Git workflow for Flux

This document explains how to set up a Flux development machine, pull the sources from the Git repository, and get Flux working. At the end of this exercise, you should have the latest Flux version running on your machine (built from source), and you should be able to run all the 3300+ Flux tests.

Pre-requisites

Download and install Visual Studio 2019 or later. You need to turn on the **.Net desktop development** workflow (all others are optional).

Download and install WinMerge. Make sure that this is in the path (turn on the **Add WinMerge folder to your system path)** switch when you install this. (The Flux.Test system uses **WinMergeU.exe** to display differences in text files).

Optional: Download and install **Notepad++**. This is not strictly required, but is convenient to use as a message editor from Git. If you skip this, choose Notepad instead of Notepad++ when installing Git.

Optional: Download and install the **ODA File Converter** from the OpenDesign alliance. Flux uses this for importing DWG files (by converting them to DXF) and if you do not install this, a few Flux tests related to DWG import will fail.

Installation

Download and install **git** from **git-scm.com/download/win**.

Overrides from the default installation process are listed below (the headings in bold are the Git Setup dialog pages, and the notes below each are the recommendations for that page). For any page not listed below, just go with the default installation setting. Items in yellow are critical – working with Flux depends on these being set correctly.

Select Components
Make sure you leave turned on the check-box to install **Git LFS (Large File Support)**.

Choosing the default editor used by Git
I recommend that you install Notepad++ first, and during the initial installation of Git, select that as the default editor.

Adjusting the name of the initial branch in new repositories
Override the default branch name for new repositories, and set that to **main** (this is what we are using for the Flux repositories).

Configuring the line ending conversions
Make sure to select **Checkout as-is, commit as-is**. In short, we don't want any LF to CRLF conversions or vice versa. We have a lot of text and HTML files in our test-data set that use Unix style endings and this conversion will cause tests to fail.

Configuring the terminal emulator to use with Git Bash
I found it simpler to use Windows' default console window than to use MinTTY.

Choose the default behavior of "git pull"
Make sure to select **Only ever fast-forward**. This is especially important for folks like us who are coming over from Mercurial. By default a **git pull** will do a *merge* of the incoming commit, and that is usually not what you want unless we can do a fast-forward merge. The equivalent to a **hg pull** is really a **git fetch** and then one can do a **git merge** appropriately. I think this is definitely better than doing silent merges of the non-fast-forward variety.

Choose a credential helper
Pick the Git Credential Manager Core – this will make it easy to log in just once to Github and you will not keep seeing name and password prompts each time.

Note: If you have already installed Git earlier, make sure that the settings above are in place. You can check by running **git checkout --list** and making sure that the following critical settings are are present. If not, set them up as global or system level settings.

```
filter.lfs.required=true
core.autocrlf=false
pull.ff=only
init.defaultbranch=main
```

One-time Git setup

Set up name and e-mail
Set up your e-mail address (use all lowercase), and name (also with all lower-case and with no space), like **gandalf** or **boromir** or **tombombadil**. If your first name is unambiguous enough (in other words, you are an old-timer) use just that. Otherwise use both first and last names. Do not use an initial.

```
git config --global user.name boromir
git config --global user.email boromir@gondor.org
```

Setting up a diff tool in Git

Here's how to set up WinMerge as the diff tool for Git. Use **git config --global --edit** to edit the global Git config file in your text editor. Then, add (or modify) the following sections (the **cmd=** lines are too long and are wrapping around in the text below, take care when you are adding them, and take care of escaping the quotes and the slashes).

```
…
[diff]
    tool = winmerge
[difftool]
    prompt = false
[difftool "winmerge"]
    cmd = \"C:\\Apps\\WinMerge\\WinMergeU.exe\" -e -r -u -dl \"Old $BASE\" -dr \"New $BASE\"
\"$LOCAL\" \"$REMOTE\"
    trustExitCode = true
…
```

Now we have set up an *external diff tool* for Git to use. However, you wil note that git diff will still use the built-in diff that just dumps on the console. To invoke the external diff tool, we need to use **git difftool -d** or **git difftool --dir-diff**. That will invoke the external tool you have set up. The **--dir-diff** switch is important. Otherwise it will invoke WinMerge once for each file instead of showing a unified interface with all the changed files.

It's painful to keep typing in this each time, so one can set up an alias. Again edit the global config file using **git config --global --edit** and add the alias **git di** to invoke the diff-tool (I also added a few more aliases here that I use):

```
…
[alias]
    st = status
    sw = switch
    di = difftool --dir-diff
    tip = log -1 head
    lg = log --graph --format=\"%h  %C(yellow)%an%Creset  %C(green)%as%Creset  %s
%C(brightblue)%d%Creset\"
…
```

Now, you can just type in **git di** to run a diff using WinMerge.

Setting up a merge tool

You can set up P4Merge as an external merge tool. Download and install it from the Perforce site. Then, edit the global configuration using **git config --global --edit** and add these lines (substitute your own path to the P4Merge application instead of the placeholder below):

```
…
[merge]
    tool = P4Merge
    conflictstyle = diff3
[mergetool]
    prompt = false
[mergetool "P4Merge"]
    path = C:\\Apps\\P4Merge\\P4Merge.exe
    keepBackup = false
…
```

Note the **conflictstyle = diff3** entry in the [merge] section. You'll see how this helps in the section on handling merge conflicts.

Clone the Flux repo

Now that basic Git setup is done, let's get started with Flux. Let's say we want to set up a Flux development environment at C:\Work\Flux. Use this command sequence:

```
md C:\Work
cd /d C:\Work
git clone https://github.com/Metamation/Flux
subst X: C:\Work\Flux
```

Note 1: The first time you use Git to access a GitHub repo on this computer, you will get an authentication prompt asking you to select an authentication method. Use the Web browser method and follow the instructions to log into Github and authenticate this device.

Note 2: Add **X:\Src\Bin** to your path for convenience. The instructions below assume that you have done this

Add the pre-commit hook to your Git repository

We need a pre-commit hook to prevent accidental commits directly into the **main** branch. The hook executable is already part of the Flux repository, but you need to create the **pre-commit** file in the **X:\.git\hooks** folder with the following content (this is just a Bash script that runs the PreCommit.exe filter).

```
#!/bin/sh
exec Tools/bin/PreCommit.exe
```

This step is very imporant – don't skip it!

Note 1: The source code for this tool can be found in Tools/Git.

Note 2: To test if this is set up correctly, go to X:\, then switch to the main branch and try to commit (with no changes):

```
git switch main
git commit
```

If the hook is correctly installed, you should see:

```
***ABORT***: Do not commit to the main branch.
Unstage your work from main using 'git restore --staged',
Create a new branch using 'git switch --create NewBranchName',
and commit your work there.
```

If you don't see this error message you have not installed the pre-commit hook correctly. Do not proceed further, ask me for help.

First build of Flux, and running Flux.Test

Set up your environment variables:

- Add **X:\Src\Bin** to the PATH (the instructions below assume this)
- Set the environment variable **FLUXDEVELOPER** to 1

Then, use this command sequence:

```
dotnet nuget add source X:\Src\Lib    <- we store some Nuget packages here
cd /d X:\Src
dotnet build                          <- should build all Flux DLLs, EXEs, including tests
```

Working on Flux cases

Here is the basic workflow you use to work on Flux cases. As you get more familiar with Git you can bend and break these rules if you know what you are doing.

In general, we are going to use a branch for each case and these branches will be merged into main periodically (typically daily). There are no per-developer branches anymore. New case branches are created starting from main. There are no commits to the main branch directly – that grows only by merging in development from other branches.

First, create a new branch to work on. Make sure to base that branch on the current version of main like this:

```
git switch main                      <- switch to main first
git pull                             <- pull the latest changes, this will never cause a conflict
git switch --create Case.12345       <- create a new branch that forks from main
```

Note that the git pull will never cause a conflict since you will have no local changes on the main branch.

Now, work as normal and periodically stage and commit your changes into the branch you created. Once you are done, push the branch to the Github server (which is called **origin**). The first time you do this, you need to create a corresponding *upstream* branch on the server that mirrors the local **Case.12345** branch you created. So you need to do this (if you forget this, git push will helpfully remind you)

```
git push --set-upstream origin Case.12345
```

Subsequently, you can just do a git push.

Eventually, you may have a few commits on this branch and you are then done. Go to github.com and create a *pull request* for this branch. This signals to me that there are changes ready to be merge into main.

You can then start working on another case by creating a new branch and starting afresh. Remember to do the sequence with git switch main etc again. Don't fork the new branch from your already existing Case.12345 branch!

Switching between cases

Now with Git you have the flexibility to switch to some other case and work there without having to complete your work on the first case. Suppose you also want start working on Case.67890 without necessarily completing the first case. Commit your changes to Case.12345 to your local branch and make sure you have a clean working directory. Now, create a new branch (remembering to switch to main first) and work there. You can now commit changes to that branch, switch back to the previous one and work on both of them in parallel.

Finally, when one of them is done, you can raise a pull request for this.

This also gives me the flexibility to merge Case.67890 or Case.12345 without necessarily having to merge the other one.

The pull request in Github is structured like a discussion. If I am not happy with some changes, I can use the pull request to make some notes. You adjust the code further based on these notes, commit and push again. This throws the pull request back to me and I can check. After some back and forth, the branch is merged good to go and is merged. After this, we typically delete the branch (like Case.12345) since it has served its purpose.

Pull periodically from main

Each time you start work on a new case, you are always starting with the latest main (thanks to the git switch main / git pull) that you do before creating a new branch. However, if work on a case takes a while, it might be useful for you to periodically merge the latest changes from main into your branch. Let's say you are working on Case.12345. You will then do:

```
git add Changed.cs                      <- assume we are now on Case.12345 branch
git commit -m "Case.12345 – Step 3: added local rendering support"
git switch main                         <- Switch to the main branch
git pull                                <- Get the latest version of it
git switch Case.12345
git merge main                          <- merge the changes in main
```

Note that this merge may cause a conflict if somebody else has edited the same area of the same file. You have to resolve this conflict.

Handling merge conflicts

Occassionally you will get merge conflicts (though less often than you would with Mercurial). Let's take a simple example. Suppose we started with a text file like this:

```
Welcome to Flux.
This is a sheet metal CAM software.
It can handle Bend and Laser machines.
```

Next, developer A, working on some branch changed this as follows (the changes are highlighted in yellow):

```
Welcome to Flux/Praxis.
This is a sheet metal CAD-CAM software.
It can handle Bend and Laser machines.
```

Meanwhile, developer B, working on some other branch changed this same file as follows:

```
Welcome to Flux.
This is a sheet metal CAM system.
It can handle Bend, Laser and Fold machines.
```

Now suppose developer A pushes their change and files a pull request. I will merge this into the **main** branch and at this time, there will be no conflicts – the merge result will be a fast-forward merge (see the Git documentation to understand this).

Later, developer B, wanting to continue working on the code, tries to pull the changes from main (what we used to call a **refresh** in Mercurial), like this:

```
git switch main
git pull                      <- get the latest changes on main
git switch BranchB
git merge main                <- and try to merge them in
```

You will get a result from git that looks like this:

```
Auto-merging greeting.txt
CONFLICT (content): Merge conflict in greeting.txt
Automatic merge failed; fix conflicts and then commit the result.
```

At this point, you have two options:

1. You can resolve the conflict using the external merge tool by using **git mergetool**. This brings up the conflicts in P4Merge and you can resolve them there.

2. You can just use a text editor to resolve the merge. Each area of conflict will have *conflict markers* around it. In this case, the file is so small that every line of it was involved in the conflict so the markers appear like this if you edit greeting.txt:

```
<<<<<<< HEAD
Welcome to Flux.
This is a sheet metal CAM system.
It can handle Bend, Laser and Fold machines.
||||||| 84df7ee
Welcome to Flux.
This is a sheet metal CAM software.
It can handle Bend and Laser machines.
=======
Welcome to Flux/Praxis.
This is a sheet metal CAD-CAM software.
It can handle Bend and Laser machines.
>>>>>>> main
```

Note that there are actually 3 versions of the text shown in the file. The part in green (in the middle) is the *original* version from before the branches diverged. The part in yellow is the local version of the file that was just edited (this is called **ours** by Git). And the part in cyan is the remote version of the file that the other person has edited (Git calls this **theirs**). Note that the part in green (the original file) is included here only if you had set up **conflictstyle = diff3** as explained in the section above on setting up the merge tool. I find this green text (original) to be very useful when figuring out how to resolve the conflict, so I strongly recomment you set up this conflictstyle.

Now you can easily clean up the conflict in the text editor since you have all 3 versions available and you can pick the exact version you want to keep. Usually, I find this is the easiest way for me to work, since I can just search for all occurances of <<<<<< in the code and quickly find all the merge conflicts.

If you edit the file manually, you will finally do this:

```
git add greeting.txt         <- mark the resolution
git commit                   <- and complete the merge
```

If you forget to resolve one of the conflicts, the pre-commit hook we have installed will prevent the commit from happening. Edit the offending files again to remove all the conflict markers and try the commit again.

If this conflict resolution becomes really complex or you are unsure of how to resolve some of the conflicts, you can just back out of the whole mess by using **git merge --abort**, push the code as is and file a pull request. Then it becomes *my* problem to merge the incoming changes into main when I am handling the pull request.