Version Control: Git Crash Course

January 17, 2022

Paul Bonsma

Version Control

- Version control is *useful* when working alone (inspecting changes, inspecting older versions, reverting changes, keeping a log, backups, working on different machines…)
- You basically *need* version control when working with a team
- Recommended tool: *git*
- To install:
    - Go to https://git-scm.com/downloads and download the proper version
    - Make sure you include the *terminal (git bash)* in the installation!
    - It's okay to choose all the default options in the installer

- Optionally, you can also install a *git client,* like *Sourcetree, GitKraken, Fork*

Git Concepts: Repositories

- A *repository* keeps track of a sequence of changes to / versions of a project, called *commits*
- The *remote repository* contains the official version of the project
- Your *local repository* is a working copy
- *Pull*: get the latest commits from remote, add to (merge with) local
- *Push*: add your local commits to remote
- Though git is smart, sometimes local commits clash with remote commits, and you need to manually decide how to *merge* changes
- You need to do this between pull & push!

Git Concepts: Making Commits

- You make changes to your files locally
- After finishing your changes, testing them thoroughly, and cleaning up your code, you're ready to commit them
- First: *stage* all the changes (files) that should be part of the commit using the *git add* command
- Next: *commit* your staged changes using the *git commit* command
- Add a clear, descriptive *commit message*!
- At that point, your changes are part of your local repository, and you may consider pushing them to remote.

Git: the Beginner Experience

- *Team*: “git is a *magical* tool that allows easy backups and team work!” :-D
- **Team proceeds to…**:
  - *Commit tons of random crap*
  - *Create many random branches*
  - *Not communicate who works on which files*
  - *Push, Pull, and merge uncontrollably & randomly*
- *Repository:* **goes up in flames**
- *Team*: “git is a headache!” >:-(

→ *Don’t blame the tool*. Just like anything on your computer, git just does what you tell it to do. Your team needs some *discipline* and *rules*

Git Rules: Push, Pull, Merge

- When getting ready to push your local commits: *Pull first, fix merge conflicts locally, then push*
- Between your pull and push, no other team members may push! (If you’re sitting at the same table: use a physical “push token”: only the person holding that may push ☺)
- To avoid merge conflicts:
  - Try to avoid working on the same files simultaneously.
  - Push regularly (though never push buggy code!)
  - So: *Communicate!*
  - *Good code architecture* helps a lot! (small decoupled classes, data vs code)

Git Rules: Commits

- Only commit *working, tested code* (NOT: "I'm done for the day, I'll fix the bugs tomorrow, but let's make a backup for now")
- Make short, *atomic commits*, that focus on one aspect (NOT: "This commit changes the player controls, level loading, adds some sound effects, and changes the HUD layout")
- Never commit temporary files or temporary changes (NOT: "I optimized level loading, but now the game always starts with my test level")
- Have clear commit messages (NOT: "Hope this works!")

Though I admit: looking at commit messages in student repositories is often entertaining ☺

Git Tips: Commits

- *You do not need to include (stage) every modified file in your commit*: if you fixed Level.cs, but MyGame.cs still includes some test code, then maybe you should only stage Level.cs! (*git add Level.cs*)
- Before starting a commit, inspect all your changes using *git diff* (typically: you see that you need to clean up some code / remove some temporary Console.WriteLines)
- Your .*gitignore* file helps to avoid committing random temporary files. Nevertheless, it's still *your responsibility* to verify that no files slipped through the cracks: use *git status* (inbetween *git add* and *git commit*)

When it (Inevitably) Goes Wrong

- Use *git log* to get an overview of recent commits
- Use *git checkout [filename]* to undo uncommited local changes
- Use *git checkout [commithash]* to *view* earlier versions (note: “detached head” state → better not make changes from here)
- Use *git diff [commithash]* to inspect recent changes (where did it go wrong?)
- Use *git revert [commithash]* to undo a commit (not necessarily the last one)
- Merge conflicts: fix them in your IDE (choose which lines to keep), then mark them as fixed using *git add [filename]*

Adding Assets to the Repository

- Bin/Debug contains many temporary files, that should not be in the repository (.exe, .pdb, (placeholder) assets, settings.txt…)
- The given .gitignore file takes care of that
- If you want to add your assets to the repository:
  - Use the *"asset folder + build action" workflow* mentioned in GP lecture 6
  - Don't put temporary (placeholder) files in your repository
  - Be careful with huge assets and changing them often!
  - If necessary, look into git LFS (large file system)
  - Remember that git is not really geared towards binary assets

Git Cheat Sheet: Basics

Command:	Use:
git add	Stage files (fix merge conflicts)
git commit	Commit staged changes
git status	Get status (staged files, local vs remote)
git log	Overview of recent commits
git diff	View difference between commits / uncommitted changes
git checkout	Undo uncommitted changes / look at previous version
git revert	Undo a recent commit
git pull	Get changes from remote
git push	Add local changes to remote
git clone	Clone a remote repository (create local copy)

Next Steps

- Working with branches
- Working *properly* with branches! (example: *git flow* convention)
- More advanced features (stash, cherry pick, rebase, pull requests, …)

- Extremely powerful + extensive tool, and information can be overwhelming
- Nevertheless, you can start simple. If you do it *carefully*, you actually don't need many of the advanced tools (for fixing problems)

- Note: working with git + GXPEngine is a lot easier than working with git + Unity!

→ *So better start learning and practicing now!*

Resources and Tutorials

- This was a quick crash course / introduction, also assuming some prior knowledge
- Focus on workflow / rules / guidelines / concepts, less on the commands
- (If you use a *git client*, the commands/actions are different anyway – though the underlying concepts are the same!)
- So you probably need more resources to learn git
- Suggestions:
  - https://rogerdudler.github.io/git-guide/
  - https://git-scm.com/docs/gittutorial
  - https://www.cprime.com/resources/blog/the-7-best-git-tutorials-to-get-you-started-quickly/
  - https://www.atlassian.com/git/tutorials/atlassian-git-cheatsheet
  - Bonus: https://girliemac.com/blog/2017/12/26/git-purr/