Download local ui @ https://www.gitkraken.com/

Git for Game Designers

A short guide for Game Design students to use Git as a version control system for managing games over their development cycle.

# What is Version Control?

Git is a form of version control (aka source control) software. It is used extensively in game & software development, or on projects that require collaboration and versioning.

There are other forms of source control, including Subversion (SVN) and Mercurial. Git is ideal for student projects as it is free, open-source, and does not require you to configure your own server.

## Why Use It?

Version control gives us several advantages over simply saving a project on your local hard drive, or even using a cloud solution such as Google Drive.

* A Git repository exists on a server *and* on each collaborator’s local system, it is decentralized.
* Git keeps track of every single change in your project, with comments.
* Collaborators can make changes, and synchronize to the master project.
* *Branches* can be created to test new features or to fix bugs.

# Git Fundamentals

## Repositories

A collection of files and folders under one specific folder that you set to be the main folder in Git is called a *repository* (aka *repo)*.

Since the repository is essentially just a folder that is watched over by Git, you can *clone* the repository and have multiple, working copies of it. You have to do this through Git though, so that Git knows to watch over that clone of the repository as well.

A clone of the repository which is hosted on a server is called a *remote repo*. A clone of the repository which is hosted on your own computer is called a *local repo*. Git is decentralized, which means that you have a full copy of the repository whenever you clone a repo. This means that you do not need to be connected to the repository you cloned from to rollback changes on your cloned repository.

## Committing

As you work on the project, you will *commit* changes to your local repository, which creates a timestamp of your project as it was when you committed it. It is good to think of it as keeping a timeline of your project, where the first commit is your project as it was when you created the Git repository (most of the time an empty folder) and your last commit is the finished project, with all the commits you made along the way in between in chronological order.

At any time, you can revert your local repo backwards or forwards to any commit you ever made on the project. Accidentally delete a file you thought was unimportant? You can even roll back changes made to individual files, not just the entire project.

Each commit should also have a message to note what has changed in this commit. This should be descriptive, as it will become part of the repo’s history and will be helpful later when you need to know which commit to need to revert back to if you made a mistake.

## Pushing

When you are ready for other people to get your changes, you will *push* to the remote repo. This adds your local commits to the remote repo for other collaborators to access. Think of this as uploading your commits to the remote.

## Pulling

You can update your local version of the repo by *pulling* from the remote, which will add all the changes that have been committed and pushed to the remote by collaborators to your local repo.

## Branches

A *branch* is an alternate timeline in a repo, with its own set of commits. Branches are often created to test new features, fix bugs, or keep track of different versions of a repo (for instance a stable release and a development branch).

The main timeline of a repo is called *master*. A new branch is created by *forking*. A branch can be *merged* back into the master or another branch.

Warning: problems can occur when multiple collaborators try to push or merge the same file, notably a scene in Unity. We have developed a set of guidelines to minimize the issues.

It is often considered best practice to work locally on a *feature branch*. When your feature is working correctly, you then then push this branch to the remote, and merge with master.

## Pull Request / Merging

A branch can be *merged* back into the master or another branch. In GitHub, this is done through a *Pull Request.*

## Git Ignore

All Git repositories will contain a file called *.gitignore*, which tells the repo to ignore unnecessary files & folders generated by the software you're using to develop your game.

When creating a new repo, GitHub gives you the option to create a .gitignore based on the specific type of project. Be sure to choose the correct .gitignore type (Unity, Unreal, etc.).

# Getting Started with Git

Just as there are many ways to create a game, there are many ways to use source control. An easy way to get started with git is by using [GitHub](http://github.com) and [GitKraken](https://www.gitkraken.com), a *graphical user interface* (GUI). There are other GUIs including GitHub Desktop, and a git *command line interface* (CLI).

1. Each collaborator needs a GitHub account. We recommend a free [GitHub Student](https://education.github.com) account, which includes some additional bonuses.
2. Install [GitKraken](https://www.gitkraken.com) and/or the Git CLI (both are already installed in the FSU game labs)
3. One collaborator should set up the initial repository as follows:
   1. Create a new private or public repository on [GitHub](http://github.com) with a short and descriptive name. Set the .gitignore to the correct project type (Unity, Unreal, etc.). Note the URL of the repo.
   2. If the repo is private, add collaborators (they need to have GitHub accounts).
   3. *Clone* the repository to the local system. (In GitKraken, Clone Repo [Ctrl-N]. Browse to set the location (local) and enter the url. The local repo will be a folder in the browsed location.)
   4. Add project-specific folders to the repo. For Unity, this would be the Assets and ProjectSettings folders.
   5. *Commit* changes, adding a descriptive message. (In GitKraken, click the dotted circle at the top of the middle window, stage all files, add a message and commit.)
   6. *push* it to the remote. (In GitKraken, click the Push button)
4. Each collaborator will Clone Repo, as described above.

## Git Workflow

* pull from *master* on remote repository
* make a new *feature branch* in local repository
* make a new scene or reuse older test scene (*only to be used by you!*)
* do all work in that scene
* prefab all objects made
* (optional) export prefabs as Unity assets, store in a different folder (gdrive?)
* commit changes, with descriptive comments
* push and merge!

(thanks to Nathan Wentworth)

## Links

* [GitHub](https://github.com) / [GitHub Student Pack](https://education.github.com)
* [GitKraken](https://www.gitkraken.com)
* Tutorial: [Learn Git | Codeacademy](https://www.codecademy.com/learn/learn-git)
* Article: [Git and GitHub for Game Developers](https://gamedevelopment.tutsplus.com/tutorials/git-and-github-for-game-developers--gamedev-11507)
* Article: [Working with Git for Games (and in general)](https://www.gamasutra.com/blogs/TrentPolack/20170728/302667/Working_with_Git_for_Games_and_inGeneral.php)