

GIT FOR UNITY

This is an ungraded team assignment. Each team should submit a memo detailing how the exercise went. It is important that all team members participate to ensure everyone knows how to contribute to the project!


If your team plans to use a version control system other than git then customize this exercise to the appropriate tool(s).

Steps 1-7 are to be performed by one person only. In case you're not familiar with git, have a look at [this](#) interactive tutorial as a good starting point.


1. Visit Georgia Tech's Github page or the Git repository manager of your choice (github.gatech.edu – Note: all team members need to log into github.gatech.edu at least once before you can add them to the dev team)
2. Create a new repository. Only one person in the team needs to do this. In the Add .gitignore section select Unity. If you don't do this, files that are not needed to be tracked will also be committed. This can cause issues across different systems and OSs. In case you forget to do this, you can find the Unity .gitignore file [here](#). However, it's best to do this when you're setting up the repo so that you don't have to 'untrack' all of these files later. Also, ensure that your repository is 'Private'.

Owner

Repository name *


 **vgargya3** ▾

 /


GitPractice 

Great repository names are short and memorable. Need inspiration? How about [jubilant-parakeet?](#)

Description (optional)

☐  **Public**

Any logged in user can see this repository. You choose who can commit.

☒  **Private**

You choose who can see and commit to this repository.

Skip this step if you're importing an existing repository.

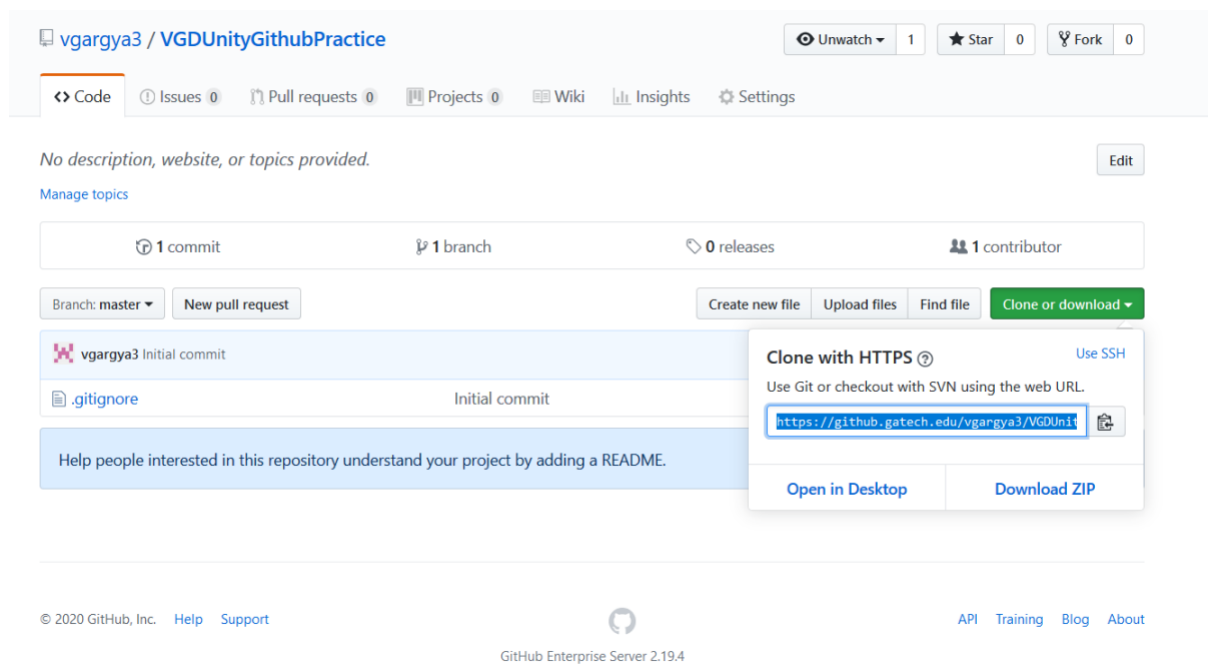
☐ **Initialize this repository with a README**

This will let you immediately clone the repository to your computer.

Add .gitignore: **Unity** ▾

Create repository

3. Now that you have a repository, you need to clone it onto your machine. Get your



repos address and go to a folder of your choice on your machine. Open the git terminal on this machine or a git gui and clone it using the command
`git clone <address_to_your_repo.git>`

4. Open Unity Hub and create a new project to a temporary folder. For example, if I cloned my repo in a folder called 'Folder A', set the Unity project home folder to 'temp_prj'.
5. Save the project and exit unity. Using your OS file browser, move all the Unity project files in the temp folder (e.g. 'temp_prj') into the git project dir (e.g. 'Folder A').
6. Use Unity Hub to add an existing project and point at Folder A. Open the project. (You might want to also remove the reference to the old location with the temporary folder in Unity Hub.)
7. Expose meta files - From the Unity documentation - "Before checking your project in, you have to tell Unity to modify the project structure slightly to make it compatible with storing assets in an external version control system. This is done by selecting **Edit->Project Settings->Editor** in the application menu and enabling External Version Control support by selecting **Visible Meta Files** in the dropdown for Version Control. This will show a text file for every asset in the Assets directory containing the necessary bookkeeping information required by Unity. The files will have a .meta file extension with the first part being the full file name of the asset it is associated with. Moving and renaming assets within Unity should also update the relevant .meta files. However, if you move or rename assets from an external tool, make sure to synchronize the relevant .meta files as well."
8. Add a 3D game object in your scene (Sphere/Cube) and save your project
9. Open the git terminal or git GUI and stage all changed files. This can be done by typing in the command '`git add .`'. Commit these files and add a commit message. Now push this to your repo on the web using '`git push`'. In case you get an error saying there was no remote directory to push to, you may have to add a link to your remote repo. This can be done using the command

```
git remote add upstream <REMOTE_URL>
```

And then push using

```
git push upstream main
```

(Note: You may wish to use github Desktop application for a visual UI instead. And even if you intend to use the shell commands, you might find installing github Desktop and then install its Command Line Tool is the easiest way to get ssh features working correctly from a shell.)

10. Now your initial project is set up and can be cloned by other team members. You can confirm that all the files that you added are in your repo by visiting your remote repo
11. At this point each member of your team can clone the repo to their local machine using the git clone command described in step 3.

12. For development, we recommend using the feature branch workflow.

<https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow>

Each issue that you work on should be worked on in a new branch. Branching in git is easy. Have a look at [this](#) lesson.

13. Create and checkout a new branch in your local repository (Branch1 is the name, you can modify it to what you want)

```
git checkout -b branch1
```

14. Create a new script. In this script, add a print statement to the start method make some modifications to the scene and save your project.

```
public class NewScript : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {
        print("Hello");
    }

    // Update is called once per frame
    void Update()
    {
    }
}
```

Now add and commit these files on your branch using

```
git add .
```

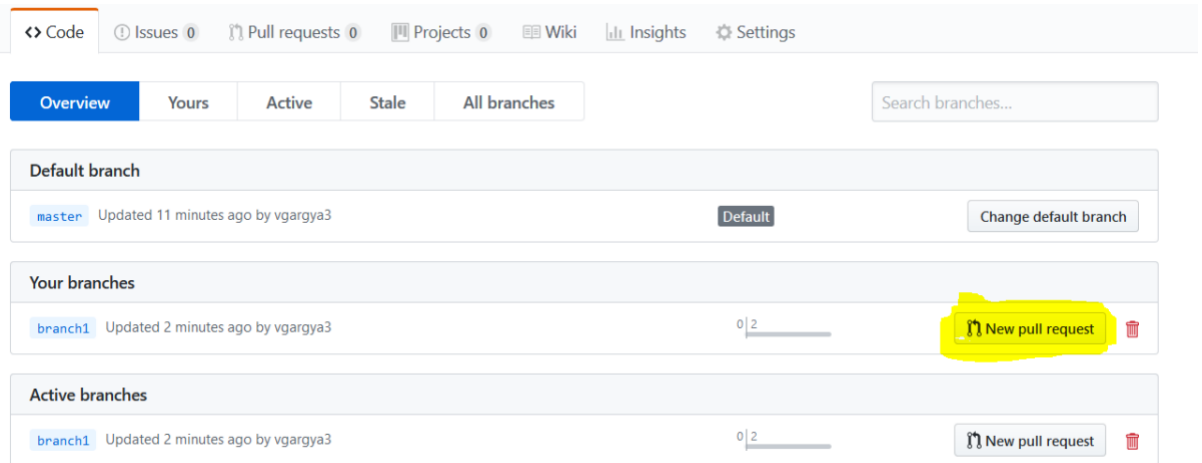
```
git commit -m "Branch1, commit 1"
```

15. Your changes are now active on your branch, but they will not be available in the main branch (note: some old projects have main branch named "master" and screenshots below will show this old name). You can verify this by checking out the main branch.
16. You now need to push your new branch to the remote repository. You can do this using

```
git push -u origin branch1
```

If you visit your github page now, you should see two branches in your repository.

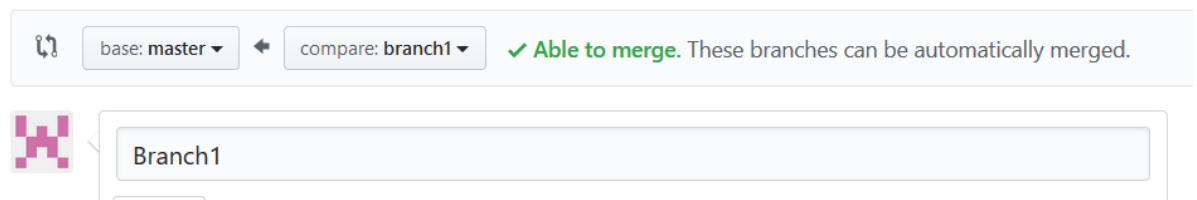
17. Create a new “pull-request” for your code changes.



18. In the pull request you can describe what changes are in your branch. Since we’re merging into main, git compares the two branches to see if any simultaneous edits were made to them. If not, it can merge these files cleanly. You can create the pull request and merge it with the main branch with no issues.

Open a pull request

Create a new pull request by comparing changes across two branches. If you need to, you can also [compare across forks](#).



19. However, if simultaneous edits were made to the same file things would not go so smoothly. Let’s see how to get around this problem. First save and merge this pull request into your main branch. Your main branch is now up to date with your new branch’s changes. Go to your local repository and pull these changes

git checkout main

git pull

20. Create and checkout a new branch, branch2. Modify the print statement in your script to “print(“Hello world”);”.

```
public class NewScript : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {
        print("Hello world");
    }

    // Update is called once per frame
    void Update()
    {
    }
}
```

Stage and commit this change on your branch2.

21. Now switch to branch1,

git checkout branch1

And in the same script make another change.

```
public class NewScript : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {
        print("Hi world");
    }

    // Update is called once per frame
    void Update()
    {
    }
}
```

Stage and commit this change as well

22. Here we have two changes on the same script but in different branches. Both the branches are unaware of these changes. Let's push the changes in branch1 to remote

git push

And do the same for branch2, while setting up its remote tracking

git push -u origin branch2

23. You should now have 3 branches in your remote. Let's merge the changes from branch1 first into main. Create and merge a pull request. It should merge with no problems.

Default branch			
master	Updated 7 minutes ago by vgargya3	Default	Change default branch
Your branches			
branch2	Updated 5 minutes ago by vgargya3	0 1	New pull request
branch1	Updated 2 minutes ago by vgargya3	1 1	New pull request


24. After this is done, create a pull request for branch2

Default branch			
master	Updated 11 seconds ago by vgargya3	Default	Change default branch
Your branches			
branch2	Updated 6 minutes ago by vgargya3	2 1	New pull request
branch1	Updated 3 minutes ago by vgargya3	2 0	#2 Merged

Github will not be able to automatically merge this branch

Open a pull request

Create a new pull request by comparing changes across two branches. If you need to, you can also [compare across forks](#).

 base: master ← compare: branch2 ✗ Can't automatically merge. Don't worry, you can still create the pull request.

25. To get around this problem, you have to update branch2 with the latest commits from main. To do this, go to your local repo, checkout main, pull latest changes.

```
git checkout main
git pull
```

Now checkout branch2, and merge main into it

```
git checkout branch2
git merge main
```

This should show you a merge conflict,

```
vinay@DESKTOP-709PVM0 MINGW64 ~/Documents/GT/CS 4455 TA VGD/VGDUnityGithubPracti
ce2/GitPractice (branch2)
$ git merge master
Auto-merging GitPractice/Assets/Scenes/NewScript.cs
CONFLICT (content): Merge conflict in GitPractice/Assets/Scenes/NewScript.cs
Automatic merge failed; fix conflicts and then commit the result.
```

26. Go to the file in question, in this case new script and incorporate the changes that you want the merged file to have

```
public class NewScript : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {
        <<<<<<< HEAD
        print("Hello world");
        =====
        print("Hi world");
        >>>>>>> master
    }
}
```

If I want the changes from branch1 to take over, I should remove the code from <<<<HEAD to ===== and the other >>>>>main pointer too. I could also choose to incorporate both changes by just removing the git markers.

```
public class NewScript : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {
        print("Hello world");
        print("Hi world");
    }

    // Update is called once per frame
    void Update()
    {
    }
}
```

Stage this change and commit it using

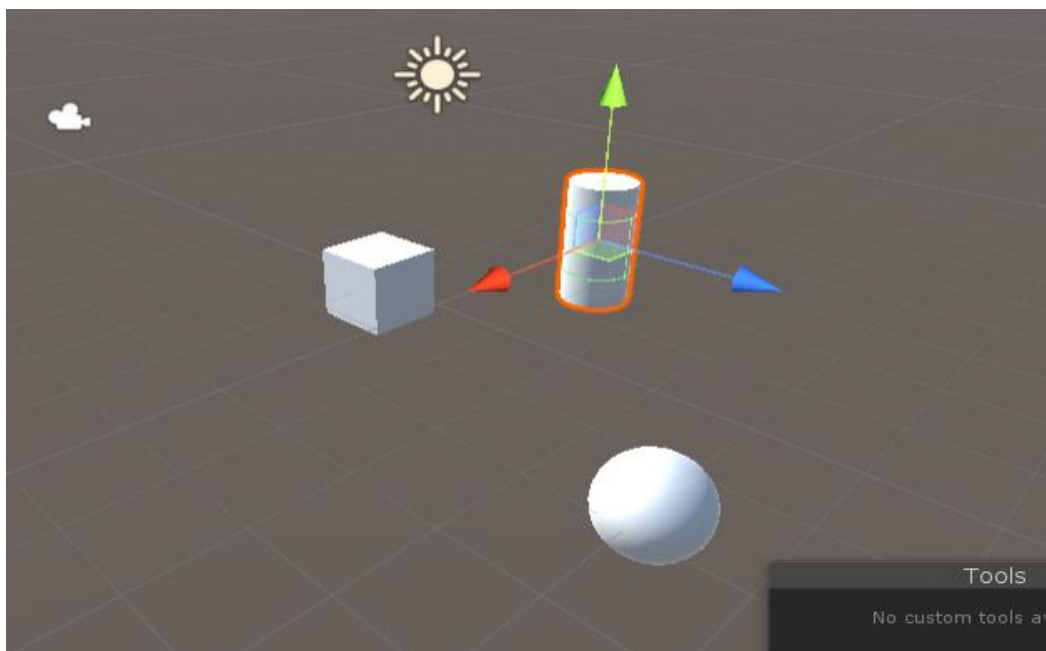
```
git add .
git commit
```

Now your branch is updated with the latest changes from branch1. You can now push this to the remote, git push and then create a pull request

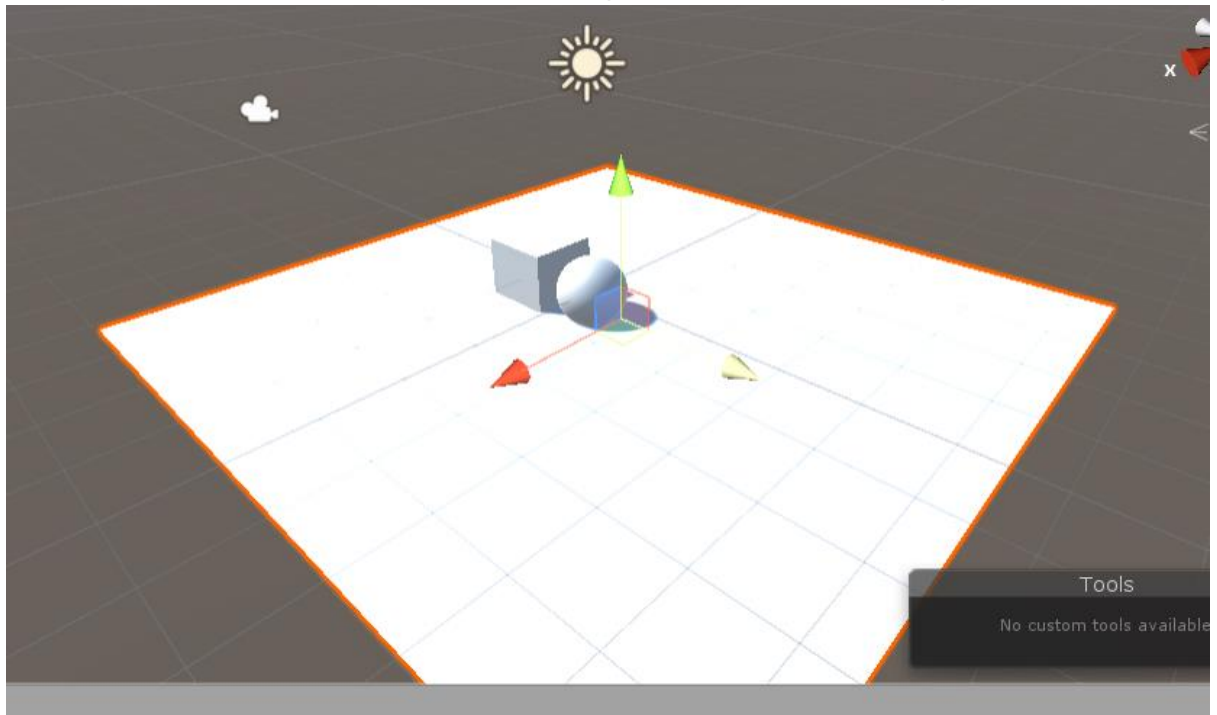
TROUBLESHOOTING (Optional)

This is an optional section in case you want to see what problems can occur when you merge scenes. Read point 25 even if you don't plan on completing this section

27. The main issue with using Git with Unity is the merging of your scene file. In order to avoid this situation, we recommend **that only one person in your team edit it**. If you can't make that work, you **should always 'pull' the latest code, work on your scene file and then push it immediately while informing your team that you've done so**. Merging scene files in Unity is a real hassle and should be avoided entirely and in order to avoid wasting your work, don't simultaneously edit scene files. In case you want to test some changes, have an **individual scene file for each of your team members**. In case you've already made some changes that you don't want to lose as a result of merging, you can **duplicate your scene file on your local machine, get the latest scene file and then copy over these changes by manually applying them again in Unity**. Modularise your work by **creating Prefabs** that can be added to the scene plug and play style without requiring much work in the scene. You can also create several small scenes and switch between them at runtime with trigger zones via `SceneManager.LoadScene()`. Look at the [Smart Merge](#) tool if you want to merge scene files.
28. We'll try to simultaneously edit the same scene file and see the problem that we encounter. Here Alice added a new cylinder to the scene and changed the position of some objects. Shed saved these changes and committed them to her local repository but did not push it to the remote.



29. Another member of the team, Bob, opened the same scene file and they added a Plane to the scene, saved it, committed it and pushed it to the remote repo



30. Now Alice has stale version of the scene and when she tries to push to the remote, she gets the following error.

```
$ git push
To https://github.gatech.edu/vgargya3/VGDUnityGithubPractice.git
! [rejected]        master -> master (fetch first)
error: failed to push some refs to 'https://github.gatech.edu/vgargya3/VGDUnityGithubPractice.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

31. She tries to pull the latest changes and gets a merge conflict in her scene file

```
Auto-merging GitPractice/Assets/Scenes/SampleScene.unity
CONFLICT (content): Merge conflict in GitPractice/Assets/Scenes/SampleScene.unity
Automatic merge failed; fix conflicts and then commit the result.
```

32. There are different ways of getting around this problem. Alice should copy her current scene file and save it in a different location outside the repo. She should then abort the merge with the command

`git merge --abort`

Revert to a last known good commit which she knows would not have merge conflicts. She can do this by running

`git log`


```

commit 332db80f938eb27ad45c219d20a97f3c31c84e55 (HEAD -> master)
Author: vgargya3 <vgargya3@gatech.edu>
Date: Sat Feb 1 13:16:50 2020 -0500

    Alice changed the scene

commit 70c99f2b696d8701c3579a7ee10e6a0d8990caaa (upstream/master)
Author: vgargya3 <vgargya3@gatech.edu>
Date: Sat Feb 1 12:29:12 2020 -0500

    Initial Commit

commit 77f1caaba64bd16b9a21dfd17700c9462e47f903
Author: Gargya, Vinayak <vgargya3@gatech.edu>
Date: Sat Feb 1 12:18:58 2020 -0500

    Initial commit

vinay@DESKTOP-709PVM0 MINGW64 ~/Documents/GT/CS 4455 TA VGD/VGDUnityGithubPracti
ce (master)
$ git reset 70c99f2b696d8701c3579a7ee10e6a0d8990caaa
Unstaged changes after reset:
M   GitPractice/Assets/Scenes/SampleScene.unity

```

She knows that the commit highlighted is a good one and she goes back to by typing
`git reset <commit_id> --hard`

Now she's back to this good commit. She can now run `git pull`, get Bob's changes and then apply hers to the scene.

33. This is one way to get around scene issues. Always make small atomic commits to scene files and don't bundle them with other changes.

Useful links:

<https://docs.unity3d.com/Manual/ExternalVersionControlSystemSupport.html>
<https://gatech.instructure.com/courses/105442/pages/version-control-with-unity>
<https://docs.unity3d.com/Manual/SmartMerge.html>
https://learngitbranching.js.org/?locale=en_US
<https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow>
<https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging>