

How to Git Gud

(or good enough for games project)

Authors: Sunny Miglani, Ibrahim Qasim,
Nathan Doorly, Joe Stephenson

What is Git?

It's a “version control system for code”.

More importantly, it's NOT

- Google Drive
- Messenger
- Whatsapp
- Emailing
- USB devices
- Floppy disks
- Punch cards

Git's great for keeping track of changes in your code, and making it super easy to rewind back to when your code actually ran.

Some basic commands

Starting a Repository:

`**git init**` : Creates a git tracked repository in your current folder. Here is where you will track files

`**git clone <https/ssh>**` : Takes an http link or ssh link and copies the repository from the cloud onto your computer.

Remotes in git

A `remote` is basically the link to the server you're using as the “cloud” for your git.

Usually these are websites like Github, or BitBucket and everytime you push or pull, it access these repositories.

Useful commands:

`**git remote add origin <link>**` ← Usually used right after `git init` to link to your cloud

`**git remote remove origin**` ← This is useful to remove the remote from a repository you clone!

Tracking Files

This basically means git is stalking your files, and looks for any changes and keeps track of it. It looks at the current saved version, and the new version and looks at the difference.

It “saves” a version only when you `commit` to the code.

“Hey, start looking at this file, don’t save it until I commit though!”

Commands:

`git add <filename>` adds a particular file to the tracking (works for directories as well)

`git add --all` takes all possible files which aren’t skipped by the `gitignore`

Tracking Files

“**.gitignore**”: It’s a ‘ hidden file’ that keeps track of files and folders that git shouldn’t track.

“Hey, ignore this folder or file because I don’t want people to see it!”

This is a good place to put things like ssh keys, passwords, raw data that shouldn’t be uploaded. It also lets you ignore things like temp folders, build files and apks.

Tracking Files

“**.gitignore**”: It’s a ‘ hidden file’ that keeps track of files and folders that git shouldn’t track.

“Hey, ignore this folder or file because I don’t want people to see it!”

This is a good place to put things like ssh keys, passwords, raw data that shouldn’t be uploaded. It also lets you ignore things like temp folders, build files and apks.

There’s a lot of **premade git ignore** files that you can find online.

(Hint Hint: Look up Unity/Unreal gitignore files!)

This is also useful if you have Java or C projects, since they create a lot of heavy builds that you don’t need backups of.

Being in a committed relationship with code

Commit's are how you “save” in git.

When you commit something, its the same as you telling git

“Hey, Look at what this file looks like, and save it as a ‘version’.”

Usually you give it a message so you know where you were

```
`git commit -a -m <or -am> <message>`
```



Pushing and Pulling: How to pull in mvb

``git push``: You do this after you've committed, this is how you push your information to github!

Pushing and Pulling: How to pull in mvb

``git push``: You do this after you've committed, this is how you push your information to github!

STOP

HAVE YOU FETCHED & PULLED?

Pushing and Pulling: How to pull in mvb

``git push origin <branchName>``: You do this after you've committed, this is how you push your information to github!

Before you push, you MUST fetch and pull from git

``git fetch origin <branchName>``: This looks at your remote, and checks if the remote is ahead of your current repository's status! It's extremely useful if you want to avoid **merge conflicts**.

Pushing and Pulling: How to pull in mvb

``git push origin <branchName>``: You do this after you've committed, this is how you push your information to github!

Before you push, you MUST fetch and pull from git

``git fetch origin <branchName>``: This looks at your remote, and checks if the remote is ahead of your current repository's status! It's extremely useful if you want to avoid **merge conflicts**.

``git pull origin <branchName>``: This pulls from your remote, and compares and *merges* your files to ensure there's not conflicts on pushing. (I'll cover merge conflicts soon).

If you have NO merge conflicts, it's now safe for you to push to the current branch!

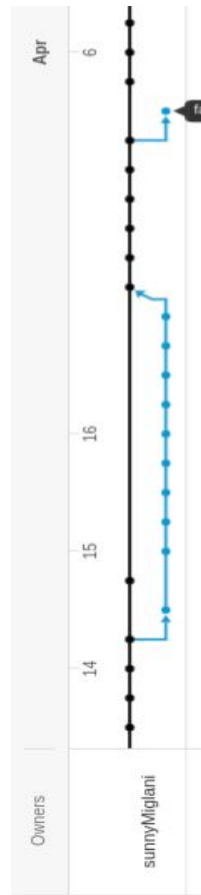
Branches!

Branches are different “timelines” of your code.

Branches can work *concurrently*, and you merge them together when both versions are ready.

It’s also great for solo projects, when you want to maintain different versions of the same code.

It’s useful as it allows you to test out different versions of the same code, and merge only the working version while still keeping a “stable” build/branch



Creating a Branch

``git branch <branchName>``: This will create a new branch from the current state of your program.

Creating a Branch

`**git branch <branchName>**` : This will create a new branch from the current state of your program.

`**git checkout <branchName>**` : This will switch you to the branch. (Make sure you commit your data or you might lose it)

Creating a Branch

`**git branch <branchName>**` : This will create a new branch from the current state of your program.

`**git checkout <branchName>**` : This will switch you to the branch. (Make sure you commit your data or you might lose it)

`**git checkout -b <branchName>**` : This creates a new branch, and switches you and your code to be in that branch.

Creating a Branch

`**git branch <branchName>**` : This will create a new branch from the current state of your program.

`**git checkout <branchName>**` : This will switch you to the branch. (Make sure you commit your data or you might lose it)

`**git checkout -b <branchName>**` : This creates a new branch, and switches you and your code to be in that branch.

`**git branch {-d/-D} <branchName>**` : The -d flag will **delete** the branch if it's merged in, and -D will delete it even if it hasn't been merged yet. Be careful when using -D

Merging Branches!

Merging is how you combine two different “timelines”.

``git merge <branchName>`` : Be really careful with this, you pull the branch into your current branch

For Unity: Always merge master / dev into your branch, to make sure it works with the “current working version” and TEST AS IF IT’S GAMES DAY

Use ``:q`` to quit vim and ``Cntrl X`` to leave Nano.

MERGE CONFLICTS

MERGE CONFLICTS EVERYWHERE

memegenerator.net

Merge Conflicts

These happen when two people code on the “same lines”, and git isn’t sure which is the needed thing.

<<< HEAD is your current branch

==== Space where your current stops, and the branch begins

>>>> branch-a is the merging branch.

If you have questions, please

<<<<<<< HEAD

open an issue

=====

ask your question in IRC.

>>>>>>> branch-a

AND NOW

WHY GIT IS ACTUALLY USEFUL

GIT RESET --HARD

HOW TO ACTUALLY RESET

``git reset --hard``: This resets ALL your work, to the last commit on your branch

HOW TO ACTUALLY RESET

`git reset --hard`: This resets ALL your work, to the last commit on your branch

`git reset <commitID>`: This resets it to a specific commit in the branch, **be careful with pushing after this**

HOW TO ACTUALLY RESET

`**git reset --hard**` : This resets ALL your work, to the last commit on your branch

`**git reset <commitID>**` : This resets it to a specific commit in the branch, **be careful with pushing after this**

`**git log**` : Returns the log of your last commits on the branch, use this to see your commitID (use q to quit the interactive mode)

HOW TO ACTUALLY RESET

`**git reset --hard**` : This resets ALL your work, to the last commit on your branch

`**git reset <commitID>**` : This resets it to a specific commit in the branch, **be careful with pushing after this**

`**git log**` : Returns the log of your last commits on the branch, use this to see your commitID (use q to quit the interactive mode)

`**git checkout -b <branchName> <commitID>**` : You can even branch at a particular commit! **Again, careful with how you push and merge now.**

**ASK ME QUESTIONS
NOW SO IT'S NOT SILENT**

UNITY AND GIT

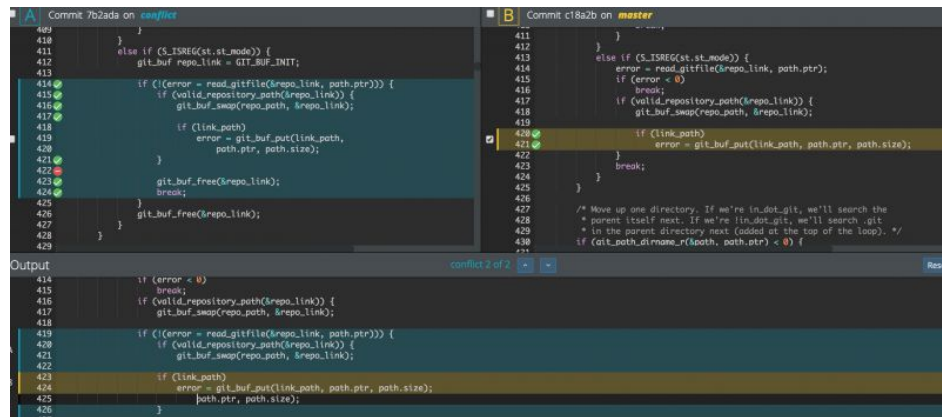


How to make Git your friend



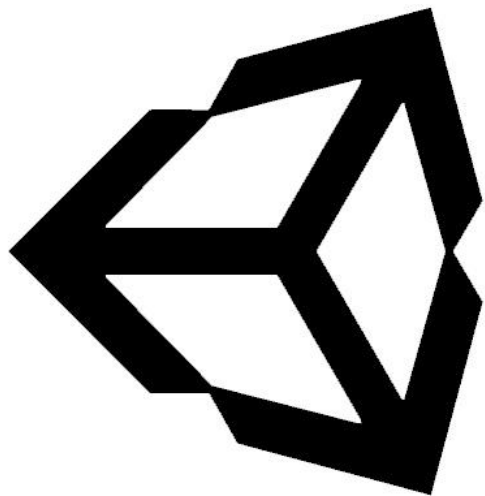
How to make Git your friend

- Great UI to help visualise various branches
- Good Interactive screen for merge conflicts
- **Great for your teammates who didn't attend this and therefore know nothing about git**



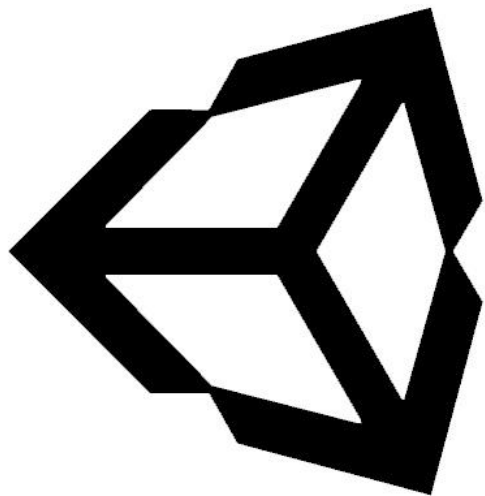
```
400 }
401 else if ($?TSRG(st.st_mode)) {
402     git_buf repo_link = GIT_BUF_INIT;
403
404     if (!(error = read_gitfile(&repo_link, path_ptr))) {
405         if (valid_repository_path(&repo_link)) {
406             git_buf_swap(repo_path, &repo_link);
407
408             if (link_path)
409                 error = git_buf_put(link_path,
410                                     path_ptr, path.size);
411
412             git_buf_free(&repo_link);
413             break;
414         }
415     }
416     git_buf_free(&repo_link);
417 }
418 }
419
420 /* Move up one directory. If we're in .dot_git, we'll search the
421  * parent itself next. If we're in .dot_git, we'll search .git
422  * in the parent directory next (added at the top of the loop). */
423 if (git_path_gitname_r(&path, path_ptr) < 0) {
```

How to make Git your friend



SMART MERGE

How to make Git your friend



SMART MERGE

**THIS IS REALLY IFFY, BUT IF
IT WORKS IT'S ABSOLUTELY
BEAUTIFUL**

Teams and Git (I didn't make slides)

1. Protected Branches
2. Private Repositories
3. Git Issues and Pull Requests
4. Accepting and Reviewing Pull Requests
5. Code Reviews (and Branches)
6. Branch Names