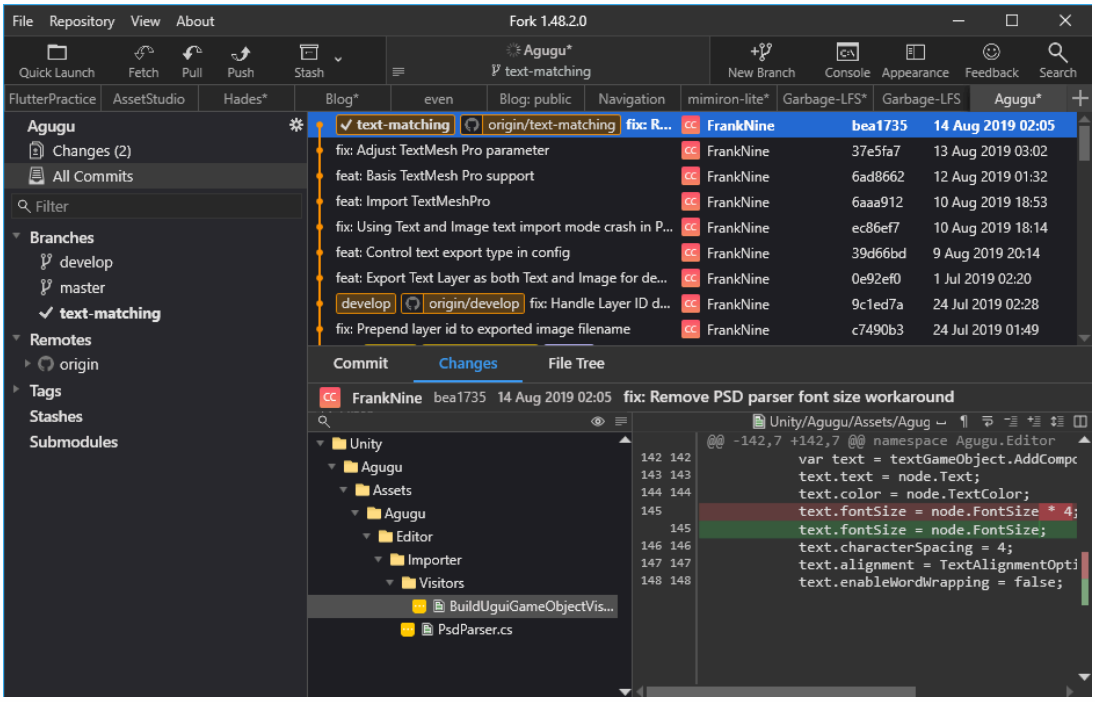
*This is the translated version of my 2020 Unity 和 git post with some minor updates and all the Chinese reference links removed. Sorry I am not a fluent English speaker. Please leave a comment if something seems wrong/out-dated. Thank you.*

I've been using Git to version control Unity projects for a while now. Recently, my company open-sourced our Unity builder mimiron-lite, along with a new set of Git configuration files that are now being used as Git configuration standard of my company . I would like to share some insights regarding the thought process behind these configuration files and the experiences of using Git.

If you prefer a TL;DR, you can directly download the Git configuration files. The latest configuration now includes an updated `.gitignore` for Wwise, available at: https://github.com/FrankNine/RepoConfig.
This repo was forked from: https://github.com/rayark/repo-config.
The company has released this configuration under the MIT License.

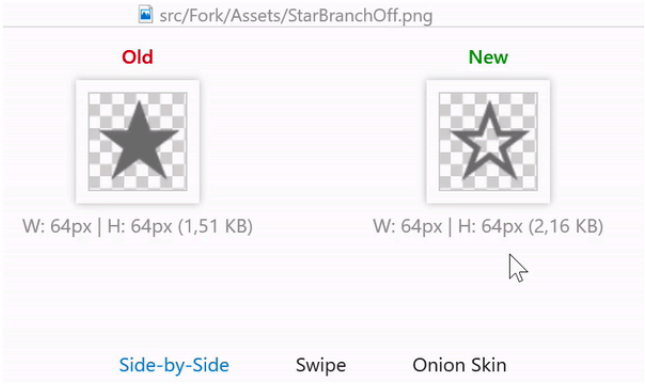
## GUI Client

Previously, in order to meet the needs of both technical and non-technical personnel for using Git, we conducted a survey of various Git clients. Currently, we have chosen the paid version of Fork. Prior to that, we were using the most common free client, SourceTree, but encountered issues with login which happen way too often and require technical personnel to solve. As well as slowness when dealing with large files and lack of preview support for images and LFS files. Some colleagues also opted for Git Extensions bundled with Git for Windows, but it's less user-friendly for artists, and those who are accustomed to clients with a separate window GUI may not be comfortable with just the Shell Extension. Another free option is TortoiseGit, which has good support for artwork assets, including displaying images within LFS, but may not be as convenient for more complex Git commands like Blame and Rebase. GitHub Desktop is rudimentary. While GitKraken has many features for artwork-related tasks, it struggles with performance in large repositories due to being built on top of Electron. SmartGit has extensive features but lacks good support for artwork assets, and also has a relatively high annual fee.




Fork

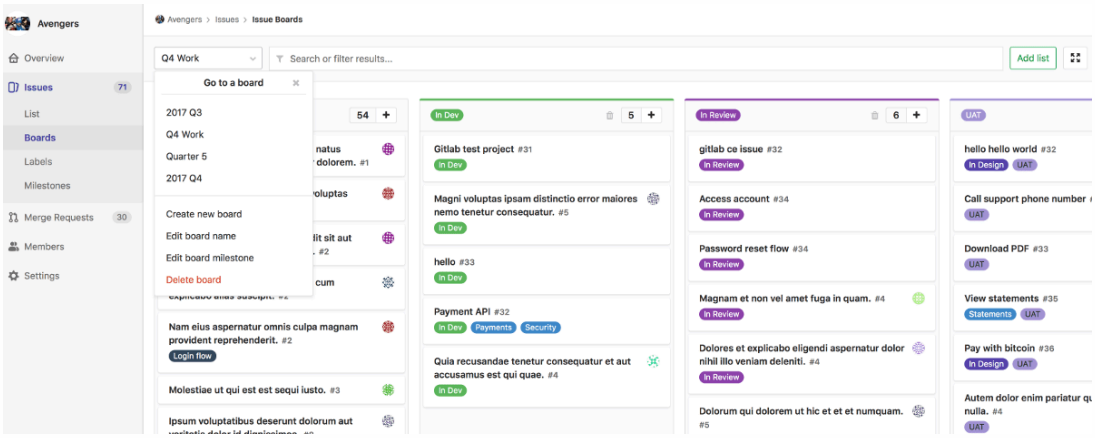
Currently, Fork provides native clients for Windows and Mac, and the speed is generally good. The payment model is currently an one-time purchase of a license key able to activate on up to three computers at $50 USD, which is more affordable compared to subscription-based tools (although things might change in the future). The interface is clean and satisfies most development needs, with a well-designed built-in rebase and merge interface. A killer feature for game development is the built-in image preview and diff (although the Mac version only has side-by-side view and lacks swipe and onion skin options), and it also works smoothly with LFS configuration.




https://git-fork.com/blog/posts/forkwin-1.38/

## Server

Most Git hosting cloud services have limitations on capacity or bandwidth for Git Large File Storage (LFS) (GitHub's LFS terms), which is not ideal for game projects. Riot's "Legends of Runeterra" mentioned in their shared article that they use GitHub Enterprise with Artifactory as their LFS server. Using hosting services means Git operations have to go through the company's external network, which may not be practical for projects with very large assets, especially considering if the Internet conditions might not be good. If possible, it's recommended to set up a GitLab instance on a dedicated server or NAS. GitLab provides Git, CI Runner, project management, and code review functionalities. Although the project management interface may not be as visually appealing or user-friendly as Trello or other tools. However, these functionalities in GitLab can reference each other, such as referencing the progress of merge requests or specific CI pipeline results in issues, which has an invaluable advantage. Just keep in mind that self-hosting requires more IT expertise and backup considerations.

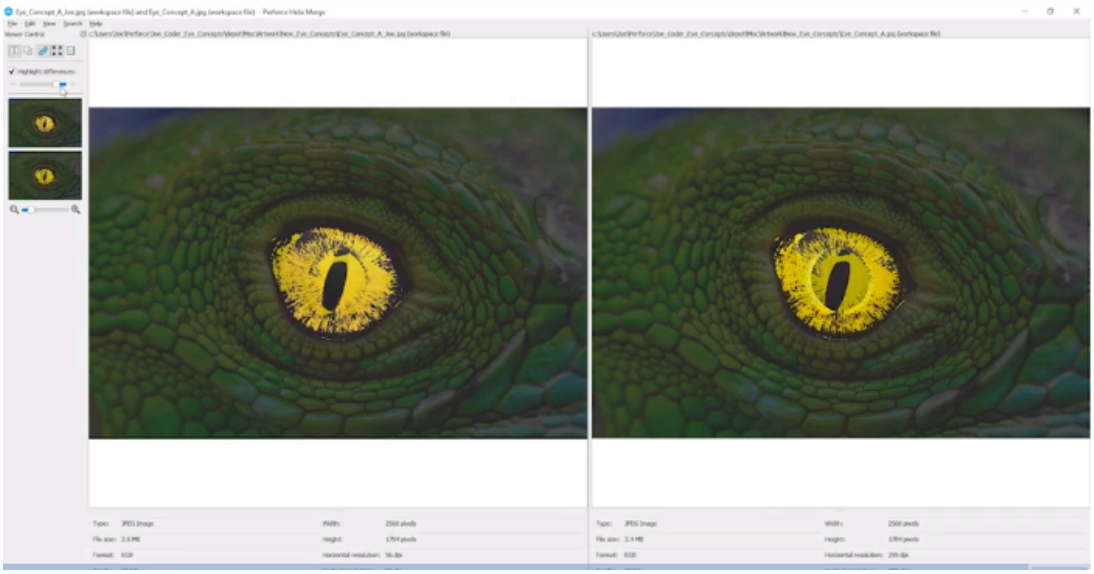



https://about.gitlab.com/stages-devops-lifecycle/issueboard/

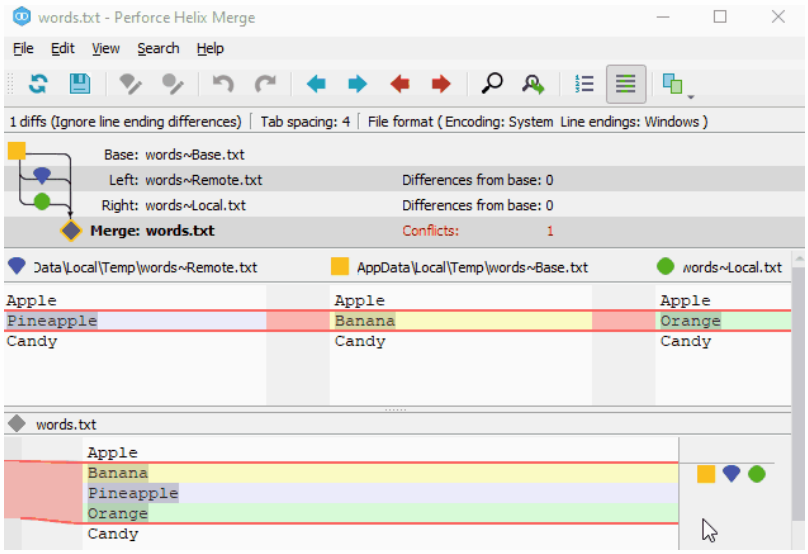
## External Merger

As for merge tools, most people use the built-in KDiff3 in Git. However, based on my experience with Perforce from my previous job, I really like the merge tool of Perforce. Fortunately, the merge tool of Perforce, P4Merge, can be installed separately and used for free. Most Git clients allow you to configure which external merge tool to use, and many of them can recognize P4Merge. P4Merge is very user-friendly

for cherry-picking changes during 3-way merging and editing merge results, and its diff interpretation is better than other tools. Additionally, it has a powerful image diff feature that can highlight the differing pixels. The only noticeable downside is that it does not properly support non-English characters, resulting in garbled text and UI issues even if the encoding is set to UTF-8.



P4Merge image diff




P4Merge 3-way merge

# Configuration

## Unity

To perform version control on Unity's internal AssetSerialization, it is considered a must to configure Unity Editor as Force Text and Visible Meta Files. With Force Text, Unity resources will be stored as YAML files that can be diffed, and Visible Meta Files allow GUID synchronization with other collaborators working on different machines. In the past, during the era of Unity 4.X, when the editor was 32-bit and could not address more than 4GB of memory, Force Text was sometimes disabled due to increased memory usage. However, with the current 64-bit Unity editor, this issue no longer exists, and Force Text and Visible Meta Files have become default options in later versions.

## .gitconfig

### core.ignorecase

The most important Git config that all the team members has to set is:

```
Code
1  git config --global core.ignorecase true
```

It's to make Git treat files with different capitalization as the same file. On systems such as Mac and Linux, which have case-sensitive file systems, files with different capitalization are treated as different files, whereas on Windows, they are considered the same file. If this setting is not configured, Mac users may push files with only different capitalization into the repository, which Windows users will be unable to pull, or encounter situations where files go back to "Modified" state immediately after being reverted. If you need to perform any actions that involve changing just the capitalization of a filename, please use `git mv` instead of deleting the file, committing, and then renaming the file and committing again, as that may break the history of the file.

As we have set `core.ignorecase true`, we do not need to specifically include patterns that match different capitalizations in our configuration of `.gitignore` and `.gitattributes`.

### core.fsmonitor / feature.manyFiles

```
Code
1  git config --global core.fsmonitor true
2  git config --global feature.manyFiles true
```

For the sheer size of game projects, it is recommended to use any settings that can improve the speed of Git operations. The File System Monitor can utilize operating system features to accelerate detection of file modifications, while Many Files can speed up detection of untracked files.

Reference:

- We Put Half a Million files in One git Repository, Here's What We Learned

## .gitignore

The `.gitignore` file is mainly used to ignore "files that should stay on the local machine but should not be shared" like Unity's `/Temp/` and `/Library/` directories, and the code editor settings. Files that should not be kept after use, such as Xcode project directories generated by Unity or APK files, are not included in my `.gitignore` file, so that they are recognized as Untracked Files, making it convenient to clean them up locally or on CI Runners by using `git clean`. One tiny detail is that if a pattern starts with `/`, it will only match files or directories at the root of the repository. For example, I would write `/Library/` for Unity's Library directory to avoid ignoring other paths with the name "Library" inside. You can refer to the following example for setting up `.gitignore`:

https://github.com/FrankNine/RepoConfig/blob/master/.gitignore

## .gitattributes

The `.gitattributes` file is more complicated since it's related to both LFS and CRLF.

### LFS (Large File Storage)

Let's talk about LFS first. Basically, LFS (Large File Storage) moves files with specific file extensions out of Git and stores them on an LFS server, leaving only a Pointer File in the original Git repository. The Pointer File contains an oid (Object ID). When needed, LFS uses the oid in the Pointer File to lookup and download

the actual file from the LFS server, replacing (Smudge is the actual name of this process) the Pointer File with real content.

Here is an example of the contents of a Pointer File, which is typically around 130 bytes. Keep in mind this file size, if you see a file of this size, you may need to check if the LFS replacement process was executed correctly.

Code

```
version https://git-lfs.github.com/spec/v1
oid sha256:4d7a214614ab2935c943f9e0ff69d22eadbb8f32b1258daaa5e2ca24d17e2393
size 12345
(ending \n)
```

Nowadays, most Git clients are capable of detecting LFS and automatically setting up LFS configuration during cloning. However, if there are issues with LFS initialization, you can manually execute the following commands:

Shell

```
git lfs install
git lfs pull
```

Using LFS can reduce the size of Git history, as Git only stores Pointer Files. This can decrease the local disk space usage during cloning and improve efficiency, especially for large binary files in 3D game projects. However, there are some drawbacks:

- Git operations now require continuous communication to the LFS server, which results in the loss of Git's fully distributed and offline capability.
- Occasionally, issues may arise with LFS replacement, such as Pointer Files not being replaced with the actual file contents during cloning or fetching. Rarely, there may be cases where files are lost in GitLab's LFS server, resulting in failed fetches. In situations where LFS and Fork functionality are used simultaneously in GitLab, re-synchronization may be needed for the LFS repository. It is currently recommended to avoid using LFS and Fork functionality together on GitLab.
- Some features may not work properly in an LFS environment, including certain tools that may not be able to diff LFS files, and `gitlab-runner exec` unable to run in an environment with LFS files.

It is recommended for game projects to use LFS since the majority of the repository size is from binary files, and it is better to adopt LFS early on. Because binary files that have already been committed into the Git history will remain in the history, and modifying `.gitattributes` will only affect newly added files. If the repository size becomes unmanageable, migrating to LFS may be the only option, but it requires all users to clone the repository again, which can be a massive undertaking.

To perform migration to LFS on existing repository, first, track all branches locally, as `lfs migrate` only applies to local branches:
https://stackoverflow.com/questions/379081/track-all-remote-git-branches-as-local-branches

Then execute following commands (This is an example of rewriting history for `.png` and `.psd` files from binary blobs to LFS pointers)

Shell

```
git lfs migrate import --everything --include=".png,.psd"
git reflog expire --expire=now --all && git gc --prune=now
git remote set-url origin <New repository location>
git push --all origin --force
```

Another common situation is that you modify the `.gitattributes` file to include new file extensions into the LFS scope, but the existing files are not replaced with Pointer Files. In this case, you will receive a following warning during Clone or Reset:

```
Code
1  Encountered 1 file(s) that should have been pointers, but weren't
```

If you don't want to rewrite whole history but just want to replace current files with Pointer Files, you can use:

```
Shell
1  # from https://stackoverflow.com/a/51626808
2  git rm --cached -r .
3  git reset --hard
```

Or

```
Shell
1  # from https://github.com/git-lfs/git-lfs/issues/3421#issuecomment-610489798
2  git add --renormalize .
```

Then commit/push the changes to the repository.

Just add the following to `.gitattributes`:

```
Code
1  *.png filter=lfs diff=lfs merge=lfs
```

You can manage PNG image files with LFS like this (since `core.ignorecase true` is set, we don't need to write `*.png` as `*.[pP][nN][gG]`). Basically, all binary files that cannot be directly edited as text should be added to LFS, but for Unity Prefab (`.prefab`), Assets (`.Assets` but not `LightingData.asset`) and Scene (`.unity`) files, I would choose not to add them to LFS. After setting AssetSerialization to Force Text, the YAML structure of Unity GameObjects can be read and compared as text diff, which can be very helpful to verify if intended changes to Prefabs or Scenes accidentally affect other SerializedFields.

To understand how to read YAML, you can refer to the official documentation:
Understanding Unity's serialization language, YAML

If you would like to read changes in Scenes and Prefabs this way, remember to use AssetDatabase.ForceReserializeAssets on all Prefab and Scene files when upgrading Unity to the newer version. By default, Unity only upgrades YAML file format to new version when changes are made (this is a design philosophy of Unity Editor, Material files are also exhibiting this kind of lazily upgrade behavior), but this can result in editing changes and upgrade changes getting mixed together that makes it difficult to read in diff. Therefore, it is necessary to force the file upgrade via `AssetDatabase.ForceReserializeAssets` when Unity is upgraded but the project is not yet modified.

Unity provides its own YAML merger, but after trying it, I found that the results are occasionally very strange. Currently, I only read YAML diff and don't automatically merge Scenes and Prefabs with this tool.

Another special case is `.dll`. When a DLL is not LFS smudged correctly, it will cause compilation errors. If Unity complains about being unable to find symbols that should have been defined in the DLL, remember to check if the file size of the DLL file in the project see if it's only 130-byte. Which means it is just a Pointer File, and the LFS execution failed.

## CRLF

Another big pitfall with `.gitattributes` is CRLF handling, as Windows (CRLF `\r\n`), Mac (CR `\r` before OSX, LF `\n` after), and Linux (LF `\n`) have different line endings. Git defaults to `autocrlf`, which automatically converts line endings during checkout and commit. However, if not configured carefully, this can cause two issues:

- Applying conversion to binary files that are not actually text files, resulting in corrupted files when opened locally.
- Some files constantly change line endings, most commonly `.meta` files, causing numerous changes in changelist that are not man-made.

Many people may use this popular Unity `.gitattributes` config:
https://gist.github.com/nemotoo/b8a1c3a0f1225bb9231979f389fd4f3

But it sets Unity's `.asset` files to LF, and Unity has many types of `.asset` files, as I previously investigated:

- CRLF
  - `.asset` files under `ProjectSettings` folder
- LF
  - Tilemap configuration files
- Still binary under Force Text setting
  - Lighting Data
  - Terrain Data
  - NavMesh Data

Forcing conversion to LF can cause issues, as reported by others but there's no update yet, so it is recommended not to use this `.gitattributes` sample.

My final configuration starts with:

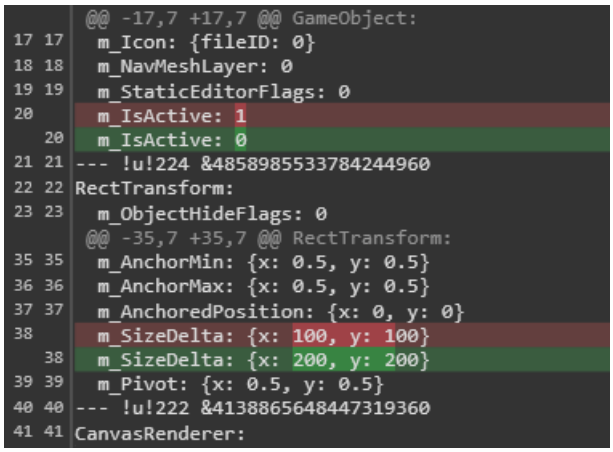
```
* -text
```

Text attribute is removed to disable line ending conversion for all files by default. Then, using the trait of `.gitattributes`, the rules written below can override the rules written above. Adding line ending conversion for pure text files in a whitelist manner.

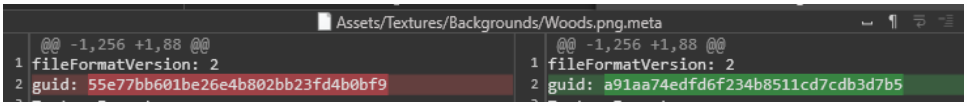
Not written as:

```
* binary
```

This is because Binary represents `-text -diff`, and we want to disable line ending conversion but keep text diff for `.scene`, `.prefab`, and `.meta` files. This allows us to observe changes in GameObjects, Components, or GUID changes in `.meta` files through diffs.




In the case where Prefab is not added to LFS and does not have -diff, the diff in the Git client will be displayed like this. You can see that we changed the size of RectTransform to 200 x 200, but accidentally turned off the GameObject at the same time.



Seeing changes in GUID inside `.meta` files in changelist is very dangerous, as it indicates that Unity references may be lost.

As for the issue of inconsistent line endings in `.meta` files, it was found through experimentation that Unity Editor saves them with LF line endings regardless of the OS, so it is hardcoded:

```
*.meta text eol=lf
```

`".mat"` files should also be forcing LF line endings, but they don't usually have the line ending flip-flopping issue like `.meta` files, so we did not hard-code it.

As for the source code files that are edited via text editors, those file should apply `text` attribute to override "* -text" to ensure consistent line endings for these files in the repository, to prevent interference with readability of Git commands like `git blame` due to line ending fluctuations.

Previously, the setting was set to `autocrlf`, but in the updated versions, we have switched to using LF uniformly.

```
*.cs text eol=lf
```

Since we are trying to calculate a custom source hash for AssetBundles, which includes all the source materials of the AssetBundle as well as all the source code files as hashing input. Source code is included because the code may have AssetImporter that can affect the import result. To ensure that the source hash calculation results are consistent between Windows and Mac, we now uniformly force LF line endings to make the binary representation of source files the same on both platforms. Most code editors on Windows can still function properly with LF line endings, so the impact is minimal.

P.S. (Modifications to asmdef can affect the Assembly names and prevent AssetBundle from dereferencing MonoBehaviour attached to GameObject. If you want to control whether AssetBundle updates based on custom source hash, you need to track asmdef changes as well.)

To use Windows CRLF and Mac LF line endings for source files like OS default, you can apply the following setting instead:

```
*.cs text=auto
```

Finally, you can refer to the complete `.gitattributes` settings for the entire setup: https://github.com/FrankNine/RepoConfig/blob/master/.gitattributes

## EditorConfig

https://editorconfig.org/

This is not directly related to Git itself, but it's a configuration file that can be included in the repository, so I'll mention it here as well. Currently, we are using a very barebone configuration:

Code

```
root = true

[*]
charset = utf-8

indent_style = space
indent_size  = 4

trim_trailing_whitespace = true
```

`.gitattributes` controls line endings, but source files may still have issues with text encoding, BOM, or inconsistent Tab/Space. Encoding and Tab/Space can be managed using a `.editorconfig` file placed within the project to provide guidance to the source editors on how to handle them. Currently, Visual Studio,

MonoDevelop, and Rider automatically detect settings from `.editorconfig`, while Visual Studio Code and Notepad++ require plugins to be installed.

Furthermore, Visual Studio or ReSharper settings can also be committed to the repository to achieve consistency in naming or coding conventions among the entire team. However, our company does not have practical experience in this practice as there is no unified adoption of JetBrains editors yet.

## Worktree

If you're developing multi-platform games, you may find it convenient to clone the same project multiple times and configure each clone to a different platform, such as one clone for Android and another for iOS. Although Cache Server / Accelerator or AssetDatabase V2 can reduce the cost of switching platforms, keeping multiple copies of the project on your hard drive is still the fastest and most intuitive way. The downside is that it consumes a lot of disk space. However, you can actually clone the project `.git` folder once and then use multiple Worktree, which means you'll still have multiple project directories, but only one `.git` folder, resulting in only one Git history on your computer. Also if you have concerns about disk space and your computer has a large traditional hard drive but limited SSD space, you can consider storing the Git history on the traditional hard drive and placing the project Worktrees on the SSD.

To do this in the Git root directory:

Code

```
git worktree add ../project-ios
```

It will create an additional Worktree named `project-ios` in the parent directory:

Extra Worktrees can be recognized by most regular Git clients, and the operations are similar to a regular clone.

To list the existing Worktrees, use the following command:

Code

```
git worktree list
```

When you no longer need a Worktree, you can simply delete it and then run:

Code

```
git worktree prune
```

## Hooks

Git allows you to add automatic triggering Shell Scripts in the settings for auto integrity checking or cleaning, it's commonly used for clearing empty directories and their `.meta` files, as well as checking if corresponding `.meta` files are added when committing new files. However, I haven't personally promoted this practice to the entire team, so I don't have much personal experience to share. You can refer to the following resources:

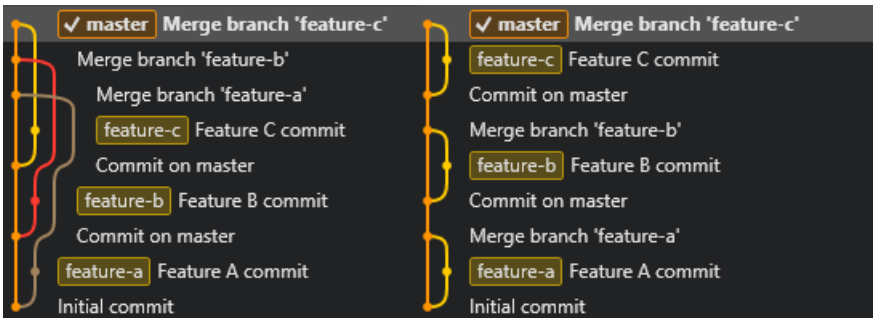
- pre-commit
- doitian/unity-git-hooks (for cleaning empty directories and checking .meta files)

If given the opportunity, I would prefer to implement this using a language like Python, as the logic would be easier to understand. Additionally, empty directory `.meta` files usually do not cause problems, so I don't feel pressured to use hooks to check for them (Missing `.meta` files in commits are also rare after promoting awareness to the team, but automated checks are always helpful). However, one situation to note is the `.framework` and `.bundle` folders in iOS and Mac environments, which contain plugins and have Import Settings stored in `.meta` files. If the `.meta` files of these directories are not properly removed

when `.framework` or `.bundle` folders are deleted, Xcode may attempt to import them as empty plugins and fail to build.

## Rebase

I am used to the process of forked repository, fetch Upstream, push Origin, and open Merge Request, as well as add other colleagues' forks as other remotes. I am also used to using Rebase or Cherrypick to modify Git branch history. If someone merges to the main branch before I do, I will Rebase my own branch to the HEAD of main branch and then open a new Merge Request with `--no-ff` (no fast-forward to create a non-overlapping branch merge in the history, which makes it easy to trace in the future. Another situation is when a feature changes X to Y, but later it turns out that it was a mistake and it should be changed to Z instead. In such cases, I will try to modify the history to be X -> Z before submitting the Merge Request, instead of X -> Y -> Z, to reduce the burden on reviewers. I feel that these operations could be included in the Code Review standards, but currently, I only do these operations myself due to time constraints and I don't have experience promoting to others.




The difference in history between using Merge and Rebase followed by a merge with the `--no-ff` flag. Rebase is more understandable shown on the right side (history shown is for demonstration purposes only, actual work situations may be more complex).

Reference:

- A tidy, linear Git history

For non-technical users, performing a rebase instead of a merge may be more challenging, but if everyone uses default pull, it may result in a large number of overlapping merge commits. One approach is to set up pull.rebase and rebase.autoStash for non-technical users.

Code

```
git config pull.rebase true
git config rebase.autoStash true
```

This approach triggers stash, rebase, and apply stash during pull, replacing merge operation during pull, which can be effective in reducing overlaps. However, the downside is that if conflicts arise, rebase conflicts are harder to deal with than merge conflicts. Whether to apply this approach or not depends on the level of acceptance among non-technical team members.

Rebasing forward will let feature branches get the new commits on main branches. While another approach is to merge the main branch into the feature branch. However, with this approach, it is important to communicate to all team members to be careful with resolving conflicts during merging. If merge conflict resolutions are done incorrectly during merging main branch to feature branch, there will be no warning when merging feature branches back to the main branch, as conflicts have already been marked as resolved in earlier merges. Non-technical team members may occasionally make this mistake, but such mistake can have serious consequences as it can pollute the main branch. Extra caution is needed if you choose not to use rebase to get new changes.

## Sparse checkout & Shallow clone

During checkout, it is not necessary to checkout the entire repository. Instead, you can specify file or folder patterns in `.git/info/sparse-checkout` to only checkout specific files or folders. The following is an example of cloning and checking out only `/Assets/` and `/ProjectSettings/` folders.

Code

```
git clone --no-checkout <Repo URI>
cd <Repo path>
git config core.sparseCheckout true
(echo /Assets/) > .git/info/sparse-checkout
(echo /ProjectSettings/) >> .git/info/sparse-checkout # >> means append
git checkout
```

Git 2.25.0 introduced `git sparse-checkout`, which allows you to use `git sparse-checkout init` instead of `git config core.sparseCheckout true` `git sparse-checkout set/add` replacing directly editing `.git/info/sparse-checkout`.

Additionally, `git clone` can be used with the `--depth` parameter to limit the depth of the cloned history.

Code

```
git clone --depth 5 -b <Branch> <Repo URI>
```

If you want to restore, execute:

Code

```
git fetch --unshallow
```

Sparse Checkout and Shallow Clone can both speed up Git operations and reduce disk space usage, but they are not commonly used on clones for authoring. The more practical application is to use them on Build Pipelines, where you can control the behavior of Clone and Checkout on the Build Server to achieve faster build times as you only need HEAD to build the project.

Git 2.34 introduced a more aggressive Sparse Index feature, which allows you to even sparse checkout the Git index. It's mainly for a Monorepo-style project. However, we are not currently using Monorepo, and I personally have some doubts about the workflow of removing things to return to a non-Monorepo state after using Monorepo. Let's not delve into this discussion for now.

## Branching Model

After using it for a long time, the once popular Git Flow has been abandoned by us, as the benefits it brings are not outweighing the difficulties caused by its complexity (Hacker News discussion). This is especially challenging to implement in game development teams where not everyone has a technical background to be able to jump back and forth between branches.

In addition, Git Flow does not emphasize that the lifecycle of Feature Branches should be short and should be merged into the main branch frequently. If you have long Feature Branches, it means the project is not integrated. **Creating long-lived Feature Branches does not provide you with an isolated working environment; it only gives the illusion of working independently**. The real effect is delaying the cost of integration, but when it's time to integrate, you pay a compounded cost. If you have experienced situations where several people want to merge into the main branch right before the end of a sprint or milestone, resulting in conflicts and unstable code on the main branch that cannot be fixed due to time constraints, then you understand what I mean.

We are now transitioning to Trunk-Based Development, where short Feature Branches are quickly merged into the main branch, or long Feature Branches are merged into the main branch periodically using Branch by Abstraction and Feature Toggles, allowing unfinished features to be merged without affecting others, thus spreading the cost of integration. If you are running Scrum, sometimes you will encounter situations where features are not fully completed by the end of a sprint. If these features are all kept in Feature Branches that haven't been merged for a long time, it can be very troublesome to carry them over to the

next sprint. This creates a lot of pressure to integrate them by the end of the sprint. However. If the features are controlled by Feature Toggles, they can be merged in smaller increments, and then re-evaluated and planned for the next sprint. After the features are completed, the Feature Toggles are typically removed, but they can also be kept in place for A/B testing or remote configuration control purposes.

In addition, Code Review is more suitable for environments where frequent merging via Merge Requests is practiced. In general, the amount of code that one can effectively review at a time is limited. Reviewing changes that have been in development for more than a week can be very challenging, and beyond that, it becomes a torture for the reviewer.

## LFS Lock

In git LFS v2.0.0, a new feature was added to lock specific files, and the Fork client also supports LFS Lock from the GUI. To declare a file as lockable, add `lockable` in the `.gitattributes` file.(In fact, it seems that you can `git lfs lock` on files without the `lockable` attribute. The only difference seems that if a file is marked `lockable` on commit, git LFS will automatically add the read-only attribute to files if it hasn't been `git lfs lock` yet.)

Code

```
*.jpg filter=lfs diff=lfs merge=lfs -text lockable
```

After that, you can use `lock` and `unlock` commands.

Code

```
git lfs lock <file path>
git lfs unlock <file path>
```

Although this feature feels like it could implement a file locking mechanism similar to checkout in Perforce, currently it still requires manual checking of lock status, which is not very convenient. Therefore, it has not been widely adopted in projects, but I believe it has the potential to be a promising direction. ~~Hopefully, future versions will improve or provide methods, such as hooks, to automatically lock files when detecting file edits, just like Perforce.~~

Later, I tried automating the process of setting files as read-only and then releasing the read-only flag only after LFS Lock, mimicking Perforce's behavior. However, I realized that Unity Editor itself does not respect the system's read-only flag, as Unity just removes read-only flags on files in Assets directory. Unable to distinguish whether the read-only flag was removed by the hook or by Unity. This forced me to abandon the idea and left me curious about how Perforce could be working with Unity, given Unity does not respect the read-only flag.

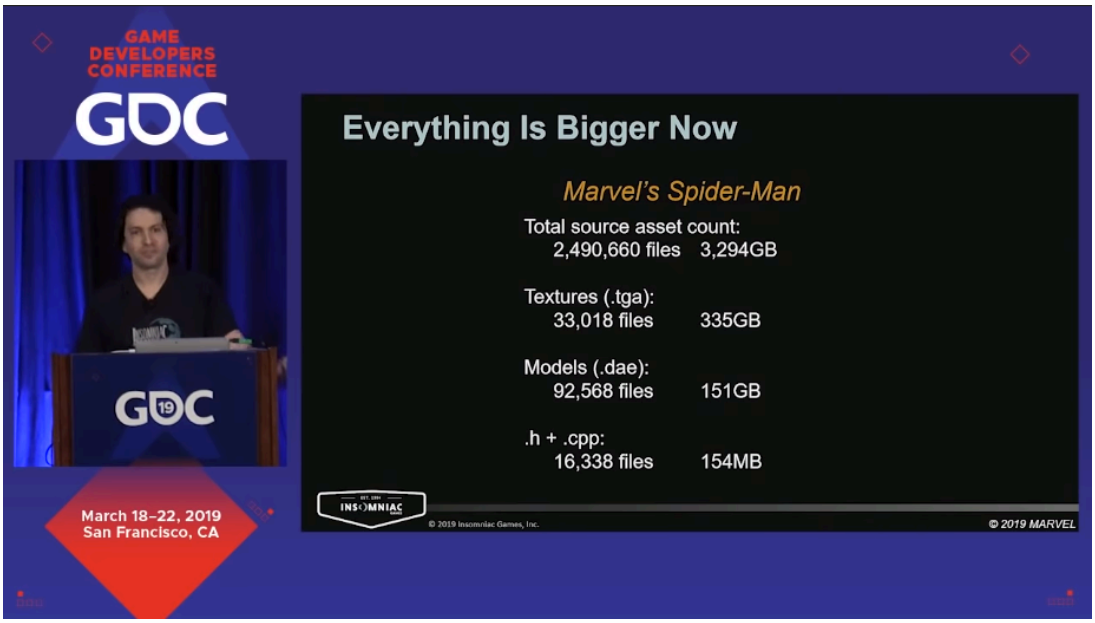
In addition, it seems that the improvements users expected, such as Per-branch Lock and Auto Unlock, have been shelved for a long time. It seems unnecessary to invest further in researching LFS Lock until these major issues are resolved.

## VFS for Git

VFS for Git is a solution proposed by Microsoft for managing the large and old Windows repository using Git. It relies on the underlying ProjFS virtual file system. When using `gvfs clone`, you may see files exist on the disk but without content, and attempting to open and read the files will trigger blocking and ProjFS will start to download the content from the server. Testing `gvfs clone` was indeed very fast as Microsoft claimed, completing under one minute. The actual contents are automatically downloaded when opening the Unity project, as it intervenes at the file system level, and requires no modification for Unity 2020 and later versions to work. However, Unity Editor still scans the entire `/Assets/` folder to build the AssetDatabase every time it is executed, which means that the entire `/Assets/` folder is still processed and downloaded. To fully enjoy the benefits of only downloading files that are needed for editing, the engine needs to have a system that only touches files when they are needed.

> With VFS for Git, developers using Git and Azure DevOps can stay productive, even in mammoth repositories like the Windows operating system, which clocks in at roughly 300 GB (3.5 million files). It's the largest Git repository in the world.

Even though Microsoft claims that the Windows repository is very large.




To be honest, compared to game repositories, it still doesn't feel very large.

Although it looks impressive, it is still not feasible to consider adopting it at this stage. The main reason is that there are very few supported Git servers. Only Azure Repos is supported, and it has a 250GB size limit which might make it impractical for larger projects. GitHub and GitLab do not seem to be actively interested in supporting it.

For those who are still interested in trying, there are several points to note when implementing VFS for Git:

- Only Azure Repos is supported.
- The Git repository must be in a non-LFS state. If it is already using LFS, you need to reverse it using `git lfs migrate export` to restore the state where binary files are directly committed in Git history.
- The contents of `.gitattributes` file must be removed from history using rebase, leaving only `* -text`. VFS for Git does not allow any .gitattributes conversion.
- You must use the Forked Git from Microsoft. Microsoft claims that even without using VFS for Git, this Fork can provide some acceleration for general Git operations. However, I encountered some issues with uncertain reasons and ended up going back to using the version built into the GUI client after experimenting with VFS for Git.

## Single Source of Truth

Because we used Mercurial to manage our project before Git, there was a period of time during the transition to Git where developers switched to Git while artists continued to work with Mercurial. Syncing changes between the two became extremely difficult, as we had to constantly find the synchronization points then jump back and forth to trace the history. It was especially challenging when we needed to release hotfixes urgently. Eventually, we couldn't bear the complexity and decided to enforce the use of Git for all team members. That's the reason why surveying clients for non-technical users came up earlier.

This experience made me realize that the data that makes up a game should be stored in the same history, i.e., the same repository. This has discouraged me from considering tools like Git Fusion or git-svn for syncing across version control systems. Instead, we now use Git exclusively and try to address pain points for non-technical users. At the same time, we have gradually reduced the usage of git submodule, as synchronizing submodules with the main repository's history can easily lead to errors. Shared libraries are

now managed in a more traditional way, where we tag and version them, create installation packages, and have each project commit the library content directly into their own repository.

Continuing with the idea that all the data that makes up a game should be stored in the same history, the source code of the server and the game data tables should also be placed in the same repository as the client project (Or even written in the same language: MagicOnion). When changes to the game data format are needed, a single commit would include synchronized updates for the server, client, and data tables (using a cross-language definition like ProtoBuf), eliminating issues caused by inconsistencies in server, client, and data schema. However, the challenge lies in the lack of readily available editing tools for the game data. Designers still have to edit the data in Google Sheets and download it, so this idea remains in the realm of my own imagination.

For discussions on game data editing tools, please refer to: [link reference]

- Game Development Needs Data Pipeline Middleware

## Why Git?

In fact, enabling LFS or VFS for Git removes the decentralized nature of Git, and using Thunk-based approaches reduces the need for branch capabilities of Git. We were also trying to enable Lock on top of LFS. Many people ask why not simply use SVN, Perforce, or switch to Plastic as some developers have mentioned while I was drafting this article. I have thought about it but have not been able to come up with a good answer. It seems that Git was not designed with consideration for common use cases in game development, such as large files and unmergeable binary files. Instead, LFS is used as a plugin to address these issues within the Smudge/Clean system. This article largely deals with the challenges that arise from this plugin-based approach. When our company migrated from Mercurial to Git, it was primarily because we felt that Mercurial had less development momentum compared to Git. Mercurial lacked suitable tools for Merge Request and Code Review at that time, while Git had Gerrit, GitHub, and later GitLab, which provided more options for code collaboration. Git also had a more dynamic and innovative client ecosystem compared to the limited options available for Mercurial. So, is this a case of Worse is Better as described in “Worse is Better”? Perhaps it is (not to say that Mercurial is inherently better, as I honestly do not know what “better” means in this context).

I believe that regardless of the reasons why you chose to use Git for version control in your Unity project, I can at least offer some experience and assistance here. I do not think that Git is the only option, and if you have a preferred alternative, that’s great too! Feel free to share your experiences. If you have no opinion about which to use, it may be beneficial to use the one with more resources and larger community where more users can provide insights and help with troubleshooting.

I hope this article can help those who have struggled with Git to spend less time wrestling with it and more time developing games. Goodbye for now, and hope to see you again.

## Acknowledgements

I’d like to thank following people for reviewing and providing suggestions for this article draft: 包子, 小善學長, 小金學長, Hugo, 建豪, 蒼時, 頭皮, Colin, 于修, Denny, JohnSu, Recca, Jonas and all the other friends. Feel free to leave comments if you have any additional points or discussions to add.