Basic workflow creating a Pull Request In order to get started with your submission, you should do something like this:

```
$ git checkout dev # switch to dev branch
$ git pull upstream dev # pull recent version
$ git branch MyFeature # create a branch for your PR
$ git checkout MyFeature # switch your feature branch
```

Now you can edit.

Don't use your `dev` branch for PRs, even if it only contains minor changes. It is possible, but complicated, so don't! I did this a couple of times and I always ended up regretting it.

```
$ git status
```

will show your changes.

In order to add your changes to your branch do

```
$ git add MyChangedFile MyAllNewFile dirWithChanges/ newDir/
$ git commit
```

. You can add as many commits as you like.

Once you are done, push the branch to your fork at Github

```
$ git push origin MyFeature
```

Visiting the three.js project, it will offer you to create a pull request. Make sure to select the `dev` branch for its base - **not** `master`.

When your feature has been merged into `dev` you can delete the branch:

```
$ git checkout dev # switch back to dev
$ git branch -D MyFeature
```

Note that the commits are being combined into one ("squashed" in Git slang) when your PR is merged and therefore Git won't recognize whether your branch was merged or not. This is why you have to use `-D` (forced delete) rather than `-d` (normal delete).

In order to remove a remote branch

```
$ git push origin --delete MyFeature
```

(you can also press a button on Github's web interface).

I accidentally staged (== `added`) a file but don't want to commit it. How to get rid of it?

```
$ git reset HEAD
```

unstages everything you added for the upcoming commit. It does not change your files. After running it you can re-add the changes for your commit.

How do I remove my (not yet `commited`) edits from a file?

```
$ git checkout MyChangedFile
```

resets the file back to its checked-in state. It also works with directories.

Github says my branch is conflicting and cannot be merged, what do?
A common reaction is to `merge upstream dev`, but this is not really so good of an idea, because it buries your changes underneath the merge.

Instead of merging the changes in `dev` into your feature branch, you can often just rewrite your feature's history on top of current `dev`.

Git provides an automation for this purpose:

```
$ git checkout dev # to my dev branch
$ git pull upstream dev # pull changes
$ git checkout MyFeature # back to my feature branch
$ git rebase dev # let me rewrite it
```

Git will guide you through the process of resolving conflicts at this point, and it's very similar to merging, except you're in fact rewriting your history, so your files will look like they did at the point of the conflicting commit.

Once your branch is rebased, test that everything is still working. Sometimes you may want to test intermediate steps while rebasing.

When everything is fine

```
$ git push --force MyFeature
```

to rewrite your PR.

Note: In case you have multiple dependent PRs (that is, one includes changes of the other), you may have to use `git rebase --interactive` and remove the commits that were already merged. The commits in the PRs are combined into one, and thus Git can't match them automatically.

I'm new to Git and scared to try new things because I may screw up!
It's important to realize that every commit refers to a complete and immutable state of the entire source tree. Therefore, as long as you remember a commit's ID you can always go back to that state.

One way to remember it is by creating a temporary backup branch:

```
$ git branch MyBackup
$ # e.g. to do your first rebase now
```

Now when your branch was messed up you can undo whatever you did with

```
$ git reset --hard MyBackup
```

. The `reset` command makes the current branch point to a specific commit and there are various ways of specifying it (using another branch is one, you can also write the output of `git log` to a text file and use the IDs directly, for instance). The `--force` option tells Git to not just reset the current branch but also your files, so you have to be careful when there are changes that are not yet checked-in (see `git status`).

How do I "take the builds out"? I assume that you followed the advice given above and did not bury your changes under a merge. In case you did, you may want to resort to more extreme measures and read the next section instead.

If you are feeling insecure, see the above section to know how to undo whatever can happen.

First, make sure everything is checked-in. Then run

```
$ git log
```

and identify the commit your branch is based on. Also remember the current HEAD of your branch (the topmost ID in the log). It could be useful to just write the log into a file:

```
$ git log >log_file.txt
```

Now switch to the state before your commits

```
$ git reset --hard <ID of base commit>
```

and copy `build/three.js` and `build/three.min.js`.

Then switch back to the old HEAD:

```
$ git reset --hard <ID of last commit>
```

Next, start an in-place rewrite of your history:

```
$ git rebase --interactive <ID of base commit>
```

an editor will open. Now change `pick` to `edit` for the commits that include the builds.

The rebase will stop at every commit you selected for editing. Every time it does, copy the builds from the base commit back to your working copy. Then

```
$ git add build/
$ git commit --amend
$ git rebase --continue
```

and on to the next. Use

```
$ git push --force origin MyFeature
```

once you have successfully rebased your branch.

Welcome to the time machine - rewriting history from scratch Upfront a word of caution: You do not want to abuse this technique in the midst of a review. Make sure to keep things transparent for the reviewer in some way. In particular do not restructure your changes in a completely different way just because you can - it may cause confusion.

There can be cases, however, where you want to give your feature branch a fresh history, e.g. after burying builds underneath a merge or when there are too many conflicting changes for rebase to be practical.

Here is how it's done:

```
$ git checkout dev # switch to dev branch
$ git pull upstream dev # update it
$ git checkout MyFeature # switch back to the feature branch
$ git merge dev # merge with dev
$ # ... resolve conflicts here
$ git commit
$ # you can take backup measures here, see above
$ git reset dev # do *not* use --force, keep your files!
$ # repeat the following steps to rebuild your history
$ git add <file(s)>
$ git commit
```

If you have unrelated changes in a single file, you can use

```
$ git add --interactive <file(s)>
```

and then use the **p**atched add option - to add these changes to different commits.

Finally

```
$ git push --force origin MyFeature
```

to Github.

Wait, does git never forget anything? Run

```
$ git gc
```

to invoke the garbage collector. It will remove commits that are no longer referenced by any of your branches and older than 30 days (you can change this default - see the manual for details).

How do I keep Git from wear-leveling my SSDs? Use a ramdisk.

The Linux kernel supports RAM-based the file systems `tmpfs` / `shmfs`, the HFS file system (Mac OS X) supports RAM-based images, and for Windows there is a (free & non-proprietary) RAM-based disk driver called ImDisk.