Links:

Github: https://github.com/
Download Git (Windows and Mac): https://git-scm.com/downloads

Local vs Remote repository

Local repositories - stored on a user's computer and contain the complete version history of a project. Developers can make changes, create branches, and experiment with code without affecting the main project.

Remote repositories - hosted on a server (like GitHub, GitLab, or Bitbucket) and serve as a centralized hub for collaboration.

Developers can push their changes from their local repository to the remote repository to share their work with others. Remote repositories also facilitate team collaboration, allowing multiple developers to work on the same project simultaneously and merge their changes together.

Part 1: Create Account & Install/Setup Git

1. **Create an account on github.**
   - Use student or personal email. Does not matter which and you can change later if needed.
2. **Install git.**
   - **Windows:**
     - Download git from https://git-scm.com/downloads
     - Run the installer. Choose options according to your preferences.
     - Once installed, open “Git Bash” from the start menu.
   - **Mac:**
     - Download git from https://git-scm.com/downloads
     - There will be a few Mac options. Choose whichever you prefer but the easiest is likely the binary installer option.
     - Once installed, use the command `git --version` in the terminal to verify installation.
   - **Linux:**
     - Open a new terminal and enter: `sudo apt-get install git`
     - Use the command `git --version` to verify installation.
3. **Git credential setup:**
   - `git config --global user.name "Your_Username"`
   - `git config --global user.email "Your_email@example.com"`

4. **Set git to configure default branch as main (master is no longer the default on Github or Gitlab)**
    - `git config --global init.defaultBranch main`
5. **Git Authentication (three options):**
    1. **Git Credential Manager**
        a. **Access Token Creation:**
        - Type `git config credential.helper`. If you see "**manager**", you are all set.
        - Otherwise, install it.
            - **Windows:** If you kept the default options during the setup, it should already be installed.
            - **macOS**: `brew install --cask git-credential-manager`
        - Configure it: `git config --global credential.helper manager`
        - Now, you can authenticate just using your GitHub username and password (no SSH keys or personal access tokens required).
    2. **Access Tokens**
        a. **Access Token Creation:**
        - On the github website, click your profile photo in the top right corner.
        - Scroll down the left sidebar to "Developer settings".
        - Select "Personal Access Tokens" -> "Tokens (classic)".
        - Select "Generate New Token (Classic)".
        - Give the Token a note/name and select expiration date.
        - Select desired permissions (I recommend enabling the "Repo" section).
        - Click generate token.
        - Copy your token to a safe location (Github will never show you this token again.)
        - Should you need to change the token permissions, do so by selecting the token on the "Personal Access Tokens" page.
    3. **SSH Keys**
        a. **SSH Key Generation:**
        - ssh-keygen -t ed25519 -C "your_email@whatever.com"
        - ssh-add ~/.ssh/id_ed25519
        - Edit ~/.ssh/config
            - AddKeysToAgent yes
            - Host github.com
            - Hostname github.com
            - IdentityFile ~/.ssh/id_ed25519
        b. **Link SSH-key to Github Account:**
        - Open the file id_ed25519.pub and copy its contents
        - Go to github.com and click your profile photo in the upper-right corner and select Settings
        - In the Acces section select SSH and GPG Keys

- Click the button that says New SSH Key or Add SSH Key
- In the "Title" field, add a descriptive label for the new key. For example, if you're using a personal laptop, you might call this key "Personal laptop".
- Select the type of key, choose authentication
- In the "Key" field, paste your public key.
- Click Add SSH key.

Part 2: Create Repo, Initial Commit, First Push

Creating a Repository:

1. Once signed in, you can create a github repository on the website by clicking this button near the top right of the page:

2. In the dropdown menu, simply select “new repository”.
3. The next page will prompt you for some basic information:
    a. Repo name
    b. Description (optional)
    c. Whether you want the repo to be public or private
    d. Automatic creation of a readme file.
    e. Automatic creation of a gitignore file (with some common templates).
    f. License information (optional)
4. Once you are satisfied with the selections, click “create repository” and you will be redirected to your new repo page.

Cloning your Repository

1. Open a terminal in the folder you want to clone/download the newly created repository.
2. `git clone https://github.com/USERNAME/REPONAME`
    a. If your repo is private and git gives “repository not found” error, try using:
    `git clone https://USERNAME@github.com/USERNAME/REPONAME.git`
    b. Then use your access token to log in.

Committing changes to your repository

1. Add some files like scripts and assets to the cloned repository.
2. `git add .`
3. `git commit -m “COMMIT MESSAGE”`

Pushing commits to GitHub:

1. `git remote add origin https://github.com/USERNAME/REPONAME.git`
2. `git push -u origin main`

Part 3: Add Collaborators

To Add collaborators:

1. Open your repo's page on Github, then go to settings along the top of the page.
2. In the left side panel, go to access > collaborators.
3. Github may require some verification.
4. This page will show a list of all collaborators, pending invites for collaboration, and the "add people" button.
5. When adding a new collaborator, simply enter their Github username or email and select them when they appear on the page.
6. Lastly, the invited user will need to accept the invite through their email.

Part 4: More Commits, Adding Features

To add your work to the remote repository you need to stage your changes and commit them to your local repository. Then you will have to push your local repo to the remote repo.

Check the following link for a quick summary of basic rules about when to use commit.
Git commit best practices

- Commit related changes
- Commit often
- Don't commit half-finished work
- Write good commit messages

How to commit

Step 0
To do this you need to have your local repo set up and linked to the remote repo. This is covered in the last few sections of Part 2.

Step 1
Next work on your feature, add files, change files. Make the changes you want to see in the branch you are going to push to(which should be the main branch for this project).

Step 2
To stage changes you can stage individual files or entire folders.

Adding a file or list of files: `git add <file>`
Adding a directory: `git add .`
Adding files that have been altered: `git add -A`

ALWAYS RUN TESTS BEFORE COMMITING! DON'T COMMIT CODE THAT BREAKS STUFF

Step 3
You will need to commit these changes in order for them to be saved to your local repository. This is required for git to track changes in the code throughout the program's history.

```
git commit -m "Some descriptive name of the changes made"
```

This command will commit the changes made with a description of your choosing. The description should describe what you worked on so that when looking at the history of the project each change is self-descriptive.

Step 4
All the changes made so far have only affected the repository on your local machine. To sync the remote repository(Github or Gitlab) you need to perform a push

```
git push
```

A push operation sends all the changes in your repo to the remote repo so that everything is synced. It will, however, throw errors if the remote repo is ahead of yours. In this case, see the next section for merge conflicts.

Part 5: Pull Requests, Merge Conflicts, Branching

Merging Temporary Feature Branches Into Main
Good Workflow for **Local Development ONLY**.
In this class, we will not do any branching.

```
git checkout -b feature
# edit files
git add .
git commit -m "spam commit 1"
# edit more files
git add .
git commit -m "spam commit 2"
git checkout main # switch to main branch
# some changes occur on main
git add .
git commit -m "edit readme"
```

```
git checkout feature
# switch back to feature branch
# feature development completed
git add .
git commit -m "complete feature"
# merge changes into main
git checkout main
git merge feature # bring the changes into main
# delete the feature branch since all changes have been merged into main in a single commit
git branch -d feature
```

Part 6: Unreal Engine Specific Stuff

- Add contents of https://github.com/github/gitignore/blob/main/UnrealEngine.gitignore to your project gitignore.
- Setup Git LFS to track binary files in a better way.

Part 7: Git Best Practice Standards and Problem Solving

So with git, people (you) will run into problems. Therefore, to best avoid these somewhat inevitable problems, here are some coding standards that you all teams should integrate into their workflows.

Git Best Individual Practices

1. For every feature or change you make, fully test it and then do a full push of the change, no matter how minor. You should commit ***every time*** you make a change to the code
2. Keep your files up-to-date by running git pull and making sure your computer returns "Everything up-to-date"
3. Merge often **NEVER** rebase without your TL1's permission
4. Make smaller commits with good commit messages explaining what you did (1-2 sentences)
5. Do not rewrite history (causes very bad merge conflicts or file corruption)
6. Avoid merge conflicts, or know how to resolve them correctly
7. Follow the workflow standards everyone on your team has
8. Learn how to undo your changes to git (explained more down below)

Git Code Application Best Practices:

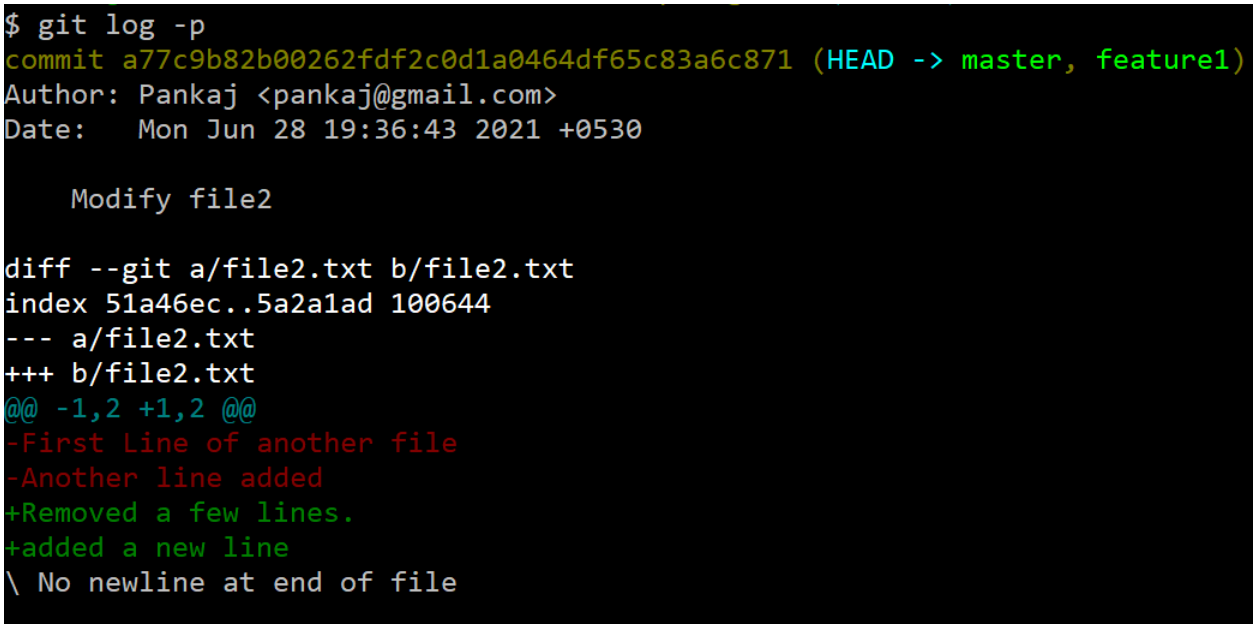
1. Before you begin coding or making in your workspace, run a git checkout and if code has been updated, perform a git pull
    a. Meaning: open up a terminal at your code's location and checkout the main branch and pull any changes
    git checkout main
    git status
    git pull
2. You should commit every time you are done in the workplace or with your code area.
    a. Meaning: After you save your code, run through a git push sequence:
    git add <path_to_file>
    git commit -m "<your_name> edits for <feature> 02/4/2024"
    git push

Undoing changes in Git

Say you committed a function that broke the code because you didn't test it, or you didn't pull and now the files are corrupted. Here are commands that will help you revert or change your code to previous versions.

To revert to previous git:

1. **git log** to see all of your previous commits



2. Copy the commit hash
3. **git reset <hash>**
    a. This command reverts it back to the commit you wanted it to be locally. Can use (--hard) if need be
4. **git commit -m "<Name> Reverting to <has> because <description>, <date>"**
5. **git reset --hard**
6. **git push --f**

THEN ALERT YOUR TEAM OR HAVE THE TL1 ALERT THEM so they know to pull the recent changes or save anything they have done since that commit because you ruined the codespace.

Git Merge

This is when you try to combine two or more development histories together. This is bound to happen, and especially if your team does not emphasize good git standards, it will be excruciatingly painful to get through. Not just painful. So, for your sake and ours. Here is what a merge conflict is, how to prevent them, and how to safely and cleanly work through them, and how to set up a mergetool.

Merge Conflict

This is when git can't automatically resolve differences between two commits. This can happen if code or other information is placed in the same areas, and git does not know which change to keep. So, here is how you will avoid these, as well as know how to peacefully get through them with the right code.

How to set up a mergetool and difftool for Git.

You want to use a visual mergetool. I prefer to use **Meld**, which is free, open-source, and can be used on Mac, windows, and linux systems.
Links to download based of your OS:
Linux: http://web.archive.org/web/20200512144500/https://linuxpitstop.com/install-meld-on-ubuntu-and-mint-linux/
Windows: The recommended version of Meld for Windows is the most recent release, available as an MSI from **https://meldmerge.org"**
Mac: https://yousseb.github.io/meld/

Configuring Meld:

1. After downloading try these commands in your system **LINUX/MAC:**
   git config --global diff.tool meld
   git config --global difftool.meld.path "/usr/bin/meld"
   git config --global difftool.prompt false
   git config --global merge.tool meld
   git config --global mergetool.meld.path "/usr/bin/meld"
   git config --global mergetool.prompt false

2. After downloading try these commands in your system **WINDOWS:**
   git config --global diff.tool meld
   git config --global difftool.meld.path "<path_to>\Meld.exe"
   git config --global difftool.prompt false

   git config --global merge.tool meld
   git config --global mergetool.meld.path "<path_to>\Meld.exe"
   git config --global mergetool.prompt false

IF THOSE COMMANDS DID NOT WORK:

3. Go into your .gitconfig file and add this:

Add the following to your .gitconfig file.
[merge] tool = meld [mergetool "meld"]
Choose one of these 2 lines (not both!) explained below. (either is fine not both but bolded line is recommended)
cmd = meld "$LOCAL" "$MERGED" "$REMOTE" --output "$MERGED"
cmd = meld "$LOCAL" "$BASE" "$REMOTE" --output "$MERGED"

   a. **FOR MAC USERS**
      i. **Note:** If you are on OSX and installed meld via homebrew, the output parameter will require an = like this: cmd = meld "$LOCAL" "$MERGED" "$REMOTE" --output="$MERGED" (stackoverflow)
   b. **ONLY FOR WINDOWS USERS**

c. Need to use full path to Meld following this link: https://stackoverflow.com/questions/14821358/git-mergetool-with-meld-on-windows

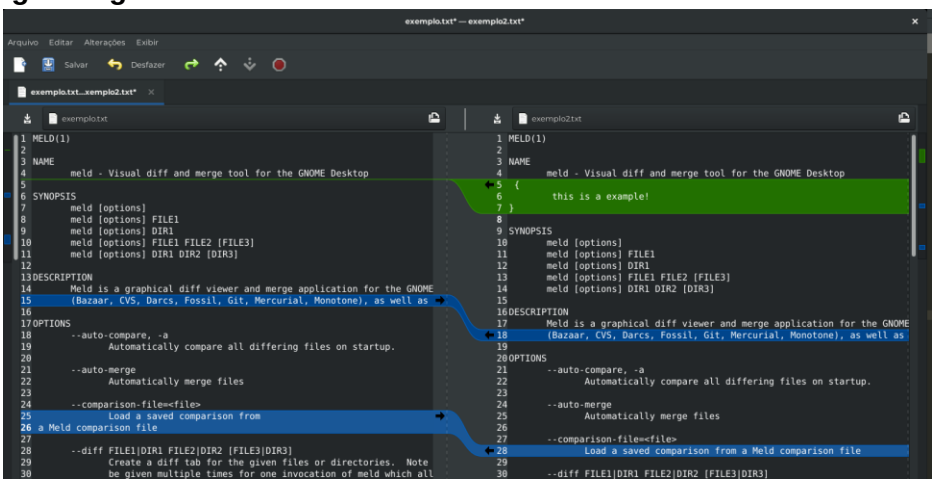
Using Meld:

So now whenever there is a merge conflict with git, here is how you will open and view the conflicts) **USING VS CODE**

1. Once you see that there was a merge conflict, you will respond accordingly
2. If it says has it been merged you answer the prompt with **N** for no

```
studytonight@ubuntu:~/awesome_project$ nano file3
studytonight@ubuntu:~/awesome_project$ git add file3
studytonight@ubuntu:~/awesome_project$ git commit -m "Edited File3"
[signin_feature 8fcb0fb] Edited File3
 1 file changed, 3 insertions(+)
studytonight@ubuntu:~/awesome_project$ git checkout master
Switched to branch 'master'
studytonight@ubuntu:~/awesome_project$ nano file3
studytonight@ubuntu:~/awesome_project$ git add file3
studytonight@ubuntu:~/awesome_project$ git commit -m "Edited File3 in master"
[master dfaa10e] Edited File3 in master
 1 file changed, 3 insertions(+)
studytonight@ubuntu:~/awesome_project$ nano file3
studytonight@ubuntu:~/awesome_project$ git merge signin_feature
Auto-merging file3
CONFLICT (content): Merge conflict in file3
Auto-merging README
CONFLICT (content): Merge conflict in README
Automatic merge failed; fix conflicts and then commit the result.
studytonight@ubuntu:~/awesome_project$
```

3. Then, you will run:
   git mergetool



4. After this, you will open the file in VS code and then you will see the conflicted files **ONLY AT THIS LOCATION**. **YOU WILL DO THIS FOR EACH CONFLICTED FILE.** Try and make them all the same, and carry over the new changes (which you can see by looking at the main history on github)

5. **HAVE FUN :)**