CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

FOMICS SummerSchool July 2013
Git

John Biddiscombe

Introduction

Git is a distributed version control system
Distributed means that many people can have local copies of the repository and it's easy to synchronize between them.

A collection of command-line tools
Though GUI tools are available.

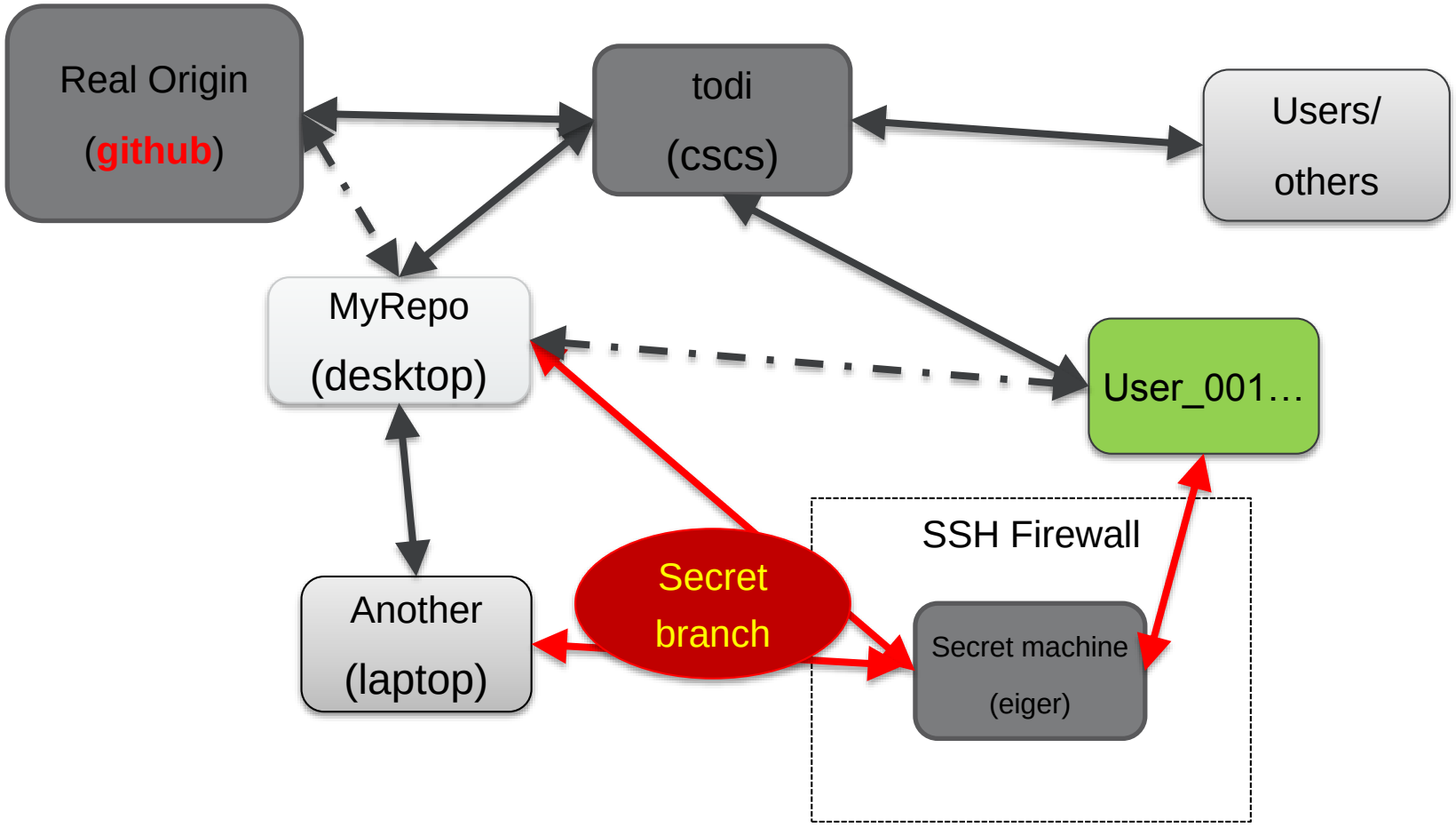
Has lots of commands
And they can be piped/scripted nicely, so it's more like a toolbox of utilities than a single program. Think of it as an environment in which you can manage source code.

Branches and branch management
Are marvellously implemented and conceptually sound.

Commits are atomic
Each commit represents a changeset (and maintains tree history)

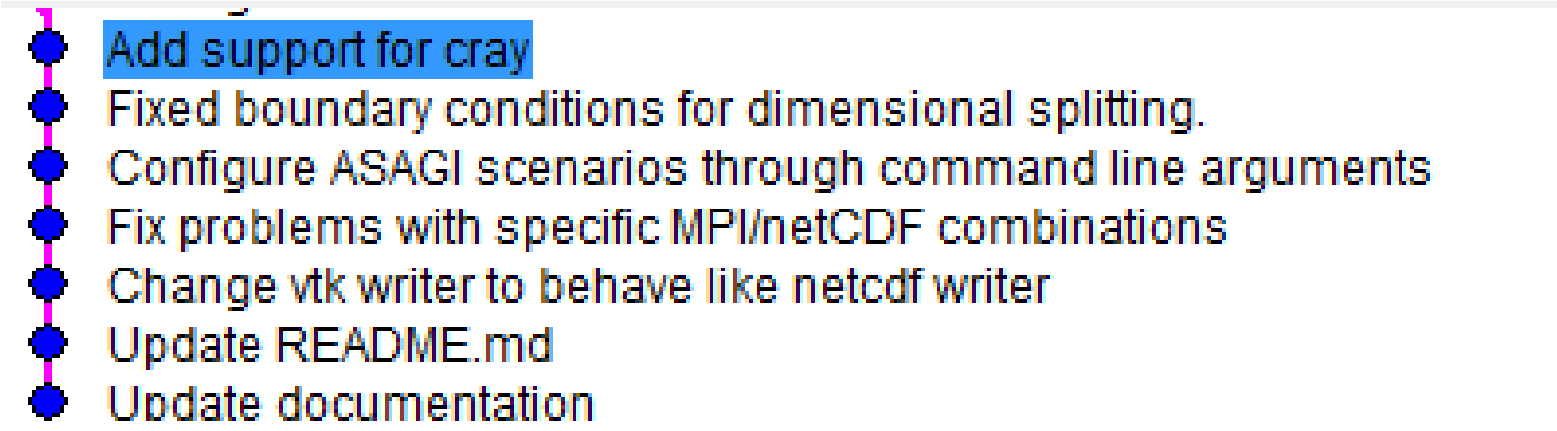
Too many commands/concepts to cover in this short intro.

Distributed



How does it work

It is straightforward
Edit some source code
Add the files you've changed (to the next commit)
You can now add some more if you want (or remove some)
Commit the files



Here you can see the effect of a number of straightforward commits

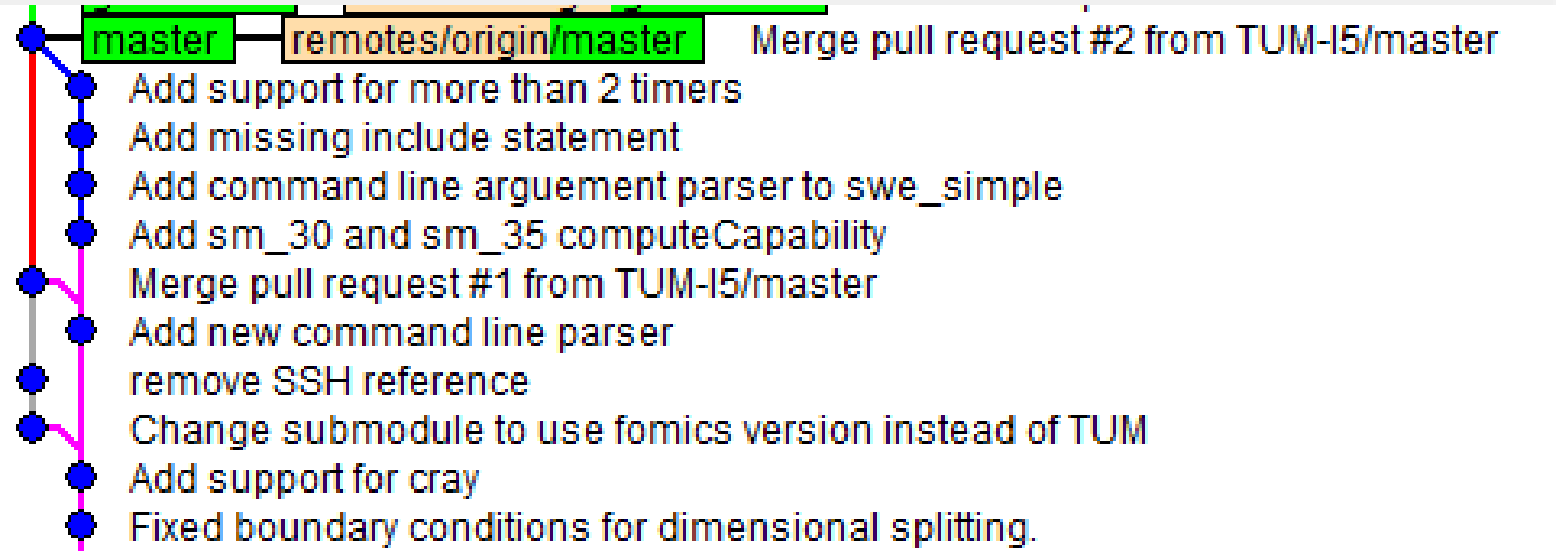
Commits

What is a commit

Each commit represents a new state of the repository

Each commit has a parent

But when you merge branches you create a merge commit which has two parents (or more if you do an octopus merge!)



Here we see development on a branch has been merged in. And then continued and merged in again. The merge commits have two parents.

What is a branch

A branch is just a commit that has a special name
Except that when you commit to a ‘named’ branch, the pointer to the branch commit is updated to point to the new commit

Every commit can be a branch if you want

You can create a branch any time, from any commit

You can go back to an earlier commit and make a branch from there

A branch that has no name is referred to as a “Detached Head”, you’ll get onto one of those from time to time.

When do you want a branch

Nearly Always!

Assuming the main development is in branch ‘master’, then anything you do should probably be in branch ‘mystuff’, or topic_fixfeature or feature_algorithm, or something of that kind.

You can create and delete branches very easily, so when in doubt, just create a branch with a good name and commit to it.

Sometimes you’ll fix two bugs and realize that they ought to be on two branches rather than the same one, so you can quickly create temp branch, commit two patches, then later copy those patches into the branches you want them to be in (cherry-pick – see later)

Branch workflow

Common practice is to have branches like

Master

Where new topics are merged into and where you branch from to create a new topic, new development will assume this is the starting point

Next

What will eventually become the next release version (candidate). Topics can be merged into here

Release

When the next branch is ready for release, you merge into the release branch

Tags or branches can be used to mark individual releases

Lots of great stuff about workflows here

http://www.vtk.org/Wiki/Git/Workflow/Topic

Tags

Are just names to commits that you want to find easily

Same as branches, except they don't update themselves the way a branch does when you add to it.

You can point them anywhere and update them like these

v3.98.1
v3.98.1-RC1
v3.98.1-RC2
v4.0.0
v4.0.0-RC1
v4.0.0-RC2
v4.0.0-RC3
v4.0.1

When you need to patch an old release, just create a branch from that tag, add commits and re tag when done

~ and ^

^ = parent
~ = ancestor

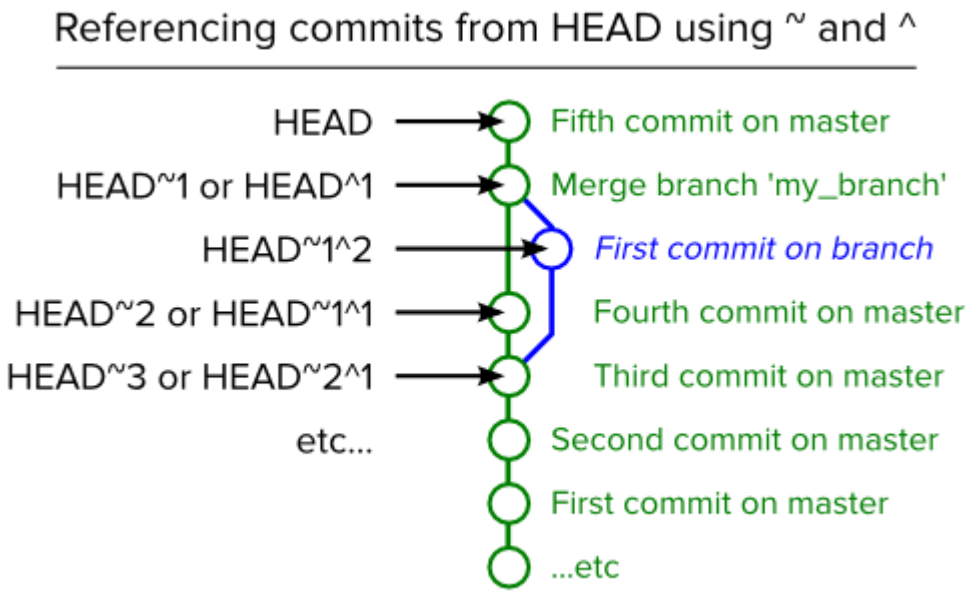


Image credit: Paul Boxley, Blog

so we can use
git diff HEAD HEAD~4

but parent ^2 is only
useful if a commit has two parents

I use this kind of thing often
git diff --name-only HEAD~x

Remotes

When you clone a repo, you have a remote called 'origin'

You can add as many other remotes as you like and get copies of branches from all of them.

When you want to see if something new has been added to a remote, you 'fetch' changes from the remote. This updates your copies of the remote branches, but does not change any of your local branches.

Nothing changes on the remote until you push your local branch to the remote.

What's the difference between a local branch and a copy of a remote branch?

Merging

You will usually merge your work into a (master/next) branch - if other people want to get your stuff synced with theirs.

Or merge master into your work branch - because someone else has added to the master branch and you want it too. You do this by doing a pull from the master. (pull=fetch+merge)

From time to time you'll want to merge friend's work branch into your work branch – this might be because you're both working on something related and before it's ready to go into master, you want to sync with each other.

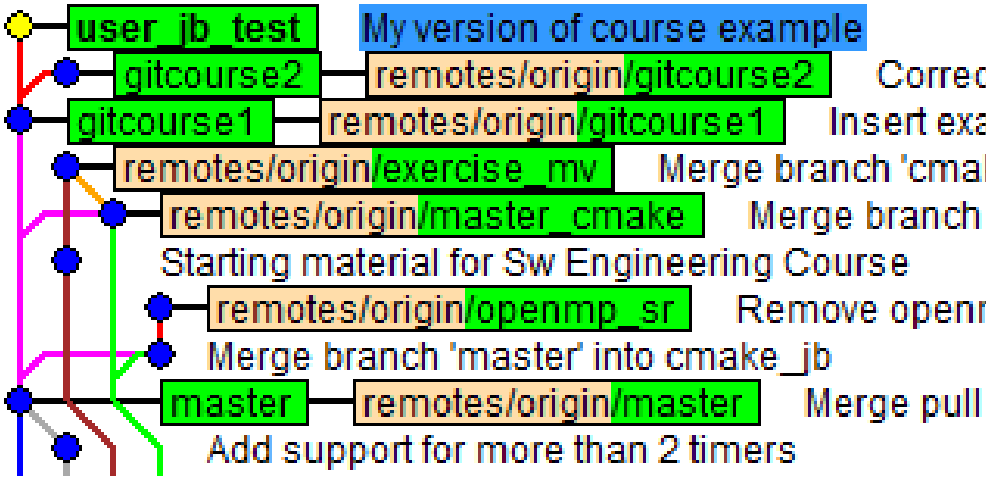
10962_find_data_initialization
10968_convert_query_seln
10968_freezier_frozen_selection
12742_scalar_bar_improvements
13024-debug-leaks-view

Some project use a branch for each bug fix, and you want one.

Local branches

After you clone a repo, you may have many branches

remotes/origin/augrie_geoclaw
remotes/origin/exercise_mv
remotes/origin/gitcourse1
remotes/origin/gitcourse2
remotes/origin/master
remotes/origin/master_cmake



They are just points on a tree

When you checkout one of those branches, you create a local branch which is initially at the same point on the tree.

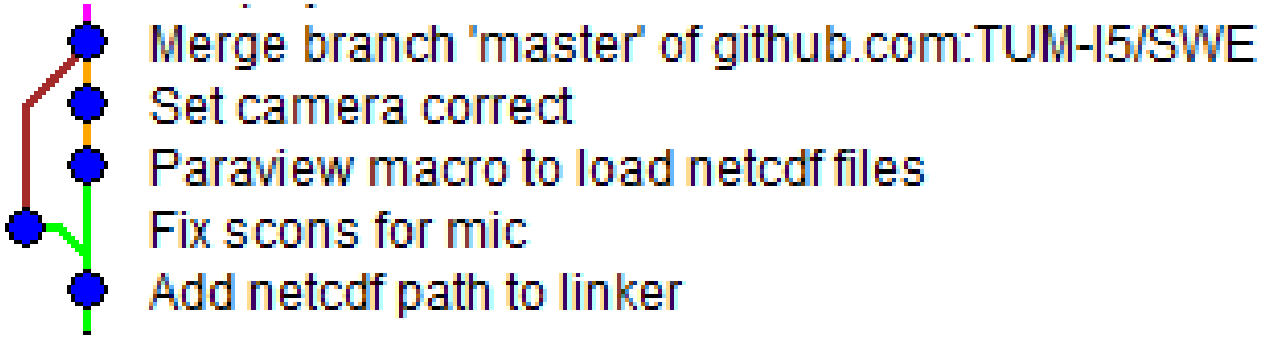
As you commit to your local branch, it diverges from the remote copy you made. When you push your local branch to a remote, the remote is updated and your copy of the remote is updated.

Merges and Conflicts

When you push your local branch to a remote, you may be refused.

If someone else has already pushed to the same remote branch, then the stuff you want to add on top of it is not a "Fast Forward", so git will say, sorry, but your branch is not up to date.

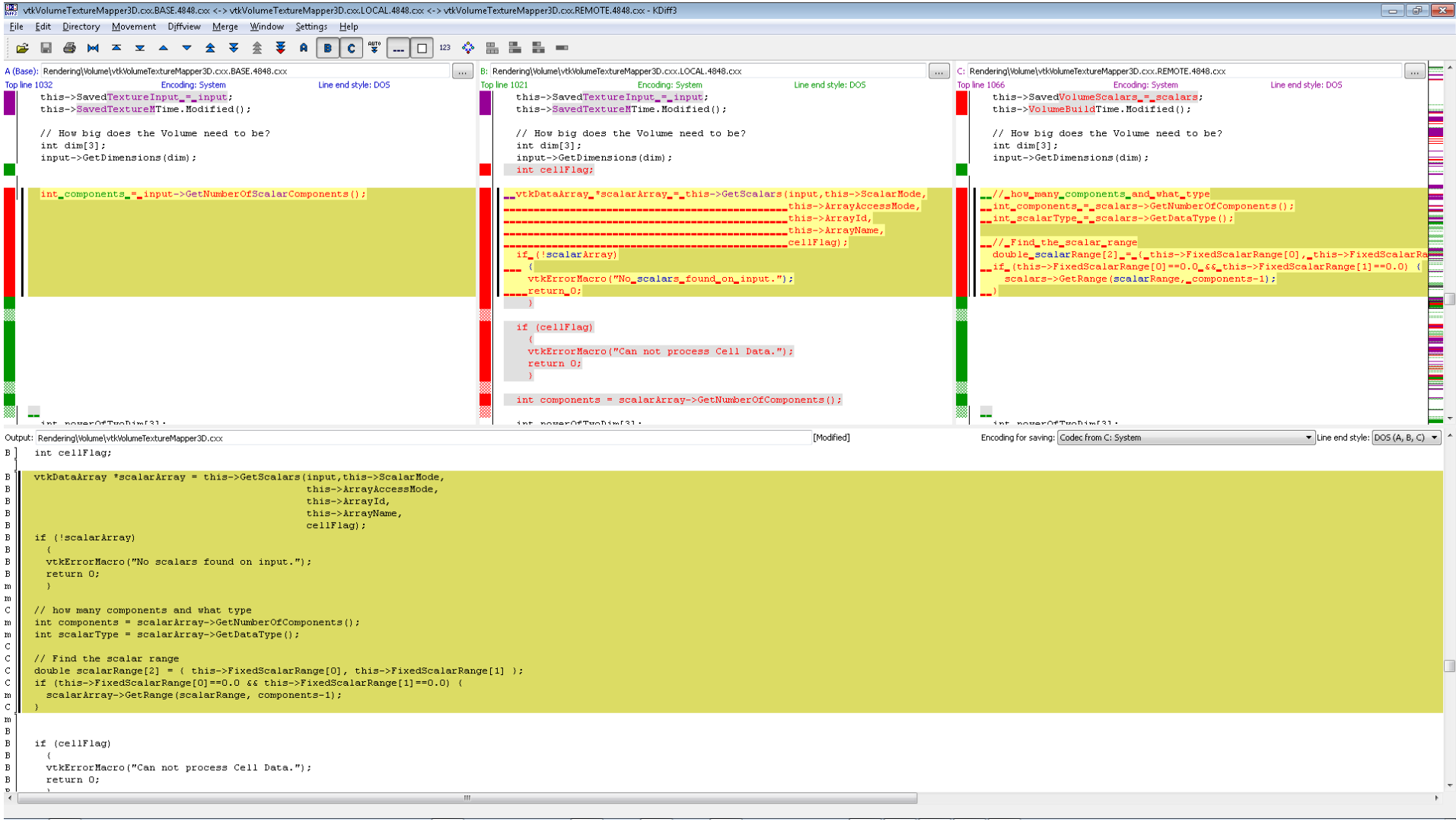
You have to first pull the changes into your branch (which may produce a conflict, which you must fix), before you can push your changes.



Two commits were made by user A, one by user B. The second user to push has to merge first. Then the merge commit become the new HEAD. The commits exist as forked regions and remain independent commits, but the merge commit fixes them into the branch.

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

3-Way merge : KDiff3, will use later



Rebase, squashing, editing history … ask me

When this happens

commit 050c638ef14fc327a676f5aabc276dfdf0bb0609
Introduction to summer school and software engineering
:000000 100644 0000000... cbf549a... A PRESENTATIONS/2013_07_08_Intro_Software_Eng.pdf

commit 81126fca1f1efe0e68d8f4949d104875cd2e6ea3
Was in wrong directory
:100644 000000 cbf549a... 0000000... D PRESENTATIONS/git/2013_07_08_Intro_Software_Eng.pdf

commit 095081bdf2d391e2ef87e81816925e348d3ffc4f
Added introduction to summer school and software engineering.
:000000 100644 0000000... cbf549a... A PRESENTATIONS/git/2013_07_08_Intro_Software_Eng.pdf

You will want to know about rebasing, or merge –squash,

git rebase allows you to go back and change/combine/remove commits earlier in the tree, but can lead to big trouble because commits made by others in their copies of the branch, now sit in different branches (even if the names are the same).

Rebase, squashing, editing history … ask me

Rebasing is very useful and frequently used, but only if you know what you're doing and are sure that nobody else will have conflicts caused by you rewriting branches.

git pull --rebase

is the most commonly used variant (because it won't damage other people). It allows you to replay your commits on top of the remote changes, rather than doing a normal merge.

The main advantage is that it keeps the history linear.

SHA values

Every commit has a SHA, the long sequence of hex values that you see by every commit log.

The SHA is a hash which encode the patch you made, plus the log message, plus the parent commits, date time etc.

The SHA represents an atomic state of the working tree at that moment because it represents the state of 'all' the files as the point on the tree.

You will read that git is different because each commit represents a complete working tree – it isn't just a patch representing the diffs, it encodes its place in the tree of commits and from a SHA the whole repo at that point can be reconstructed.

Examples of Git Usage

Slides taken from older presentations which can be used for ideas follow now if time permits.

Otherwise, exercises should be followed.

Setup

Do this just once.

```
git config --global user.name "John Biddiscombe"
git config --global user.email "biddisco@cscs.ch"
```

or edit ~/.gitconfig

```
[user]
        name = John Biddiscombe
        email = biddisco@cscs.ch
```

All git config commands edit this (global) or local (repo) repo/.git/config files

Creating/Cloning

```
git clone url/to/repo
 is what you'd normally use for a working repo

git init --bare
 sets up a new repo with no working files.
 Just the 'index'

git clone --bare url/to/repo
 is good for making a copy of a remote repo which
 you will use like a server (push/pull)
```

Clones/Copies/Working trees

What happens when you clone?

You get a copy of the remote index (which includes all the remote branches)

What happens when you checkout a branch?

You get a working tree which represents what's stored in the index for that branch.

What happens when you commit to your local repo?

Your local copy of the remote index gets updated, and you have now diverged

Nothing gets changed on the remote until you push

Clones/Copies/Working trees

```
git add somefile
git commit -m 'adding a new file'

 updates our local index. only. Commits are a two
 stage process - unlike CVS/SVN

git rm is not git unadd.
git add, is used to add an initial file, or just a
 'hunk' of changes -
 to unadd a change, you really want to just remove the
 file you added from the index before the add
 git reset HEAD -- file
```

Customization

```
git config --global alias.st status
git config --global alias.wc whatchanged
git config --global alias.dn 'diff --name-only‘
git st or wc or dn

cat ~/.gitconfig
[user]
        ...
[alias]
        st = status
        wc = whatchanged
```

External diff program

in ~/.gitconfig

```
[diff]
 external = c:/Users/biddisco/git-diff.sh
```

and in c:/Users/biddisco/git-diff.sh

```
#!/bin/sh
"C:/Program Files (x86)/WinMerge/WinMergeU.exe" -e
 -ub "$2" "$5" | cat
```

The basics

- **git add path/filename …**
- **git commit**
- **git branch branchname**
- **git checkout branchname**
- **git checkout -b branchname**
- **git checkout -t remote/branchname**
- **git checkout branchname -- path/filename (* useful)**
- **git reset -- path/filename (to unadd path/filename)**
- **git reset --hard 31281c1**

Log

```
git log
 shows you whats on your branch
git log -5
 shows the last 5 logs
git whatchanged -5

git log --pretty=oneline

git log --diff-filter=D --summary
 summarize commits when files got deleted

git log --diff-filter=A --summary
 summarize commits when files got added
```

Log

```
git log -Sstring
 searches logs for the string so you can find when
 a certain line of code got tweaked.

git log -- file
 show the logs that affected a certain file

 the suffix --[space] path is used on many git
 commands to restrict the operation to the
 file/path in question. very handy.
 Get used to git command --option -- path
```

Log+branches

```
git log master..topic_pvfixes
git diff master topic_pvfixes --name-only
git diff master topic_time --name-only
 shows unusual files - because the merge base if
 not correct (our master is way ahead of when we
 created topic_time)
 try
git merge-base master topic_time
git diff f7156718f61 topic_time --name-only

git wc master..topic_time
```

Show

```
 what happened in this commit
git show 83f7324d1ae

 Too much detail
git show 83f7324d1ae --name-only

git show 83f7324d1ae --
 DSMManager/XdmfUtil/XdmfGenerator.cxx
```

Diff

I'd like to see the diff between the file I changed in that last example and now

```
git show 83f7324d1ae --
 DSMManager/XdmfUtil/XdmfGenerator.cxx

git diff 83f7324d1ae --
 DSMManager/XdmfUtil/XdmfGenerator.cxx
```

Diff

```
 I'd like to see all files changed between two
 branches
git diff master steering --name-only

 how about the diff between the current head, and
 the version on branch sttering
git diff steering -- XdmfSteeringParser.cxx

 the diff between commit XXX on branch YYY and AAA
 on branch BBB
git diff
```

Show

show commit logs for all commits on volrender that are not on jb.
You can list multiple branches to include and exclude, e.g.

```
git log volrender ^jb --no-merges --author biddisco
 --name-only
```

for your copy of a remote branch

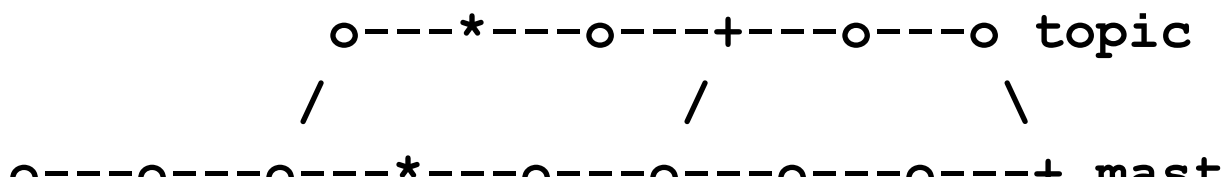
```
git log origin/topic_cosmo ^master
```

rerere

```
You work on a topic
                o---*---o topic
               /
    o---o---o---*---o---o master
and merge back with master to be sure all is still
 ok
                o---*---o---+ topic
               /            /
    o---o---o---*---o---o master
 and resolve the conflicts,
```

rerere

you could just carry on with the merged version



but you end up with lots of merge commits
and you might want to keep your topic clean, so
you ditch the merged stuff and carry on as before.

next time you merge with master - you'll have to
resolve those same conflicts again.

git rerere remembers conflicts resolutions for you
and reapplies them with a prompt

Merge-Commits

git log --no-merges

helps to remove them, but why are they there. because if the merge is not a fast forward (ie just copy one tree onto the other) then

1) **the commits being replayed from one tree to the other are not actually identical any more, so some record that they were modified is necessary. The extra merge comit basically holds the difference between the original commits and the modified commits. often it is actually empty**
2) **if you merge a topic branch and later delete the topic, you can't ever recover that branch without the merge commit records that tell you where those commits are/came from. (also force merge commits)**

Undoing a merge

you can always return to the pre-merge state with

```
git reset --hard HEAD
git merge --abort
```

Or, if you've already committed the merge that you want to throw away,

```
git reset --hard ORIG_HEAD
```

i.e. git always remembers the HEAD just before you start doing a merge so that you can go back to it.

You can always just look at the logs and

```
git reset --hard @#$@$@#$
```

Cherry Pick

```
git cherry-pick ######
```

Pick one commit and apply the diffs from it to the current HEAD.

Extremely useful if you commit multiple small patches to a temp branch and then later cherry pick them onto topic branches.

Headless branches (submodules)

if you checkout some arbitrary SHA, and then start making changes, you can still commit those changes.

but they go into a headless branch. They are just commits hanging off an unnamed tree.

this happens with submodules all the time. because the submodule reference is not a branch, just a SHA

When you make commits on a headless branch, git will warn you that you ought to create a branch to help find those commits (tey still exist, but are hard to find. Google 'git reflog' for tips.

Stash

A cheap temporary branch that you can store changes on.

Use :
edit some files
Realize that you want to commit them onto a specific branch
git stash save "temp work for topic X"
git checkout topic_X
git stash pop

pops those changes off the stash stack and puts them into the working tree.

git commit --m '....message'