Working with Git (Version Control)

- Git Overview
- Git Terms
- Tutorials
- Working with PureRef
- Configuring Git to rebase when pulling upstream changes

Git Overview

All professionally managed projects require some kind of version control system.
This type of software goes by several names:

- Version Control System (VCS) / "Version Control"
- Source Code Management (SCM) / "Source Control"

We'll stick to the name "Version Control" to keep it simple. Professional game development typically uses either **Perforce** or **Git**. - We will be using Git, as it is free and open source. While it's less popular for game development than Perforce is, it is a tool that you can continue to use for free in your personal projects after you learn to use it during BUGJam.

There are four components to Git that you need to know about:

- Git
- Git LFS
- Git GUI
- Software Forge

Git

Git is a "decentralized" version control system. It is a command-line tool used primarily for software development. It manages source code repositories, and tracks versions of files that you check in. Git by itself is a powerful tool, but it is incomplete for our needs. Since it is decentralized, it doesn't need to connect to a central server in order to function, but without a central server you can't easily collaborate with other people.

Git LFS (Large File Storage)

By default, large art assets (binary files) aren't well supported by Git. The de facto solution to this is a plugin for Git called "Git LFS" (Large File Storage). You will not interface with LFS differently than you would with regular Git, it runs in the background and moves your large files into a large files storage container on the server. Once it's installed you can forget that it even exists.

Git GUI (Graphical User Interface)

Git is a command line tool, this means you interface with it by typing commands into the terminal, so unless you're well versed in typing all of your commands into the console, its usually not a desirable way to work. So third-party tools are available to add a graphical user interface to Git.

There are dozens to choose from but we will be focusing on a free and open source tool called "SourceGit".

Git GUI tools are all just wrappers on top of git, and they just issue commands via the Git command line on your behalf, they all do the same thing, but some offer a much better experience than others. SourceGit is very new, and a little rough around the edges as it is in rapid development, but it's one of the best FOSS options.

Software Forge

Git's decentralized nature means that it doesn't have it's own system for hosting the repository. During collaboration we need a place to push our changes to so that others can pull those changes.

Among very many other awesome features, this is something that "Forgejo" provides. This is hosted on the server, you do not download this, instead you access it via a web browser.

Git Terms

Repository / "Repo"

A repository is the what Git uses to manage projects. Every version of every file, every change, every diff, every reference log, every blob, is all stored in the repository. You can think of it as the thing that contains the entire project.

Some projects put all of their project content into a singular repository making one massive "monorepo". This is not really appropriate for game development since it's more ideal to separate out the art source content from the in-game content. For BUGJam we will have two repos: One for art source content, and one for in-game content.

Origin

Each of the project repositories are stored in the software forge on the BUGJam server at https://git.bugjam.dev/ Origin refers to the version of the repository that is hosted on the server. Since it's not stored on your computer, we call this a "remote" repository. Origin is the "source of truth" for the project.

Clone

Nearly identical to "downloading", cloning is a function of Git that can reach out to a remote repository and copy the files down to your computer. With one very important difference: It keeps a connection to that remote repository so that you can track your changes and push them back up to the remote repository.

History

A full list of every change that has happened within the repository. Every file that was checked in, every modification, addition and deletion is stored in the project's history.

Branch

Git is able to track multiple timelines of history for the project. Whenever you make a split in that history, it's called a branch. The primary branch is called 'main'. The main branch is meant to represent a stable state of the project, you should always be able to view the main branch and play a version of the game that's free from crashes or major issues. We do not develop directly in the main branch. Other branches will contain the actively developed new features, this ensures that each contributor can add content to the game without breaking the content that other people are working on at the same time.

Local Changes

Any change you make to the files within the repository will automatically be detected by Git: Creating a file, editing a file, moving a file, deleting a file. These will all show up in your local changes. These changes will not immediately show up for other users. Local changes are volatile, they aren't checked into the version control system yet, so be careful, if you revert your local changes without committing, they are lost forever.

Diff (Difference)

Whenever a change is made, Git isn't just tracking the change that happened, it looks through the file to track the difference between the two versions. For text files, it's able to show you line-by-line what changed, this is the "diff".

Stage

After you've made changes to files in the repository, Git will be aware of the differences in the files and they will appear in the local changes. But they still need to be added as new versions in the repository. We have to tell Git which files we want to add, picking which files we want is called "staging" those files.

Commit

Every time you have a change you want to submit to the repository, you need to commit it into Git's history. This represents an actual check-in which saves your work into the repository. Without a commit, your work is in a volatile state, you must commit to make your changes permanent. First stage your files, then write a message, then commit. Note that this still will not make these changes available for the other users yet, since the commit only happens on your local computer.

Commit Message

Sometimes it's unclear what changes took place within a commit, all commits must have a commit message which is usually one or two lines of text describing the changes you're about to check in. Example: "fire extinguisher: Create block mesh". This will make it clear what this commit contains when we view it in the project history.

Commit Hash

Every time you commit it is given a unique identifier in history, this is the hash. You won't need to concern yourself with this unless you're doing very advanced Git operations.

Fetch

Because Git is decentralized, it's unaware of the remote repository on the server for most operations that you run on your local machine. Changes could be happening upstream from you and your local copy of the repository will become out of date. "Fetch" is an action that reaches out to the remote repository to ask for the latest information.

Pull

Where "Fetch" just asks the server for the latest information, "Pull" will take those changes and pull them into your local copy of the branch. This is how you stay up to date. Keep in mind that "Pull" actively makes changes on your machine, if you're not careful it can erase your local data in favor of what's on the server. Don't forget to commit your changes first.

Push

"Push" is the opposite of "Pull", it takes the commits in your local repository, and pushes them up to origin. This is how you share your changes with the rest of the team.

Merge

If multiple branches were used, you can't see the content of both branches at the same time. Eventually you have the "merge" the content of one branch into the other to unify the timeline.

Tutorials

Overview of Git:

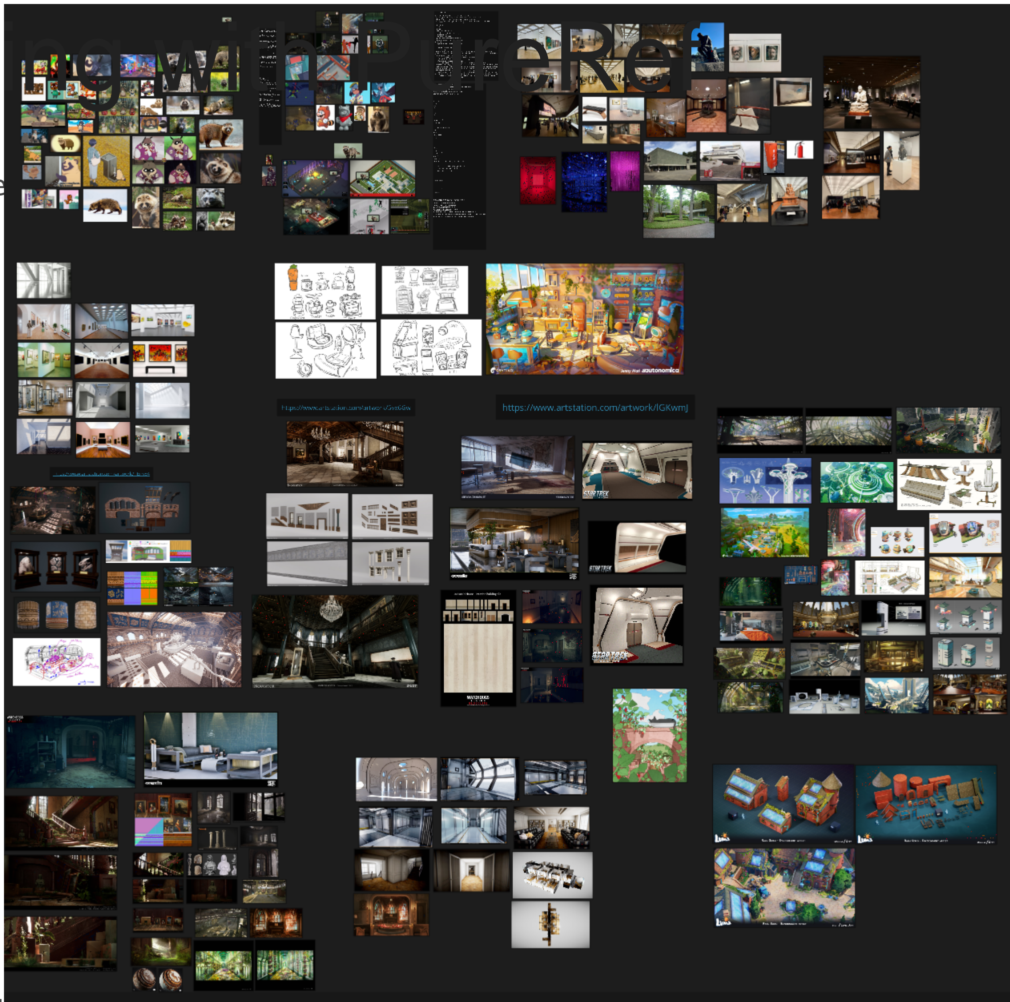
https://www.youtube.com/embed/ap_Mvgl0h1g

How to clone the repositories:

https://www.youtube.com/embed/6gzm0WX_O3E

Working with PureRef

PureRef produce ... ce images



and concept art.

Optimization - Use linked images, not embedded

Images are large files, often around 500 KB - 10 MB each. If you add 200 images to your PureRef file, that can easily add up to 100 MB of (data depending on the compression used in each image). Let's keep using 100 MB as an example to keep it simple.

Committing in each of these images by themselves is manageable by Git LFS; These images files aren't likely to be modified, so they only have to be committed to the repo once. This means we will use the storage space and bandwidth of each image once, a total of 100 MB. - Not bad.

However, if you embed the images into the PureRef file directly, it will produce one massive 100 MB `.pur` file instead of saving the individual images. The total size is the same, but now there's a horrible issue: Every time you make a change to the PureRef document such as:

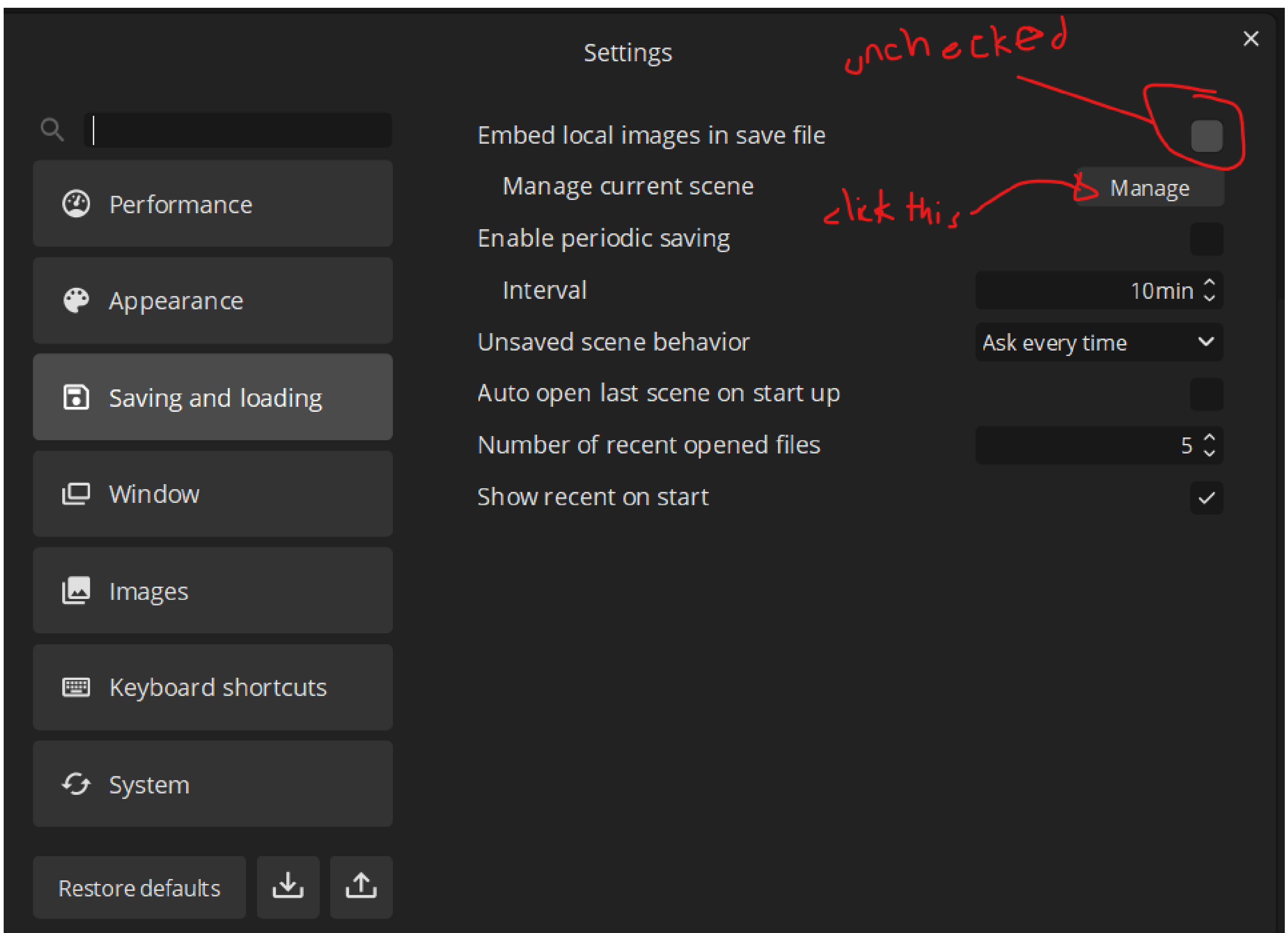
- Adding one new image
- Removing an image
- Moving an image around
- Or even just panning the canvas around

When you save it, the ENTIRE 100 MB has to be updated when it's committed to the repository. So every time you save you're paying the cost of all of the embedded images every time. If you save that file 20 times, that's 1 GB used up on the server's storage space, and 1 GB that every user has to download to their machines for basically no benefit.

Because of this we must instead save the images individually into a "references" folder, and use the "Linked" images option in PureRef, not the "Embedded" option. This will keep the `.pur` file only around 300 KB while the 100 MB of images are individually saved. This way if we make 20 commits with the .pur file it will only equal a total of 6 MB instead of 1 GB in the previous example.

Setting up linked files in Pureref

1. Right click and choose Settings (Ctrl + U), and go to Saving and Loading:



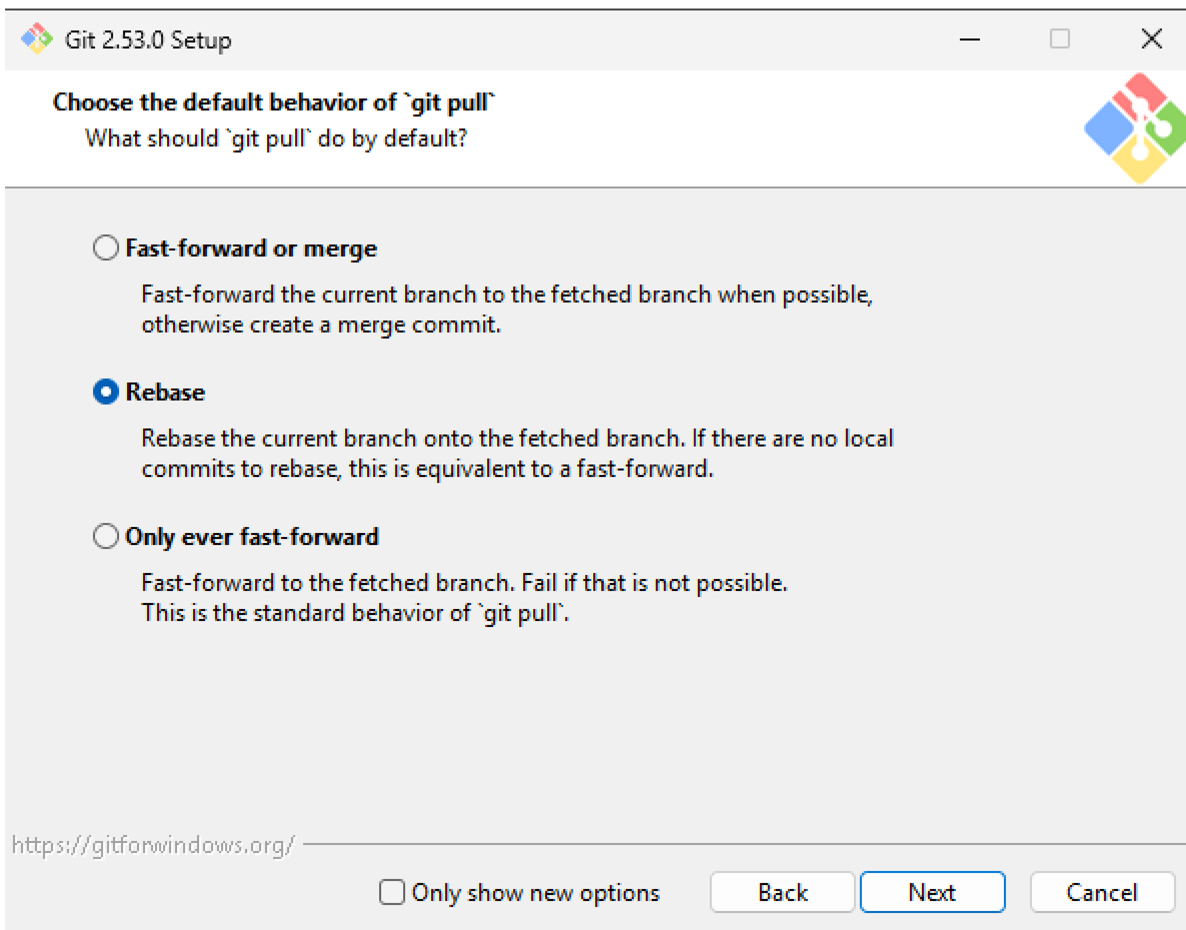
2. Click "Manage" and set up linking images with the proper folder output. It'll warn you that this can't be undone.

And that's it!

Configuring Git to rebase when pulling upstream changes

We are experimenting with following a Git workflow where everyone is allowed to work from and commit directly to the `dev` branch. This comes with the consequence of having your local version of `dev` frequently falling behind upstream changes made to the `origin/dev` branch (the server's version of your local `dev` branch) as other people push their commits. This can easily lead to a pitfall when pulling changes from the `origin/dev` branch into your local `dev` branch because Git defaults to fast-forward or merge when pulling. This means that by default, if the branches diverge (meaning there are both upstream and local changes) it will create a commit with the subject line `Merge branch 'dev' of https://git.bugjam.dev/BUGJam/pounce-game into dev`. These extra merge commits clutter the repo by creating unnecessary cruft in the commit history and make it difficult to understand the sequence of changes. What needs to happen instead is your local changes need to be rebased on the upstream changes when pulling. The instructions below show how to configure Git on your system to default to rebase when pulling so it will do the right thing for you automatically.

On Windows, you can make this change while installing Git when you get to the prompt in the installer asking "What should `git pull` do by default?". It defaults to Fast-forward or merge. You need to set it to Rebase like in the following screenshot. If you did not change the default pull strategy in the installer, you can still change it manually from the command line. See **On all platforms** section below.



On MacOS / Linux, installing git works differently and does not have an installer like on Windows. However, the default pull strategy is still fast-forward or merge, and needs to be changed to rebase.

On all platforms, in a command prompt / terminal set the `pull.rebase` option to `true` with the following command:

```
git config --global pull.rebase true
```

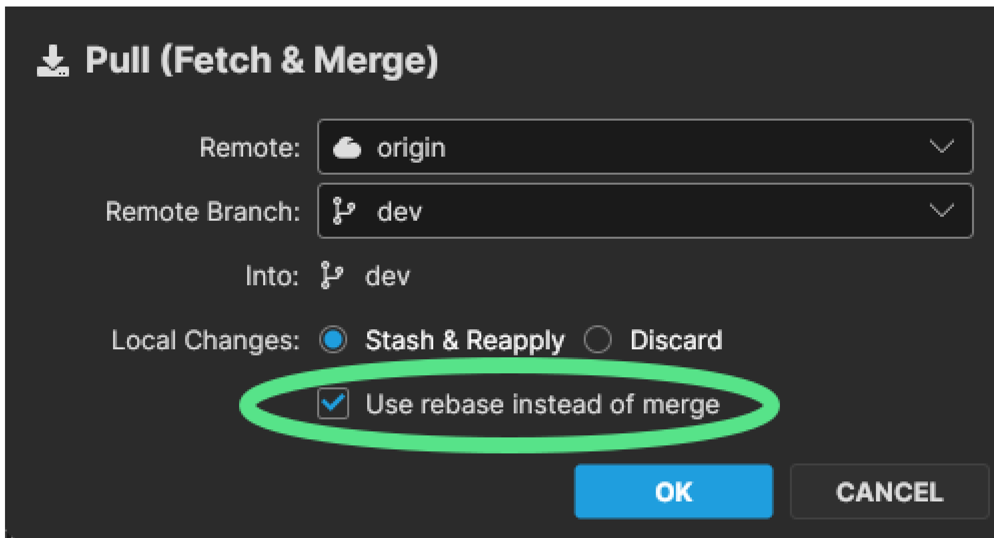
Confirm the change has been made by running the following command:

```
git config --get pull.rebase
```

the command should just print `true`.

In SourceGit, with the above Git configuration change, it should default to use the system Git pull strategy configuration. Still, make sure when you are pulling that the checkbox for "Use rebase instead of merge" is checked before pulling.

It should look like this:



Going forward please do not merge `origin/dev` into your local `dev` branches. You should be rebasing your local `dev` branch changes on `origin/dev` instead. If you ever see the `origin/dev` branch get merged into your local `dev` branch with a merge commit with the subject line `Merge branch 'dev' of https://git.bugjam.dev/BUGJam/pounce-game into dev` (or similar), please do not push that extra merge commit to the repo. Instead, please reach out to one of the many Git experts on our team for help with correcting the mistake before you push your changes.