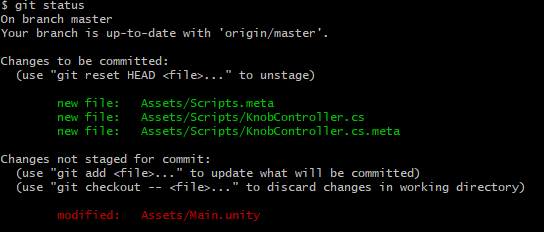
## Step 4 (Martyr to commit, Hackslash to pull afterwards)

Let’s have THE Martyr make a commit

Remember how in grade school you sometimes had a talking stick where the person holding the stick was able to talk? For this lab we’re going to start by doing something similar, where I want your team to take turns making a change, committing the change and pushing it to the repository. To help with this, **make sure everyone remembers their title from the top of this lab.** While one person is holding the talking stick they will be the only ones making changes to code, everyone else should just watch them. Now this isn’t realistic, but once you get used to the workflow then you can try to cause havoc on making changes on this empty project, since you haven’t started anything yet for your project.

Let’s have the martyr make a change to the unity project, making sure they open up the Main.unity scene, add a sprite to the hierarchy (select the knob as the sprite renderer) and attach an empty c# script named KnobController to that object, putting it into a new *scripts* folder. They should also save the scene. Then use the git command line to first add those files and then commit that change. Everyone else should help with figuring out the commands that they need to do and make sure that they added all the files to be uploaded.

If you do a git status after adding all the files, it should look something like this:



One note on committing. When committing and using **git status** you’ll see staged changes and unstaged changes as being two directories that git often refers to.

Staged Changes means that these are changes that are going to be submitted as part of that commit, while unstaged changes are changes that may have been made but that you don’t want the repository to bundle as part of the commit. Any files that you would potentially add to the directory will show up as untracked files which means that git won’t pay attention to them unless you add them specifically.

To add files you simply say

git add NAME\_OF\_FILE\_OR\_DIR

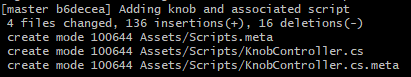
git add \*

A commit needs at least one staged change, but you do not always want to move files that you just modified from unstaged to staged. For example, let’s say you were testing out some code and added a picture of yourself to the project so that you could use it. You may not necessarily want that picture to be added into git and stored on github as once you add it to the repository, it’s a bit more difficult to get it out. Another thing could be certain development preferences that you have like having a dark color theme. Git is agnostic about tracking changes it does not know much about your project, all it does is checks all the directories in your git repository for new files or files that are different than what’s on the server and asks if you want to commit them. If you don’t want git to track certain files, you can ask git to **ignore** those files. On the other hand, your c# files and particular files that aren’t in the library should typically be in staged changes everytime you commit though, as you want to git to keep track of those changes in each commit.

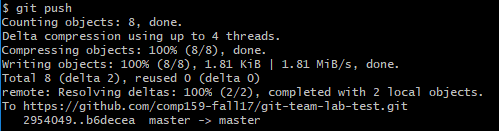
Remember that when you commit, you will have to enter a commit message which is just a simple message that describes what the change they made was. It’s easiest for this first change for the martyr to say something like:

git commit –am “Adding knob and associated script”

They should then get some type of message with a new id and the changes that were made, like below which is just a sample.



Once the martyr commits and then pushes (pushing is done by saying **git push**), if it was successful they’ll get a message like this (not this exact one).

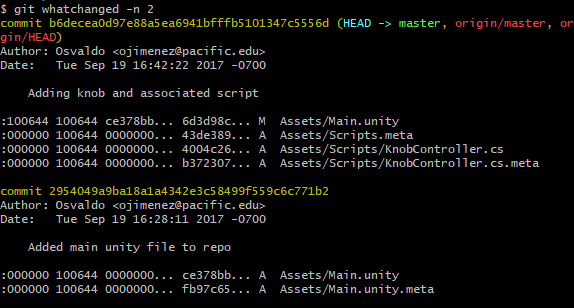


It’s important to look at the resulting message, since if it fails it’s going to give you a message saying something went wrong. We’ll get into this more in a later step. Everything else will just have information on the date of the commit, when it was, and what repository it was for. Once you’re finished clicking the message, you can click OK.

Once the martyr pushes, the hackslash should pull the project from the git command line by saying **git pull**. If hackslash does have files that were modified, have them do a hard reset like we did in step 2 before pulling, (they can check if they have files that are modified by doing a git status before pulling.

## Step 5: Looking at the history (Everyone)

Everyone can see a history of what has changed by typing **git whatchanged** (wait to do this). This will bring up the history of commits, along with the authors and the little messages detailing what happened. If you want to limit the number of changes to show, you can use the **–n** and then a number which will show the last **n** commits (eg **git whatchanged -n 2**). If you get into a place where you don’t have the command prompts available anymore, you can press ***space*** until you reach the end of the changes. Alternatively you can press ***q*** at anytime to exit looking at the log and get back to the command prompt.



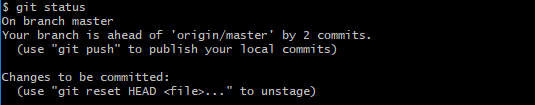
In this particular situation, I actually prefer to use a GUI tool or to look at the history on github, since I start with an overview and then can click to get more details. The history will show you the list of all the changes that were made for a particular project and when they were committed, along with an ID that you can just think of as being a version number for that particular file. Looking at the commits tab of that page and then clicking on individual repository will show you the additions and deletions that were made to that file as part of that commit. Git will keep track for you line by line of each change on each file that was made as part of that commit.

When you do the **git whatchanged**, also notice that the hackslash & martyr have different commits than everyone else. You can verify with your team members that their history might have a different commit ids in yellow at the top, depending on whether they pulled or committed or did nothing. That is because git uses your local copy of the repository to bring up the history. It does not give you the latest version unless you ask it to fetch the latest version or pull. **At this point let’s have everyone on the project do a git pull to receive the latest changes.** If you do a git whatchanged –n 2 at this point, you’ll notice that the change log will now be updated with the latest changes.

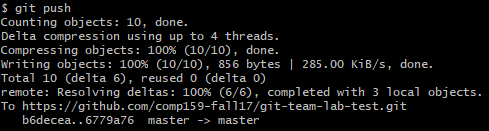
## Step 6:

Hackslash commits multiple times and then pushes

Let’s have the hackslash, who just pulled in the previous step, now add a comment at the top of that c# file, **but not push. They should do this without opening Unity.** After committing, let’s have the hackslash **add an additional comment** to the beginning of c# file and commit again without pushing. If the hackslash did push that’s ok, just start the process over. What the hackslash should have is a series of two commits that have not been pushed onto the server, where the changes are only in the comments at the top. Hackslash should also not edit any other lines! Once the hackslash has this, they should use git log to verify that they have two commits that their teammates don’t have any. The other option would be to have the hackslash just use git status to read the status message up top, right after git status.



Once the hackslash is two commits ahead, they can then do a **git push**. Doing a push will take those local commits that they have on their computer and send them over to the github server, making them visible to everyone. Hackslash, make sure you get a message that does not say it was rejected, but rather, some message saying you were successful like this



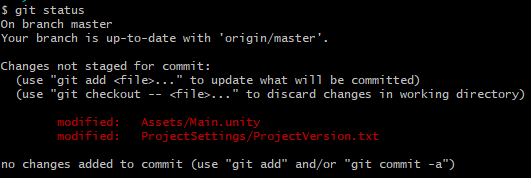
About committing…You should think of committing as being like a meta-save. Generally you want to commit every time you add some new feature into your code that works. If you get something to work or there is new functionality then you should commit to describe what it is that you added. **For the purposes of this class I’m going to ask you to commit very often.** Committing code is going to be an important factor in your grade. While you shouldn’t commit after doing every line or every move of a gameobject like I’m going to ask you to do here, I would say once you’ve added something that makes the game different than what you had before (say a difference or change of 20 or 30 lines of code or a just a new piece of functionality, animation etc) it would make sense to commit. It makes sense that when you have something that can be described in a short message as being different than what you had before, you should commit it. Then after having several commits, once you have something that you feel is working well enough that it won’t break other people’s code and is bug free and that they should now have, that is when you should push the code to github.

Once hacklash has pushed, have the martyr pull the latest results, and verify that they have the same history. Notice that when the martyr pulled, their version of the code was replaced with the newest version. This is what generally happens when you **pull**, it will not only fetch the information from the server, but also change your code so that you have the latest changes in your project’s files. Luckily Unity will detect when the files have changed and just ask you to reload, but you may think about just having unity closed when pulling**.** So far the hackslash and martyr should have the latest changes, and the others should not. This is important before moving on, if not you can just repeat this step to make additional changes to the code. No need to revert back.

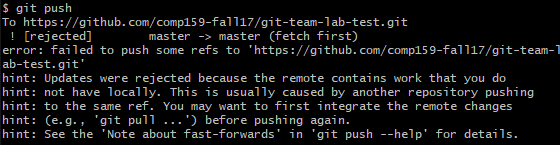
## Step 7

Let’s have the procrastinator push changes

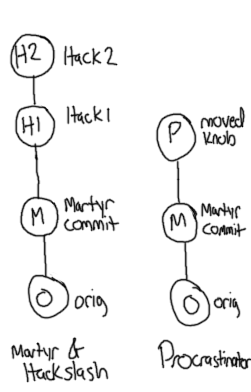
Now it’s the procrastinator’s turn. First let’s have them open up unity. They may get some messages about reimporting assets, which is fine. Then once unity is open, they need to open the Main.unity scene. With the scene open, have them move the knob object to have an x value of 6. Then save the scene and run a git status. They should get something like this:

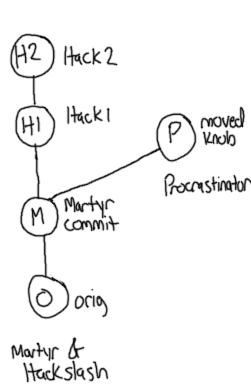


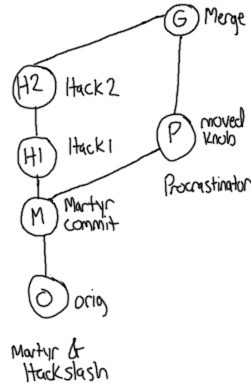
While it makes sense that you changed the Main.unity file, one that might seem weird (which you may or may not have is the change in the ProjectVersion.txt) ProjectVersion.txt is a specific file that Unity uses to record which version of unity is being used to make the project. I’ve seen some people discuss that you should all be using the same version of Unity when working on a project. I believe that it uses that project number to be able to read the contents of the main.unity file. If you don’t have that modified file that’s ok, that just means you have the same project version. Go ahead and have the procrastinator do a commit saying you moved the knob. That will show as being a success. Once you’ve committed, go ahead and then try to push your commit. At this point you should get a message like this one.



Why was it rejected? Let’s have a quick drawing to see what’s going on here.

Think of the two sides here as being a simplified view of the two competing histories. On the one hand the left side represents the history of commits that the server has, as well as martyr & hackslash. For example, first everyone was on the original version. Then martyr ended up making a commit based off of that version which would be the little M and it’s based off of that original commit. Everyone pulled that version, including the hackslash, who then made 2 commits that were based off of martyr’s changes (which we are calling H1 and H2). Meanwhile, the procrastinator, ended up committing a change that was based off of martyr’s commit. When the procrastinator tried to push their changes, the server noticed that the procrastinator’s newest change was based on the martyr, but that they did not have the latest changes on the server, and therefore rejected the change, saying it couldn’t update the server since they don’t have the latest information. A more accurate representation of this graph, which we’ll get to is show below:

Notice that in this drawing we are more accurately reflecting that the procrastinator and the martyr commits are both based off of martyr’s commit. Because the procrastinator’s build is not based off of the latest version, git won’t allow it to say that it is in fact the latest version. This is completely reasonable behavior, and so you should not worry as much about getting into this situation but rather just know how to get out. The most important thing is not to panic but to know how to resolve this situation. To resolve this, what the procrastinator is going to have to do is to pull the code (**git pull**). Because they have already committed, git will do its darndest to do the right thing and merge the procrastinator’s changes with the hack’s changes and create a new version based on the changes. That new change or the merge would theoretically look like the following graph below.

As you can see what git is doing is basing the new commit off of the two changes, what was on the server and the changes that the procrastinator did. As mentioned previously when you pull, git will automatically try to create a merged version. Based off of the merge it might be able to merge the two commits completely by itself, or it may need your help. When the changes are done on completely different files, git does not have much trouble merging, since it will just use the latest version of the file from each commit. If the changes are on the same file, if git has an idea of where those lines are git will also do its best to keep the changes from each version, as you can see from this pull. However, if it needs more information from you to make the decision, you will enter into what’s considered a merge mode since it needs to know what version is the latest based on your decisions. In this case, since the procrastinator pulled if they were to get conflicts then they would be the ones making the decision as to which lines of the file are the most current. We will get ourselves into that mess in the next step, as it’s important for all of you to be aware of and to at least try it. If you did you yourself into that mess already, congratulations! You can move on to the next step, just read this part here until you get your back to the command prompt.

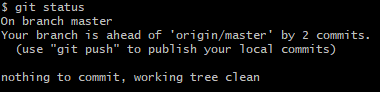
Most of you procrastinators however, should after doing that **git pull**, have something like this



What git has done here is to take you to the commit screen. Normally we do not enter the commit screen because we have been using –m to give your commits a message instead of entering this, but if you didn’t change your settings, you will be entered into the vim editor so that you can amend your next commit. In this situation git was able to successfully merge the two sets of commits. Because of this, it wants you to commit that merge right away, so that you know that what is coming is a combination of the two commits and to have that be recorded. The procrastinator in this case can just leave the message as is or can help explain why the merge happened. If you are going to modify the message and have this editor, to insert in vim, press ***i***, and then start typing. To save and exit in vim, you should press the **esc** key and then simply type **:wq**, which should appear at the bottom of the command screen, then press enter. Once you do that, you’ll get a successful commit, showing the merge.

In this case, the Procrastinator, will now have the representation on their own machine like the graph that I placed up at the top, they can confirm this by showing the history of the git commits in their history (**git log –n 3** should be enough)

The procrastinator will type **git status**. In that status, they’ll receive some information about where they are in relation to what’s on the server, for example if they had 3 commits done, they would get a message like this. They should verify they got a result similar to this (though there’s will most likely say 2).



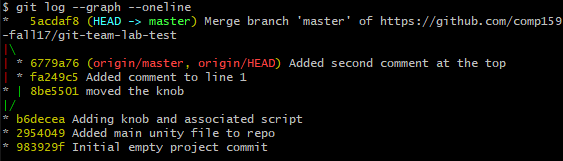
If the procrastinator wants to now push their files to the server, they will be able to by simply typing **git push**. This time because they were able to merge the changes that occurred while they were working on the other ones, the server knows that they have the history of all the changes, and so the push will succeed (unless someone else pushes additional code while they are trying to merge)

## Step 8 - Committing and then checking the status

Now the rest of the team can pull the latest changes, using **git status** to make sure they don’t have any unstaged changes. Once they pull they can also look type

git log --graph --oneline

To see the changes and how they’ve been introduced

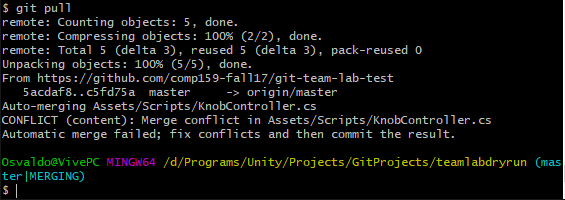


If you notice the left most side, it shows an ASCII graph that shows a similar diverging graph to the one that I showed you up top in Step 7. In this case, the two lines are flipped however, with the hackslash having the two commits on the right and the procrastinator having the commit on the left, which you can confirm via commits.

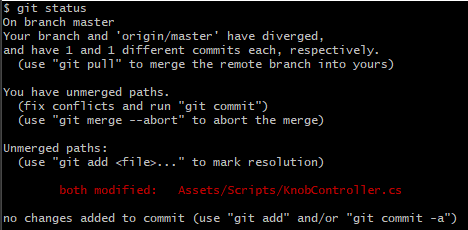
## Step 9

Let’s cause some havoc

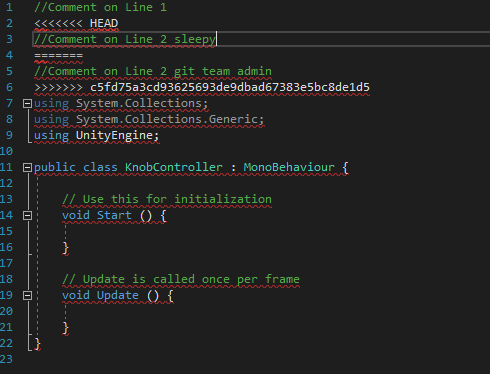
So at this point you either had success getting git to merge something for you or git failed at being able to merge things for you. If git was successful in merging, let’s have you get into a situation where git can’t figure out what the latest version or the best lines to have in the file are. To do this we’re going to set it up so that the git team admin and sleepy try to modify the same line in the same file. Ensuring both of them have pulled the latest changes from the server, have both sleepy and the martyr change line 2 in KnobController.cs. The easiest thing to do in this case is to change the comment by adding in your name or your given title for this lab. You can use your own real name or you can use the name I’ve assigned you. Then go ahead and commit. Let’s have the git team admin push their commit to the server and then try to have sleepy push theirs. Sleepy will notice that they will be rejected from pushing their changes and asked to pull. When they pull to get the latest changes from the server, they’ll notice that the result will be slightly different, where they’ll get a merging state.



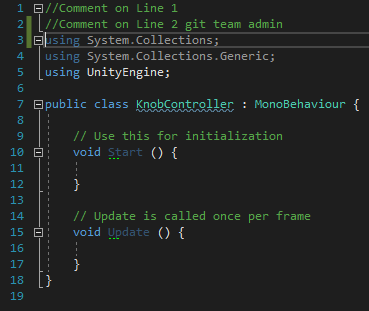
Doing a git status at this point will give you a somewhat scarier sounding message:



This is git letting you know that you have conflicts or changes that it doesn’t know how to resolve, with the line *both modified* being placed on the files that it cannot resolve. Because it doesn’t know how to resolve the changes, you will be tasked with solving them and updating the server. If Sleepy does not have KnobController.cs open already they should go ahead and open it up. When they open it, they’ll notice a lot of extra text! Git has added text to indicate where in the text file it needs help resolving issues, when you open it up it should look something like this:



The important parts to look at are lines 3 and 5. Line 3 shows the version of the file that you did, while line 5 shows the version of the line that was on the server. When it doesn’t know which change is right, git will use this notation of having the <<<<<<< HEAD in there to let you know you need to make a decision. Everything between the HEAD notation and the line of equals signs on line 4 are the changes you made, and everything below the line of equals and up to the >>>>> branch label are the changes that the server has. If you are the one in the conflicting state, it’s your job to decide whether to keep the code above the equals sign, below the equals sign or some mix of the two. Once you decide what lines are the right ones, you will go ahead and delete the rest of the markings so that the errors go away and it looks like a C# file. In this case, let’s use what was on the server instead. This means that after editing and saving the file, it should look something like this.



While the syntax errors went away, you’ll notice that sleepy will still be in Merging state. To get rid of the merging state, the git team admin will need to commit and push the changes to update the server, just like they have done before:

git commit -am “Fixed the conflict”

git push

If this all sounds scary to you, then you should certainly have your team members with you to decide what is the best version when you have conflicts, though if they committed changes and they disappear, those changes will always be tracked and they are viewable so that you can get them and change your code back if needed.

## Step 10

Wreak More Havoc

This part about resolving conflicts is so integral that I now want you to as a team to all try to make a change on the same line and go through the process of each of you resolving the changes being in that conflict mode and having to re-add the file and commit and push. I’d also like for you to try this using unity, having two people change the same game object, introducing new game objects at the same time to try to cause a conflict, etc. Part of your credit from this lab will be me looking at your git history and seeing that all of you have tried to do some craziness, so make sure to help each other along the way. Since sleepy had to manage a bunch of conflicts in the last step, have them push first so that they don’t have to do the process again. Also make sure that no one tries to pull after each push, assume they don’t know the changes have been pushed. So that the last person will be possibly many commits behind everyone else. This is the time to understand how to get through the git process in a throwaway project before it starts to count, so make sure you make lots of commits and try to break git in using unity. Work together and if you get into trouble, make sure you leverage each other as well as ask me.

One type of havoc that I’d like you to try is to **make a conflict with the Unity scene** (Main.unity). While this may seem scary at first, it turns out that unity files are text files, and so the Main.unity file is really like a save state file, so you can go in and edit the text as if you were merging conflicting code. Go ahead and give that a try, fix the conflicting unity file and then append it again.

The last type of havoc that I want you to try is to **make a conflict with binary files**, such as png files, mp3’s, etc. Once you get to a conflict, then while you are in merging mode, you’ll want to use an ours or theirs flag to help you replace the file that is conflicting with the correct version.

To use your own file, use the ours tag

git checkout --ours -- path/to/conflicted-file.txt

To use the one on the server use the theirs tag

git checkout --theirs -- path/to/conflicted-file.txt

Once you replace the file, you will still have to add the file again (**git add path/to/conflicted-file.txt**) and then commit and push to have those changes be reflected on the server.

## Step 11

Taking out things from gitignore that shouldn’t be there.

While this is the end of the official lab, I want you to be aware of one more issue that seems to pop up a lot when folks are using git, which is when people add things to git that shouldn’t be there. This tends to happen when folks aren’t careful of what they have added to the repository. The sad thing is that even if you add something to be ignored by adding it to the ignore file, once it’s in the repository git will ignore you .gitignore file. To take those files out of the repository entirely, and remove them from your directory, you can do this:

1. Update your .gitignore file.
2. git rm –r --cached .
3. git add .
4. git commit -am “Remove ignored files”

Your other option would be if you don’t want your changes to the file that is in gitignore to be continually updated, you can use this command instead, which won’t delete the file from the repository, but won’t keep updating the repository every time it changes on your computer. For this to happen you need to type this.

git update-index --assume-unchanged <file>

If you want that file to not be updated anymore, but still want it in the repository, then everyone would have to issue the command uptop.

## Step 12

Practice the workflow!

Based on the lab, I think it’s important to remember how to work on the project. One thing that I’ve been hearing over and over again with Unity is that it may make sense for only one person to work on one gameobject or prefab at a time (don’t both work on the same gameobject). Also generally, before you start working on the project, make sure to pull to get the latest changes **while Unity is closed**. The easiest way to know you are up to date is just to check. Then you go ahead and commit commit and commit again. Once you are finally ready then push your changes. If that push fails, you’ll need to do a pull to merge with your team members changes. Once you merge and commit, you should then push your changes. There are more advanced ways of using git, but this is the way that I’m going to be looking for you to use git in this class, as simply getting used to source control when working on projects is a big undertaking. For the rest of this lab you guys should just get into the habit of trying to quickly modify the file that you have and making changes to it together. This is your chance to break things! You should also look into using branches in this git lab. Branches work exactly the same as the graphs I showed you up top. The only difference is that the branch has a label which means that you don’t necessarily need to merge all the time when things diverge, you can just create a branch, and switching to another branch is very easy. For this lab, I’ll also want you to practice making a branch and committing on two different branches. For more info on branching you can take a look here:

<https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging>

Also if any of this confuses you, you may start searching google for more resources to learn git. There is some good stuff out there (and also lots of bad stuff). Most of it revolves around using the command line, which is why we’ve switched to using it here. If you prefer to use a GUI based solution, you can do so at your own willingness, however, the command line will be the most flexible (but also the most unforgiving)

Let me know what still confuses you so I can make this lab better in the future!