UNITY FOR GAMES

E-BOOK

VERSION CONTROL AND PROJECT ORGANIZATION BEST PRACTICE GUIDE

2020 LTS EDITION

Unity®

Contents

Introduction

Software development becomes a different beast when you move from working on your own to with a team. Where do you store the project so that every team member has access to it? What happens if more than one person works on the same file at the same time? Programmers often understand the concepts behind source control, but what about artists and other non-technical team members? How can you minimize the amount of support they need from programmers, so they don't have to worry about doing something wrong?

Source control, or version control, can be a daunting topic for game developers and creators, especially if you're not from a technical background. But it doesn't need to be that way. There are a number of tools that integrate with Unity to help your team work effectively with versioning.

This guide explains the key concepts of version control and compares some of the different version control systems (VCS) available. It provides tips and tricks you can use when setting up your Unity project to help ensure team collaboration is smooth and efficient. Finally, you'll pick up some version control best practices for working successfully in a team.

Source control vs version control

In the beginning of computing, all software development was pure code. Even as 3D graphics evolved, everything was still described as code. As such, the term “source control” was used to describe the systems in place to manage the project's contents, while the term source code management, or SCM, was given as a label for those tools.

Moving into the modern era of software and game development, we now work with a lot more than just the source code. 3D model formats, such as FBX, textures, materials, audio files, and more, mean that SCMs now have to handle more than just text file changes. The term “source control” no longer covers what we need, and thus “version control system” or VCS, became a more apt description and is now the common label given to the tools used.

The terms can still be used interchangeably. However, when talking about Unity projects that often deal with large binary assets, version control and VCS are most accurate, so that’s how they’ll be referred to throughout the rest of the guide.

Three of the main version control systems that work best with Unity are Plastic SCM, Git, and Perforce Helix Core. This guide presents the benefits and shortcomings of these systems when working as a team on a Unity project.

[1] Plastic SCM joined the Unity family in 2020, which means that these tools are closely integrated into the Unity Editor.

Foundational concepts

This section covers some of the core concepts of version control. If you don't know your commit from your push, read this section to learn about the core concepts and terminology of version control.

How version control works

Version control allows you to keep a historical record of your entire project. It brings organization to your work and enables teams to iterate efficiently. But how?

Project files are stored in a shared database called a repository, or “repo.” You backup your project at regular intervals to the repo, and if something goes wrong, you can revert back to an earlier version of the project.

With a VCS, you can make multiple individual changes and “commit” them as a single group for versioning. This commit sits as a point on the timeline of your project, so that if you need to revert back to a previous version, everything from that commit is undone. You can review and modify each change grouped within a commit or undo the commit entirely.

With access to the project’s entire history, it’s easier to identify which changes introduced bugs, restore previously removed features, and easily document changes between your game or product releases.

What’s more, because version control is typically stored in the cloud or on a distributed server, it supports your development team’s collaboration from wherever they’re working – an increasingly important benefit as remote work becomes commonplace.

Why use version control?

Aside from the reasons mentioned above, version control is useful for making experimental changes. You can add a new feature in your local version of the project, and if things don’t work out, you just revert your changes to go back to working on a clean, functional version of the project.

You can iterate on experimental ideas, and if you need to help out on a major issue in the main project, version control allows you to save your changes for a later date. Then you can get your local version back to the main branch to help out with whatever needs to be worked on. Once you’re done, you can restore and carry on with the experimental work.

Most version control systems prevent you from accidentally overwriting work that someone else in your team has done. As you commit your work to the repository, you will also need to “pull” the latest updates from the repository. This allows you to check that someone else hasn’t been working on the same file as you. This is the dreaded “merge conflict,” one of the things that can be scary to people who are not used to version control. However, merge conflicts can usually be resolved easily once you understand the tools.

Centralized vs distributed version control

For the most part, version control systems fall into one of two categories: centralized or distributed. Depending on which kind of version control system you work with, some of the terms outlined below will apply, some won't, and some may even have a different meaning. Let's take a look at the differences between these two categories.

Centralized

The first key difference between centralized and distributed systems is where the repo resides. Many companies choose the centralized option to keep the servers hosting their proprietary software on-site. Source control security is often an important factor in choosing this kind of system. A centralized system doesn't have to mean on-site servers since the repo can still be hosted in the cloud, but this setup is less common than in distributed systems.

The other key difference between the two approaches is how users deploy their changes to the repo. Centralized version control is often seen as the more straightforward option. When working with a centralized repo, changes are fetched from and sent to the repository directly. This process is called updating from and committing to the repo.

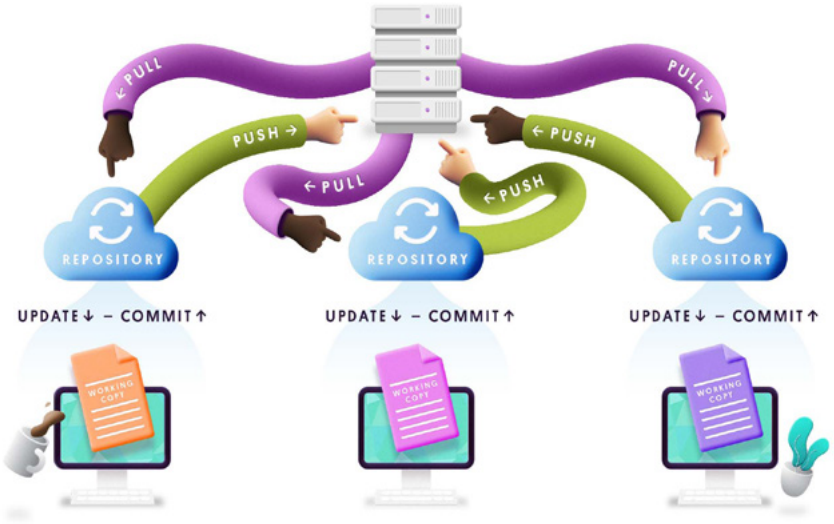
The downside to this setup is that users must be connected to the server to submit any work. To avoid conflicts, users can lock files for modification. This is known as checking out the file, and it prevents anyone else from committing changes until the file is checked back in.

In a centralized workflow, a user only ever has the latest version of the project files on their workstation, and the server holds the project's entire history.

Distributed

In a distributed workflow, there is still a single location where the repo lives, usually on a cloud service such as GitHub, but users clone the entire project history to their workstation. This allows users to work on their own local copy and commit changes quickly since they don't need to be connected to a central server. To send those changes so others can access them later, the user needs to push them to the server and pull any other changes down. However, they don't need to be always working with the latest files like on a centralized system.

Working this way allows you to create a group of changesets that perhaps equate to a larger feature before pushing them up for the rest of your team. In fact, it's encouraged to commit little and often, but we will get to those best practices later on.



File locking is still available in some distributed workflows, however, it's less common since you can handle merges more easily. By pulling the latest changes from the server to your local project, you can compare anyone else's changes to your own to be sure there are no conflicts before pushing your changes to the repo.

While the distributed approach is often preferred, it also has a few disadvantages. Firstly, having the entire project history on local machines takes up a lot of space, especially for teams working with binary file types. Git has an option called Large File Storage (LFS), which converts the history of certain files to text pointers, offloading some of the weight. However, other files have the entire history, and repos can end up with a load of old or stale test data. Studios working with small M2 drives may then find the size of the repo gets bloated with old versions, overloading their drives.

Secondly, as developers don't have to stay in contact with a central server, they can end up working in isolation for long periods. Their local version can become quite detached from the main repository, and when it comes time to merge their changes back in, this may be more work than they bargained for.

Throughout this book, we'll focus on three main version control systems, and it's worth keeping in mind which workflow each supports:

— Git – distributed

— Perforce – centralized

— Plastic SCM – both

Typical workflow

Centralized

1. Update your working copy with changes from the server
2. Make your changes
3. Commit your changes to the central server

Distributed

1. Pull any remote changes into your local repo
2. Make changes
3. Commit changes
4. Perform steps 2 and 3 as many times as you like
5. Push all commits back to the remote repo

Key terms

Term	Explanation
Working copy	Your local version of the project. Sometimes also called a checkout or workspace. You make changes to your working copy, and, when you're happy with them, commit them to the repository.
Commit/check in	A commit encodes file modifications. A centralized workflow sends those changes to the server and is more commonly called checking in. In a distributed workflow, it adds them to the changeset that needs to later be pushed to the server.
Pull/update/ check out	Pulling or updating retrieves the latest changes available on the server. Check out is the more common term when working in a centralized workflow.
Locking	Locking a file prevents it from being edited by another user. You are telling the server, "I'm working on this; please don't make any other changes." Locking is generally not supported in distributed workflows.
Clone	In a distributed workflow, cloning a repo is how you initially get a copy of the project and its entire history onto your local machine.
Tags	Tags are special notes that can be added to a commit. They are often used to mark a point in time where a build was made.
Branch	A branch creates a new copy of the codeline, which can then be worked on in parallel. This allows someone to work on parts of the project in isolation, for example a new feature, without affecting the main line of development.
Merge	Merging can happen either when a branch is finished and needs to be merged back into the main line, or even just when two people make changes around the same time. The two changesets will need to be compared and merged together to create the new working copy. Most merges can be handled automatically.

Term	Explanation
Conflict	A conflict is what happens when merges cannot be handled automatically. This usually results from two people making changes to the same lines of code or the same binary file. Code conflicts can usually be resolved by comparing the text and working out which changes should be accepted, or even whether both can be brought together in a way. For binary files, such as Unity scenes or Prefabs, merging a conflict becomes a lot trickier. However, sometimes a quick conversation with the other contributor is the easiest way to resolve what changes make the most sense to keep.
Pull request	When work on a branch is complete, it's good practice to open a pull request. This signals to the rest of your team that work on that branch is complete and ready to be merged back into the main line. This system gives team leads and/or seniors a chance to review the changes before accepting them back into the main branch.
Head	Head refers to the latest commit on your working copy.
Reset/revert	Depending on your VCS, reset or revert can be used to discard all your local changes back to their state at the head.
Index	The Git index is a file that describes all the current changes you have in your working copy. These changes sit in what's known as the staging area, where you can select which changes you want to add to your next commit.
Git stash	If you have some changes that aren't ready yet for a commit, but you need to move onto some different work, you can use a stash to save those changes in a temporary file and reset your working copy back to head.

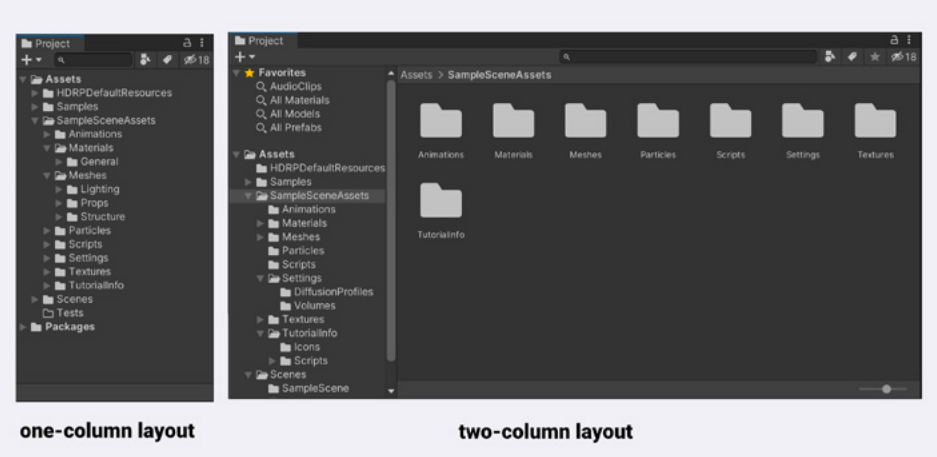
Best practices for organizing a Unity project

Regardless of which VCS you choose, there are several things to consider that will help streamline your version control workflow when working in Unity. First, let's take a look at some of the different ways your team can work together effectively.

Project organization

Folder structure

Here's what a typical Unity project folder structure looks like. Remember, in the Editor, you can switch your view in the Project window between one and two columns.



One-column and two-column Project window views

Although there is no single way to organize your project, in general, follow these recommendations.

— Document your naming conventions and folder structure. A style guide and/or project template makes your files easier to find and organize. Pick what works for your team, and make sure everyone is on board with it.

— Be consistent with your naming convention. Don't deviate from your chosen style guide or template. If you do need to amend your naming rules, parse and rename your affected assets all at once. In cases where the changes affect a large number of files, consider automating the update using a script.

— Don't use spaces in file and folder names. Unity's command line tools have issues with path names that have spaces. Use CamelCase as an alternative for spaces.

— Separate testing or sandbox areas. Create a separate folder for non-production scenes and experimentation. Subfolders with usernames can divide your work area by team member.

— Avoid extra folders at the root level. In general, store your content files within the Assets folder. Don't create additional folders at the project's root level unless it's absolutely necessary.

— Keep your internal assets separate from third-party ones. If you are using assets from the Asset Store or other plug-ins, odds are they have their own project structure. Keep your assets separate.

> ***Note:*** *If you find yourself modifying a third-party asset or plug-in for your project, then version control can really help you out when you need to get the latest update for the plug-in. Once the update is imported, you can look through the diff to find where your modifications may have been overwritten and reimplement them.*

While there is no set folder structure, here are a couple of examples of how you might set up your Unity project. These structures are based on splitting up your project by asset type. The Asset Types manual page describes the most common assets in greater detail. You can use the Template or Learn projects as an example of organizing your folder structure. While you're not limited to these folder names, they should give you a good starting point.

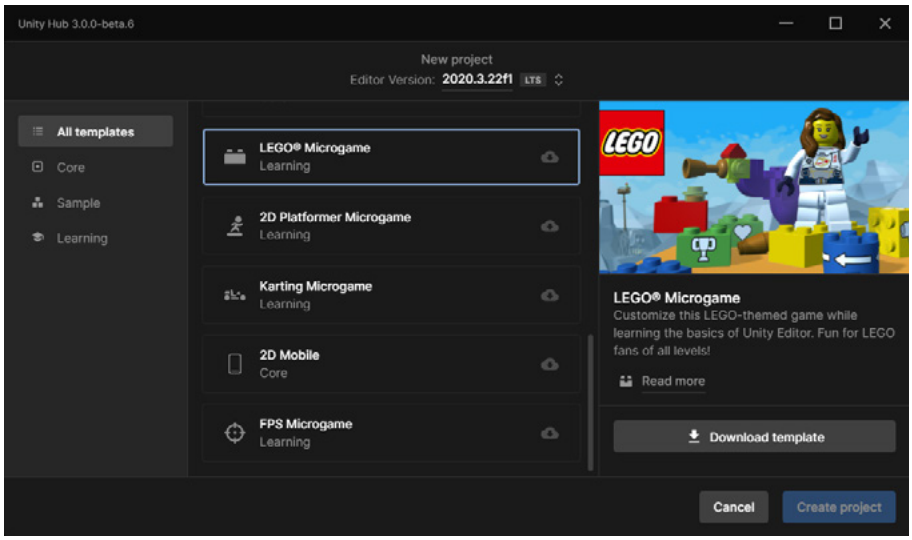
Eample 1

```
Assets
+---Art
| +---Materials
| +---Models
| +---Textures
+---Audio
| +---Music
| \---Sound
+---Code
| +---Scripts  # C# scripts
| \---Shaders  # Shader files and shader graphs
+---Docs   # Wiki, concept art, marketing material
+---Level   # Anything related to game design in Unity
| +---Prefabs
| +---Scenes
| \---UI
```

Eample 2

```
+---Art
| +---Materials
| +---Models
| +---Music
| +---Prefabs
| +---Sound
| +---Textures
| +---UI
+---Levels
+---Src
| +---Framework
| \---Shaders
```

If you download one of the Template or Starter Projects from the Unity Hub, you'll find that those projects have their subfolders split up based on asset type, as seen in the image below.

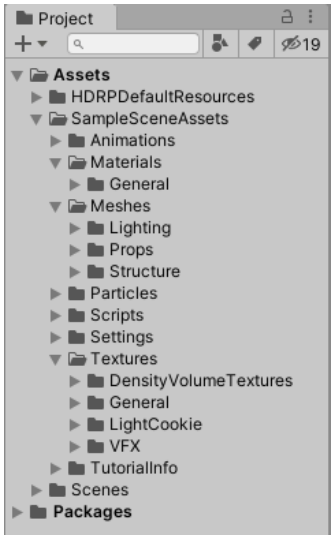


Templates available to download in the Unity Hub

Depending on which template you've chosen, you should see subfolders that represent several common assets.

Asset Type	Explanation
Animations	Animations contain animated motion clips and their controller files. These can also contain Timeline assets for in-game cinematics or rigging information for procedural animation.
Audio	Sound assets include audio clips as well as the mixers used for blending the effects and music.
Editor	This contains scripted tools made for use with the Unity Editor but not appearing in a target build.
Fonts	This folder contains the fonts used in the game.
Materials	These assets describe surface shading properties.
Meshes	Store models created in an external digital content creation (DCC) application here.
Particles	The particle simulations in Unity, created either with the Particle System or Visual Effect Graph.
Prefabs	These are reusable GameObjects with prebuilt Components. Add them to a scene to build.

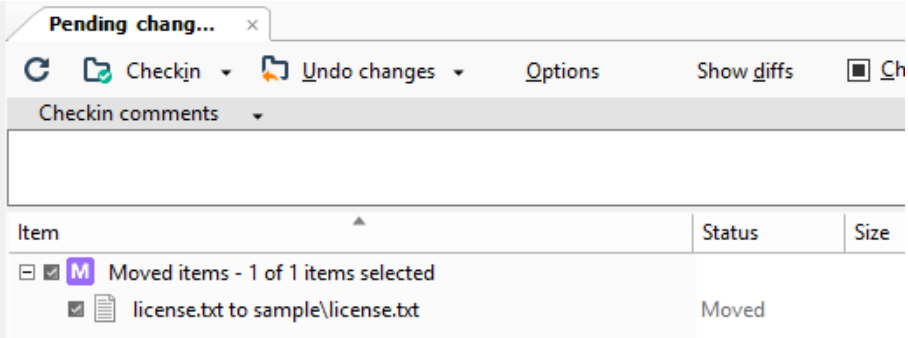
Asset Type	Explanation
Scripts	All user-developed code for gameplay appears here.
Settings	These assets store render pipeline settings, such as for the High Definition Render Pipeline (HDRP) and Universal Render Pipeline (URP).
Shaders	These programs run on the GPU as part of the graphics pipeline.
Scenes	Unity stores small, functional portions of your project into Scene assets. They often correspond to game levels or part of a level.
Textures	Image files can consist of texture files for materials and surfacing, UI overlay elements for user interface, and lightmaps to store lighting information.
ThirdParty	If you have assets from an external source like the Asset Store, keep them separated from the rest of your project here. This makes updating your third-party assets and scripts easier. Third-party assets may have a set structure that cannot be altered.



The sample scene with the HDRP template includes several asset folders.

Defining a good project structure in the beginning will avoid version control issues later. If you move assets from one folder to another, many VCS will see that as just deleting one file and adding another, rather than the file being moved. This loses the history of the original file.

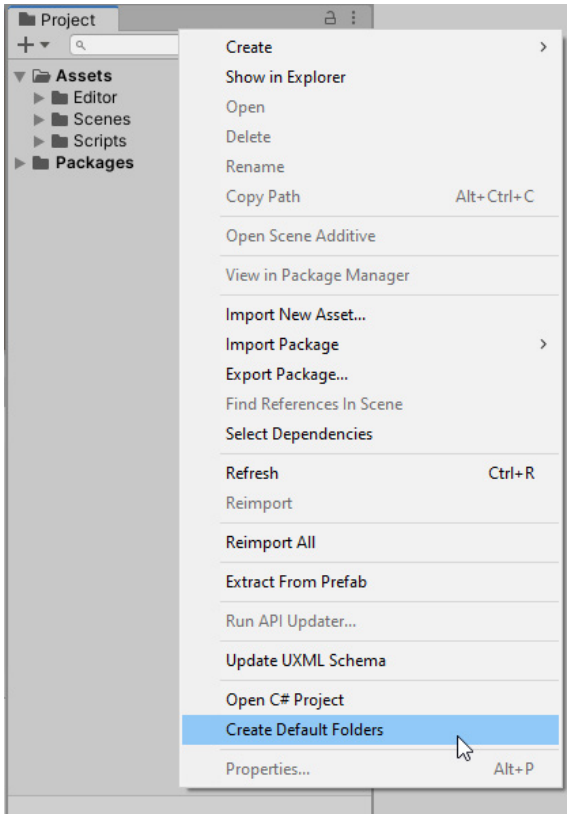
Plastic SCM can handle file moves within Unity and maintains the history of any file that's moved. However, it's essential that when you move a file, you do it in the Unity Editor so that the .meta file moves with the asset file.



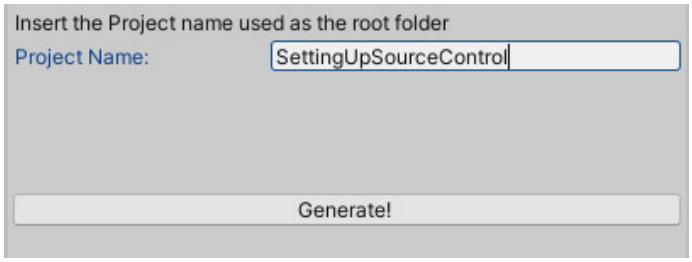
Tracking file movements

Once you've decided on a folder structure for your projects, use an Editor script to reuse the template and create the same folder structure for all projects moving forward. When it's placed in an Editor folder, the script below will create a root folder in Assets matching the "PROJECT_NAME" variable. Doing this keeps your own work separate from third-party packages.

```
using UnityEditor;
using UnityEngine;
using System.Collections.Generic;
using System.IO;

public class CreateFolders : EditorWindow {

 private static string projectName = "PROJECT_NAME";
 [MenuItem("Assets/Create Default Folders")]
 private static void SetUpFolders()
 {
  CreateFolders window = ScriptableObject.CreateInstance<CreateFolders>();
  window.position = new Rect(Screen.width/2, Screen.height/2, 400, 150);
  window.ShowPopup();
 }

 private static void CreateAllFolders()
 {
  List<string> folders = new List<string>
  {
   "Animations",
   "Audio",
   "Editor",
   "Materials",
   "Meshes",
   "Prefabs",
   "Scripts",
   "Scenes",
   "Shaders",
   "Textures",
   "UI"
  };

  foreach (string folder in folders)
  {
   if (!Directory.Exists("Assets/" + folder))
   {
    Directory.CreateDirectory("Assets/" + projectName + "/" + folder);   }  }

  List<string> uiFolders = new List<string>
  {
   "Assets",
   "Fonts",
   "Icon"
  };

  foreach (string subfolder in uiFolders)
  {
   if (!Directory.Exists("Assets/" + projectName + "/UI/" + subfolder))
   {
    Directory.CreateDirectory("Assets/" + projectName + "/UI/" + subfolder);
   }
  }

  AssetDatabase.Refresh();
 }

 void OnGUI()
 {
  EditorGUILayout.LabelField("Insert the Project name used as the root folder");
  projectName = EditorGUILayout.TextField("Project Name: ", projectName);
  this.Repaint();
  GUILayout.Space(70);
  if (GUILayout.Button("Generate!")) {
   CreateAllFolders();
   this.Close();
  }
 }
}
```



Go to menu > Assets > Create Default Folders.



Creating empty folders at the start of your project will help keep your teamwork organized and efficient.

Empty folders

Empty folders like those shown in the previous images can present a bit of an issue in version control – so only create the folders for what you need. With Git and Perforce, empty folders are ignored by default. If these project folders are set up and someone attempts to commit them, they'll be unable to until something is placed in the folder.

> ***Note:*** *A common workaround is to place a ".keep" file inside an empty folder. This is enough for the folder to then be committed to the repository.*

Plastic SCM can handle empty folders. Directories are treated as entities by Plastic SCM and have a version history associated with them.

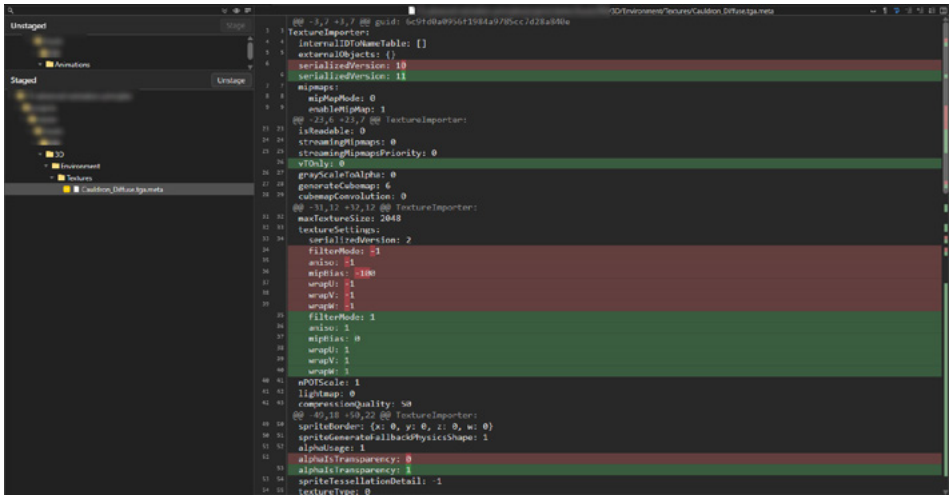
This is a point to note when working in Unity. Unity generates a .meta file for every file in the project, including folders. With Git and Perforce, a user can easily commit the .meta file for an empty folder, but the folder itself won't end up under version control. When another user gets the latest changes, there will be a .meta file for a folder that doesn't exist on their machine, and Unity will then delete the .meta file. Plastic SCM avoids this issue by including empty folders under version control.

The .meta file

Unity generates a .meta file for every other file inside the project, and while it's typically inadvisable to include auto-generated files in version control, the .meta file is a little different. Visible Meta Files mode should be turned on in the Version Control window (unless you're using the built-in Plastic SCM or Perforce modes).

Turn on Visible Meta Files when working with Git.

While the .meta file is auto-generated, it also holds a lot of information about the file with which it's associated. This is common with assets that have import settings, such as Textures, meshes, audio clips, etc. When you change any import settings on these files, the changes are written into the .meta file, not the asset file. This is why you commit the .meta files to your repository, so everyone works with the same file settings.



Changes to a .meta file when import settings were adjusted on a file

Naming standards

Agreeing on standards doesn't stop with project folder structure. Setting a naming standard for GameObjects in a scene or Prefabs inside project folders can make things easier for your team to understand when you end up working in one another's files.

Though there is no definitive naming standard for GameObjects, consider the following.

Standard	Example
Use descriptive names, and don't abbreviate. Use names that you will remember several months from now. Consider whether another person will understand your notation, and choose names that you can pronounce and remember. Be aware that abbreviations and spelling mistakes can create confusion.	largeButton, LargeButton, or leftButton NOT: lButton
Use Camel case/Pascal case. Avoid spaces in your object names. Camel case or Pascal case improve readability (and typing accuracy according to this study).	OutOfMemoryException, dateTimeFormat, NOT: Outofmemoryexception, datetimeformat
Use underscores (or hyphens) sparingly. Avoid underscores and hyphens in general. However, they can be useful in certain circumstances. Prefixing a name with an underscore puts it alphabetically first. You can also use underscores to denote variants of a specific object.	Active States: EnterButton_Active, EnterButton_Inactive Texture Maps: Foliage_Diffuse, Foliage_Normalmap Level of Detail:Building_LOD1, Building_LOD0
Use number suffixes to denote a sequence. Likewise, don't suffix with a number if it's not part of a list.	For a path, name the nodes: Node0, Node1, Node2, etc.
Follow the design document naming.	If your design document names locations like HighSpellTower or RedDragonLair, use those exact spellings.

Workflow optimization

Aside from how and where you keep your assets inside the Assets folder, there are several design and development choices you can make to help speed up your workflow, especially when you're using version control.

Split up your assets

Large, single Unity scenes do not lend themselves well to collaboration. Break your levels into many smaller scenes so that artists and designers can collaborate better on a single level while minimizing the risk of conflicts.

At runtime, your project can load scenes additively using SceneManager.LoadSceneAsync passing the LoadSceneMode.Additive parameter mode.

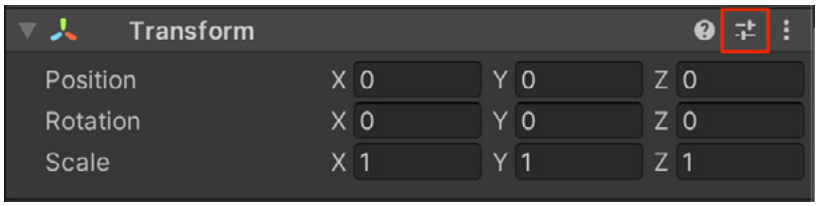
Additionally, break work up into Prefabs where possible. If you need to make changes later, you can change the Prefab rather than the scene it's used in to avoid conflicts with anyone working on the scene. Prefab changes can often be easier to read when doing a diff under version control.

And if you end up with a scene conflict, Unity also has a built-in YAML (a human-readable, data-serialization language) tool specifically for merging scenes and Prefabs. For more information, see Smart merge in the Unity documentation.

Presets

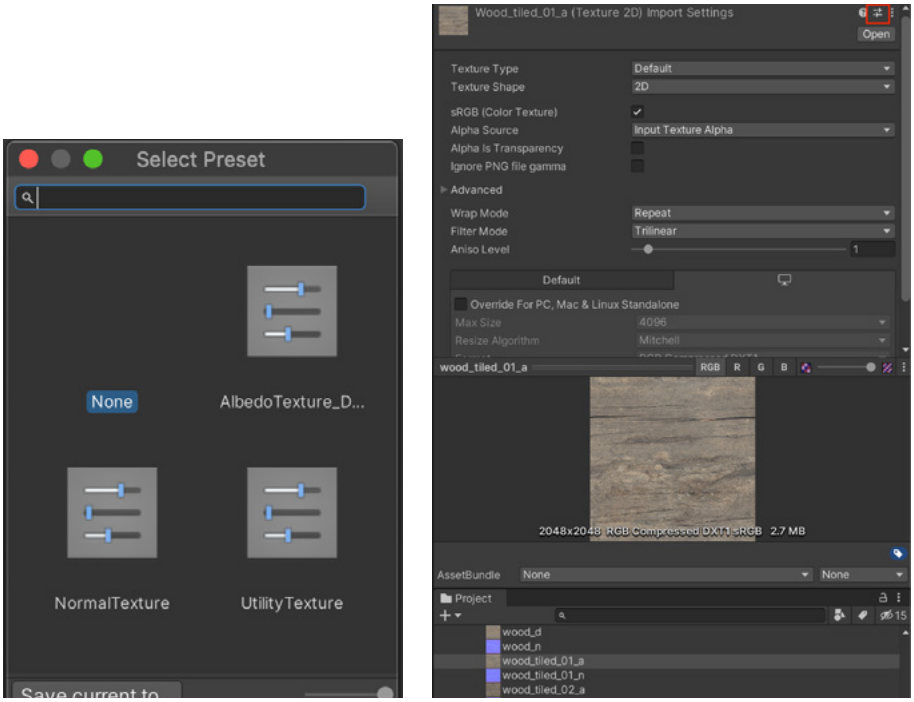
This feature allows you to customize the default state of anything in your Inspector. Creating a Preset lets you copy the settings of a component or asset, save it as an asset, then apply the same settings to another item later.

Use Presets to enforce standards or to apply reasonable defaults to new assets. This ensures consistent standards across your team, so commonly overlooked settings don't impact your project's performance.



The Preset icon is highlighted here in red.

Click the Preset icon to the top right of the component. Click Save current to... to save the Preset as an asset. Click one of the available Presets to load a set of values.



In this example, the Presets contain different Import Settings for 2D textures depending on usage (albedo, normal, or utility).

Other handy ways to use Presets include:

- **Create a GameObject with defaults**: Drag and drop a Preset asset into the Hierarchy to create a new GameObject with the corresponding component that includes Preset values.
- **Associate a specific Type with a Preset**: In the Preset Manager (Project Settings > Preset Manager), specify one or more Presets per type. Creating a new component will then default to the specified Preset values.
    - Pro tip: Create multiple Presets per type, and rely on the filter to associate the correct Preset by name.
- **Save and load manager settings:** Use Presets for a Manager window so the settings can be reused. For example, if you plan to reapply the same tags and layers or physics settings, Presets can reduce setup time for your next project.

Code standards

Coding standards will also help keep your team's work consistent and make it easier for developers to swap between different areas of your project. Again, there are no set-in-stone rules here. You need to decide what is best for your team – but once you've decided, stick with it.

As an example, namespaces can help organize your code better. They allow you to separate modules inside your project and avoid conflicts with third-party assets where class names may end up repeating.

> When using namespaces in your code, break your folder structure up by the namespace for better organization.

A standard header is also a good practice. Including a standard header in your code template will help to document the purpose of a class, the date it was created, and even who created it. All of this is information that could easily get lost in the long history of a project, even when using version control.

Unity employs a template script to read from whenever you create a new Monobehaviour in the project. Every time you create a new script or shader, Unity uses a template stored in **%EDITOR_PATH%\Data\Resources\ScriptTemplates:**

— Windows: C:\Program Files\Unity\Editor\Data\Resources\ScriptTemplates

— Mac: /Applications/Hub/Editor/[version]/Unity/Unity.app/Contents/Resources/ScriptTemplates

The default Monobehaviour template is this one:
81-C# Script-NewBehaviourScript.cs.txt

There are also templates for shaders, other behavior scripts, and assembly definitions.

For project-specific script templates, create an Assets/ScriptTemplates folder, and copy the script templates into this folder to override the defaults.

You can also modify the default script templates directly for all projects, but make sure you back up the originals before making any changes. Each version of Unity has its own template folder, so when you update to a new version, you need to replace the templates again.

The original 81-C# Script-NewBehaviourScript.cs.txt file looks like this:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

#ROOTNAMESPACEBEGIN#
public class #SCRIPTNAME# : MonoBehaviour
{
 // Start is called before the first frame update
 void Start()
 {
  #NOTRIM#
 }

 // Update is called once per frame
 void Update()
 {
  #NOTRIM#
 }
}
#ROOTNAMESPACEEND#
```

There are two keywords that may be helpful:

— **#SCRIPTNAME#** indicates the filename entered or the default filename (for example, NewBehaviourScript).

— **#NOTRIM#** ensures that the brackets contain a line of whitespace.

You can also use your own keywords and replace them with an Editor script implementing the OnWillCreateAsset method.

```
// /*--------------------------------------------
// --------------------------------------------
// Creation Date: #DATETIME#
// Author: #DEVELOPER#
// Description: #PROJECTNAME#
// --------------------------------------------
// --------------------------------------------*/

using UnityEngine;
using UnityEditor;

public class KeywordReplace : UnityEditor.AssetModificationProcessor {

public static void OnWillCreateAsset (string path)
 {
  path = path.Replace(".meta", "");
  int index = path.LastIndexOf(".");
  if (index < 0)
   return;

  string file = path.Substring(index);
  if (file != ".cs" && file != ".js" && file != ".boo")
   return;

  index = Application.dataPath.LastIndexOf("Assets");
  path = Application.dataPath.Substring(0, index) + path;
  if (!System.IO.File.Exists(path))
   return;

  string fileContent = System.IO.File.ReadAllText(path);

  fileContent = fileContent.Replace("#CREATIONDATE#", System.DateTime.Today.
ToString("dd/MM/yy") + "");
  fileContent = fileContent.Replace("#PROJECTNAME#", PlayerSettings.product-
Name);
  fileContent = fileContent.Replace("#DEVELOPER#", System.Environment.User-
Name);

  System.IO.File.WriteAllText(path, fileContent);
  AssetDatabase.Refresh();
 }
}
```

Use the header in the script above inside your script template, and any new script will be created with a header that shows its date, the user who created it, and the project to which it originally belonged. This is useful should you reuse the code in future projects.

Version control systems

Now that you're familiar with some of the key terms and concepts in version control and project organization, it's time to introduce some of the key players. Of course, no one solution is best for everyone. There are many things to consider when choosing which VCS to use in your team. Hopefully, by the end of this book, you'll have all the information you need to make that decision.

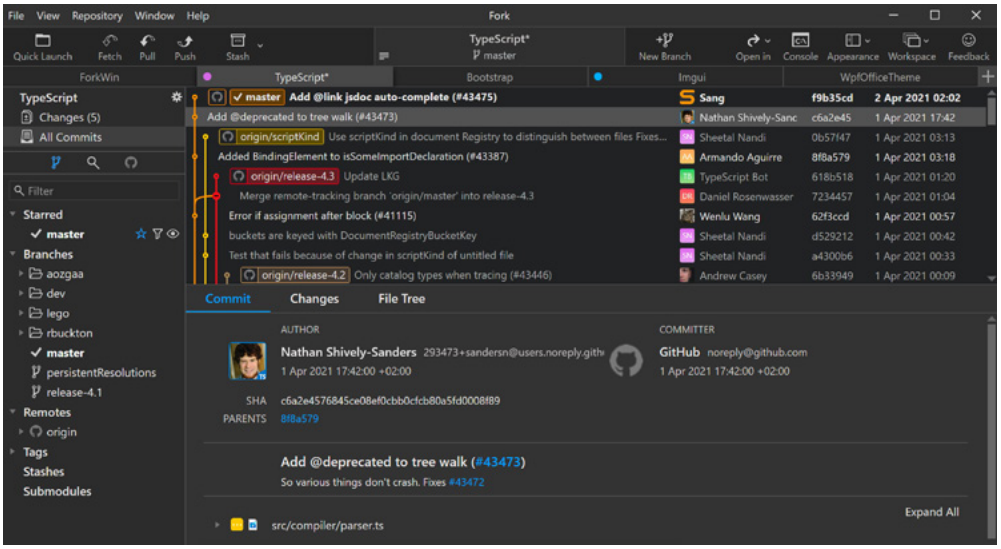
Git

Open source, free, and flexible, Git is one of the most popular version control systems around. However, as a distributed setup it can be daunting to non-technical users.

Developed in 2005 by Linus Torvalds to control the Linux kernel development, it's remained well-maintained and open source since. Git as a platform is a command line-only tool. But many different GUIs have been developed for it, making the system more accessible to users.

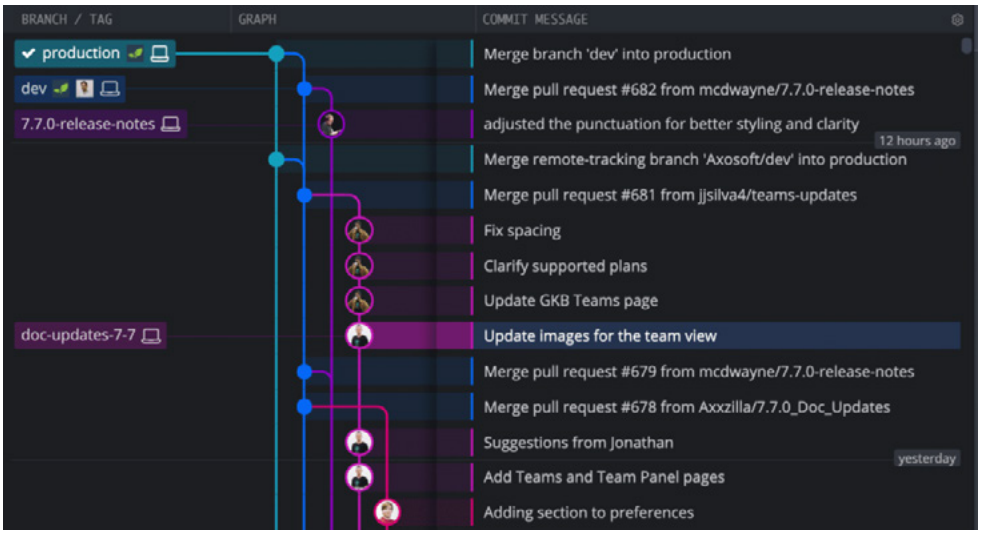
There are a few popular GIT GUI clients.

Fork: Fast and friendly. It's technically free, but occasionally asks you to pay.



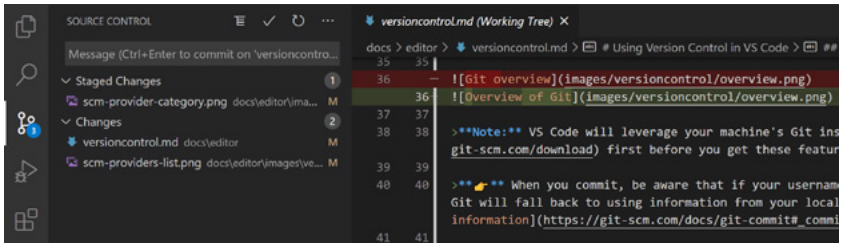
Fork

GitKraken: This offers a more visual and accessible way of working with Git with an intuitive UI as well as the flexibility to switch between a GUI or a CLI terminal.



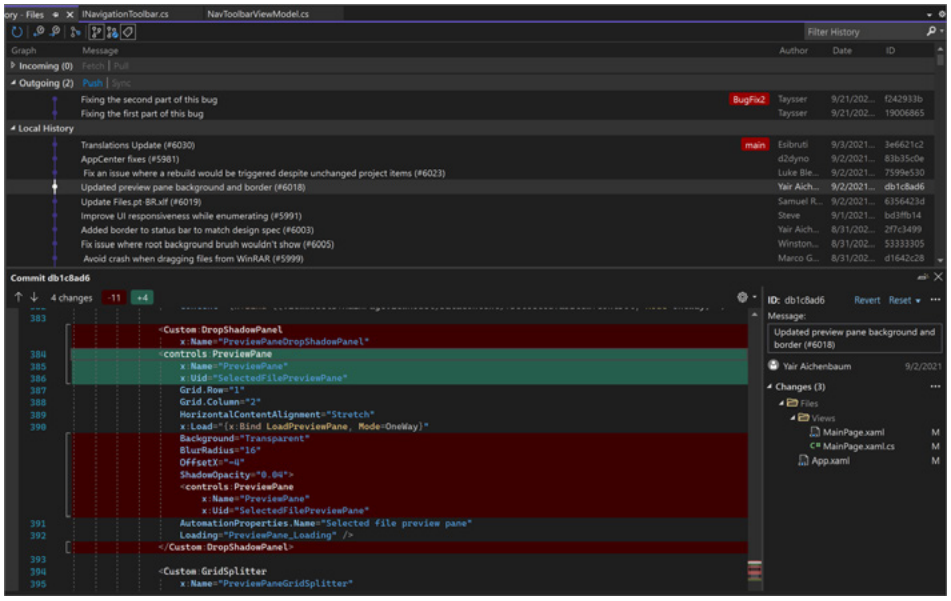
GitKraken

Visual Studio Code: VS Code has source control integration built in, and with all the extensions available, you can avoid using a separate program altogether.



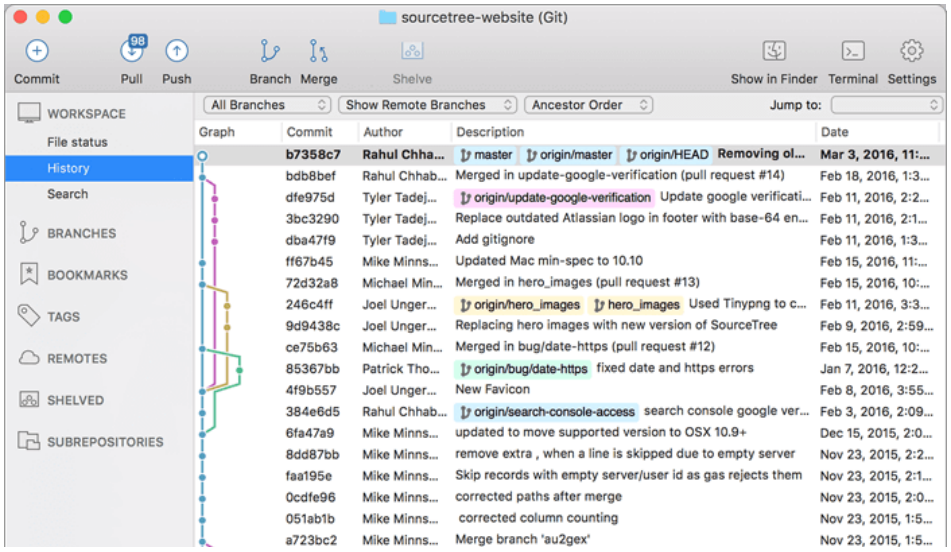
Visual Studio Code

Visual Studio: As with VS Code, Visual Studio also has Git controls built in and includes a GitHub extension.



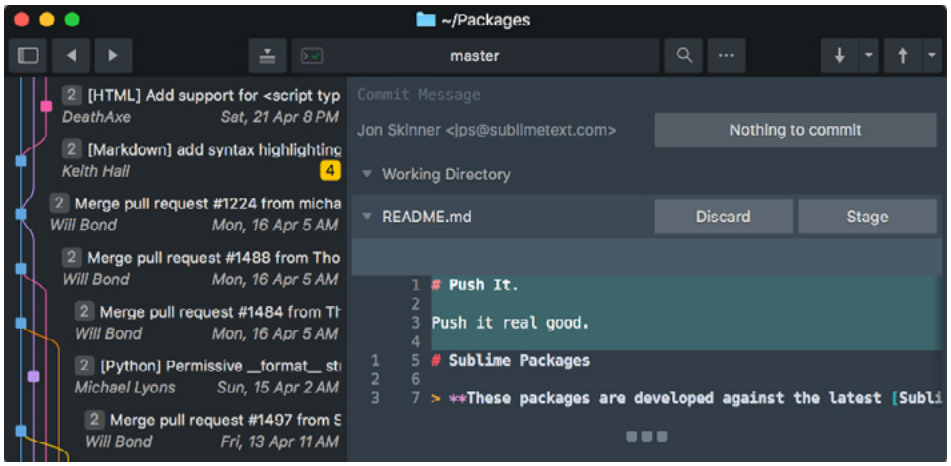
Visual Studio Code

SourceTree: Part of the Atlassian product group SourceTree is a free Git client for Windows and Mac that can also help you visualize and manage your Git repositories easily.



SourceTree

Sublime Merge: This system excels in offering tools for speeding up code reviews with side-by-side diffs and syntax highlighting. It's a lightweight, high-performance client.

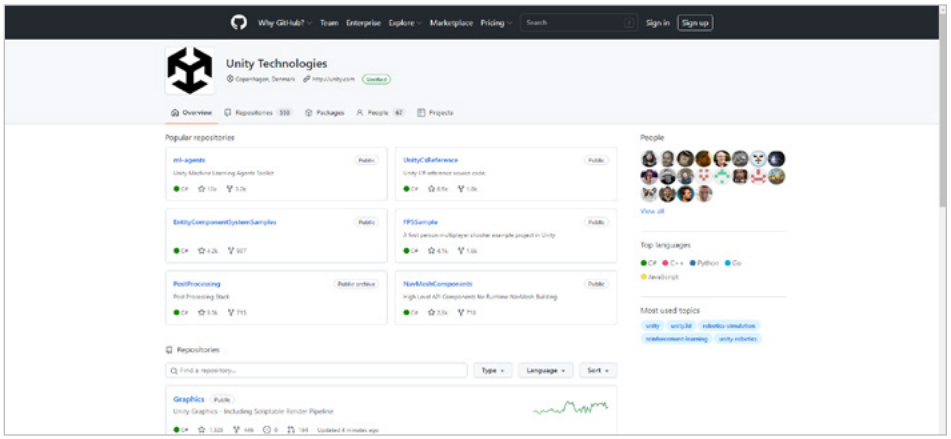


Sublime Merge

Git features strong branching and merging capabilities, but it can't handle large binary files as effectively as other solutions on the market. Git Large File Storage (LFS) goes some way to rectifying this.

Since Git is a distributed client, the entire repository and complete history is on the developer's machine. This makes actions such as switching branches or reverting back to a point in history extremely quick. If you're working on a large project with multiple features and release branches, a Git workflow can save countless hours.

There can often be some confusion between Git and GitHub. GitHub is a hosting service for Git repositories, but you can use Git without using GitHub. That said, GitHub is a very popular service because there is a free version (with some limitations), and it doesn't require any custom server setups.



GitHub

> Unity has released their C# editor and engine code to the public on GitHub. This is incredibly useful when you need to know how some functions work or how to replicate a feature of the Editor inside your own project.

GitHub also has its own Git GUI, GitHub Desktop. When working in Unity, you can also use the GitHub for Unity package to bring the Git tools directly into the Unity Editor.

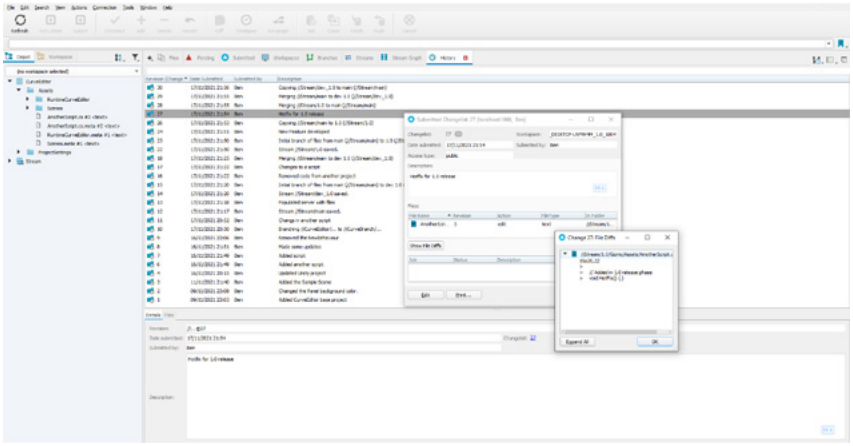
As mentioned, GitHub isn't the only hosting service available for your Git projects. You can also use Bitbucket (from Atlassian) or GitLab, which have many more DevOps features available to them, or one of the many other hosting services available.

Check out this talk from Unite Now 2020 on how to get started with Git and Unity.

Perforce (Helix Core)

Helix Core is an enterprise-level version control system used by large game studios such as Electronic Arts and Ubisoft. These studios use Perforce because it features centralized repos that are most often hosted on their own servers. It does not feature visual repos, so its adoption might be more challenging for non-technical developers, but in larger studios there will be DevOps and Release Engineers to help manage the code base. Plus, as an enterprise solution, it includes a global support team.

Helix Core can also be used by small teams. In fact, it's free for teams of up to five users and 20 workspaces. And you can still deploy to the cloud through solutions like Amazon AWS or Azure.



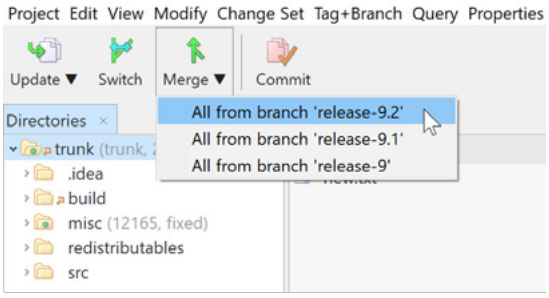
Helix Core P4V interface

Because Helix Core handles large files exceptionally well, it can be a popular option for Unity projects. There is also built-in Unity Editor integration that's covered in a later section.

To learn more about integrating your Unity workflow with Helix Core, check out this Perforce blog post.

Apache Subversion

Like Git, Apache Subversion (known as SVN) is a free and open-source version control system. Unlike Git, it's a centralized VCS that can handle large binary files. However, it's still a command line system that requires one of the many third-party GUI clients to be a bit more user friendly. One such client is SmartSVN.



SmartSVN GUI

Before Git LFS, SVN was a popular choice when working in Unity. As a centralized solution, it was simpler to work with and, as mentioned, better for working with large files. Where SVN falls behind the other tools is when you start to use branches and need to merge between them. Merging in SVN has many pains, especially when it comes to conflicts – or even false conflicts – between files. A merge operation that would take minutes in another VCS may take hours to go through manually in SVN.

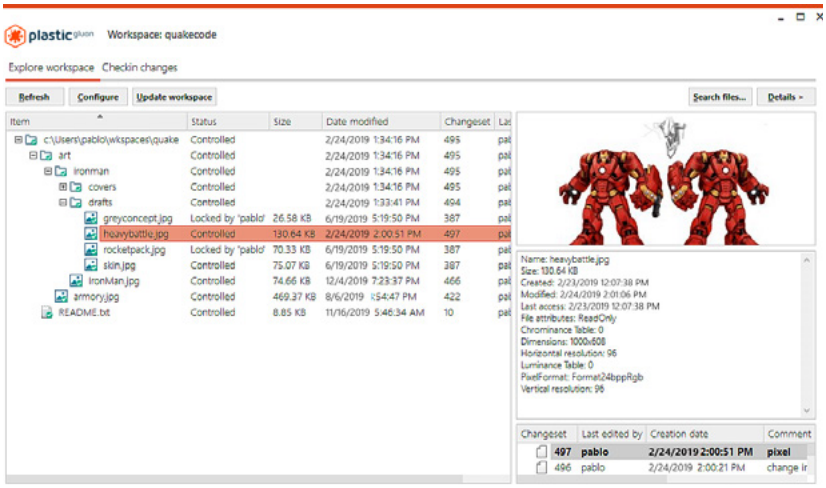
For more information on setting up Unity to work with SVN, check out the Unity documentation.

Plastic SCM

Plastic SCM is a flexible version control system that supports programmers and artists alike. It excels at handling large repos and binary files, and as both a file-based and changeset-based solution, it gives you the capability to download only the specific files you're working on, rather than the entire project build.

Plastic SCM offers hosting, the actual VCS tools, and a GUI client as part of the same solution. Small teams of up to three users can sign up for the free Cloud Edition of Plastic SCM and get up to 5GB of cloud storage, along with access to the Plastic SCM tools, including Gluon.

Gluon is a slimline client designed specifically with artists in mind. It allows you to pick only the files that you're going to work on and check them out from the server, locking them from being modified by anyone else. Once you complete your work, you check the files back in. The Gluon GUI removes the more complex concepts that work better for programmers than for other, less technical users.



Plastic offers a workflow especially designed for artists, making it easy to preview files and history as well as to check in changes.

For artists, both Plastic SCM and Gluon include ways to diff images. The image diff tool lets you compare two versions of the same file visually, a feature that many other systems don't offer.

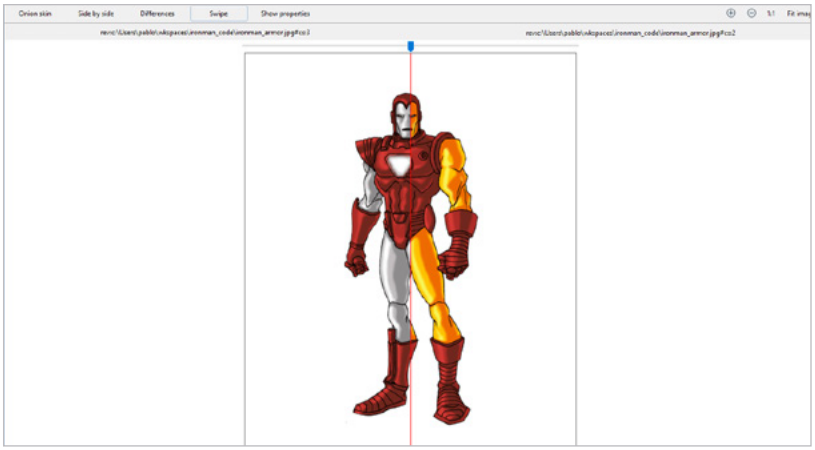
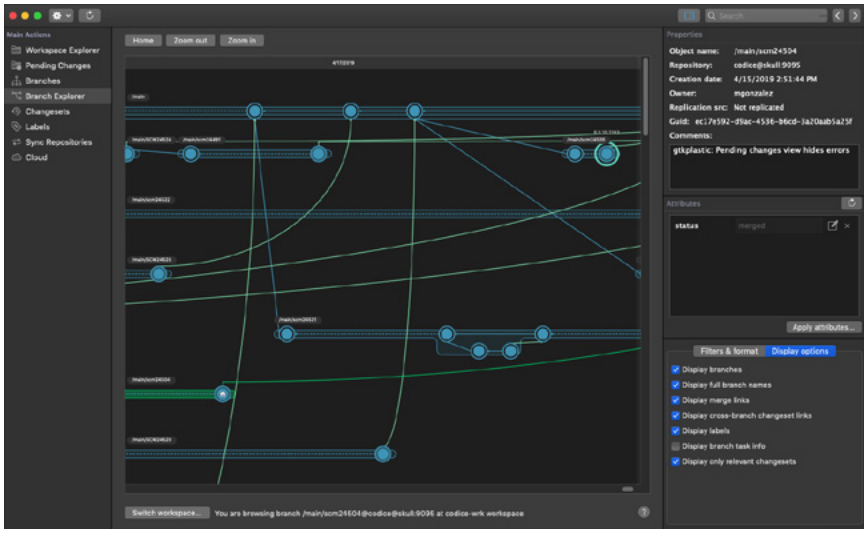


Image diff showing the Swipe mode where you can go from one version to the other by just dragging the swipe control. This is very useful to follow image evolution.



The branch explorer visualizes the merge structure of the project. It evolves horizontally from left to right.

The standard Plastic SCM GUI client has all the features they would be looking for and more for the programming team. The GUI has an interactive visual Branch Explorer that shows the true relationships of all the branches in a project. There is also a built-in Code Review system that you can use to request the review of your work from a senior developer.

Plastic SCM joined the Unity family in 2020, which means that the tools are now closely integrated into the Unity Editor.

One of the key strengths of Plastic SCM is that it has the flexibility to be configured for a distributed or centralized workflow. In fully distributed mode, developers work with a repository on their local machine, checking in, branching, and merging with ease. Developers will then push and pull changes to the server to share them when ready.

In centralized mode, users check out and check in their changes directly to the server so everyone is working on the latest changes. However, as development teams have grown into global organizations, everyone communicating with one central server isn't always beneficial. Plastic SCM can also be configured to work in a multi-site system. In this system, servers are set up at each site, so teams can check in to their local server, keeping their workflow fast and hard drives happy. Then, the distributed servers communicate with each other to a central or cloud server.

For a great video on setting up Plastic SCM with Unity, check out the Unite Now 2020 video, Version control for games with Unity's Plastic SCM by Arturo Nereu. You'll also learn more about how Plastic can be leveraged for games and more here.

Comparison ● Fully Supported ● Partially Supported

		Plastic	Git	Preforce	Subversion
Flexibility	**Good to work centralized** Just checkin, no push/pull	●		●	●
	Good to work distributed Push/pull + local repo	●	●		
Binaries	**Good with huge repos**	●		●	
	Good with huge files	●	●	●	●
	Can lock files to avoid merging	●		●	●
GUI	**Visualizes your repos** (so you don't need a PhD in branching)	●	●		
	Comes with great GUIs	●	●	●	
	Special GUI and workflow for artists and non-coders	●			
Workflow	**Creates effective task branches**	●	●		
Merge	**Very good detecting merges between branches**	●	●		
	Comes with great diff and three-way merge tools	●		●	
	Tools help you understand the merge	●			
	Good merging renames, moved files, directories, refactors	●			
Cloud	**Can host repos in the cloud**	●	●	●	●
	Cloud hosting is good with huge repos	●			
DIFF	**Can diff code moved across files**	●			
	Can show you the history of a method	●	●		
	Enterprise Support	●		●	

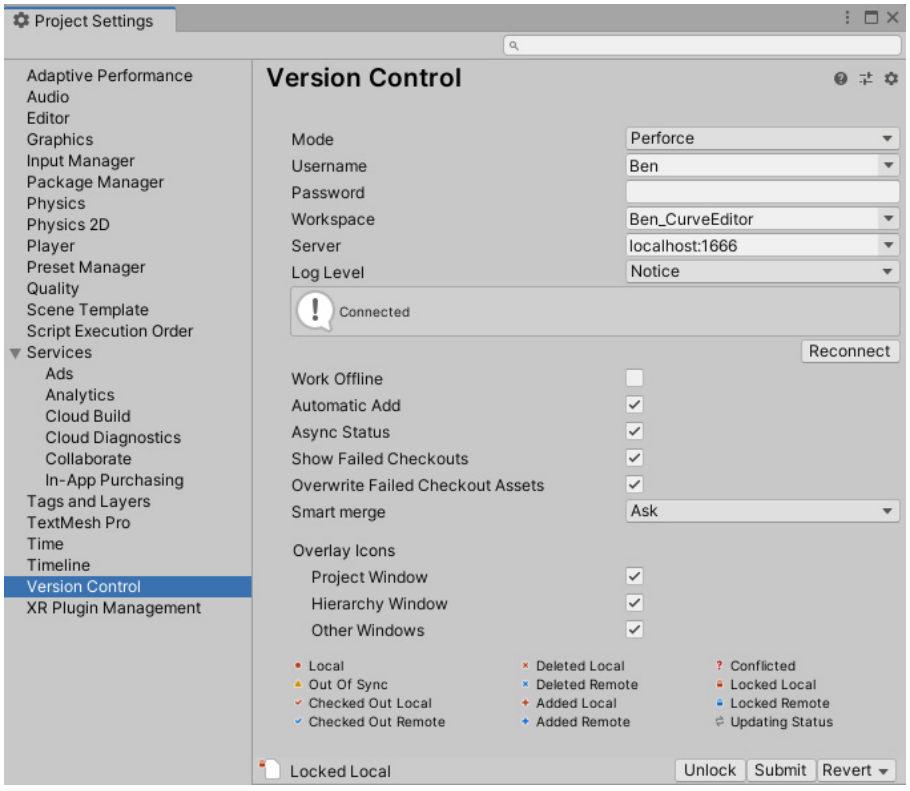
Setting up Unity to work with version control

This section provides information on setting up Unity to work with Git, Perforce, or Plastic SCM. By understanding some of the key workflows for each solution, you can make an informed decision about which system will best suit your team.

Editor project settings

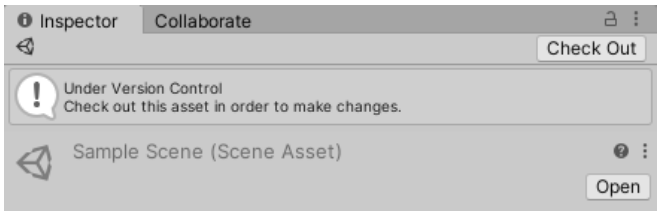
Perforce Helix Core

Unity Editor integration is available with most version control systems, and Perforce Helix Core integration is built into the Editor. You only need to enable it via **Edit > Project Settings > Version Control**. Set the Mode to Perforce, and fill in the information of your workspace and server settings.



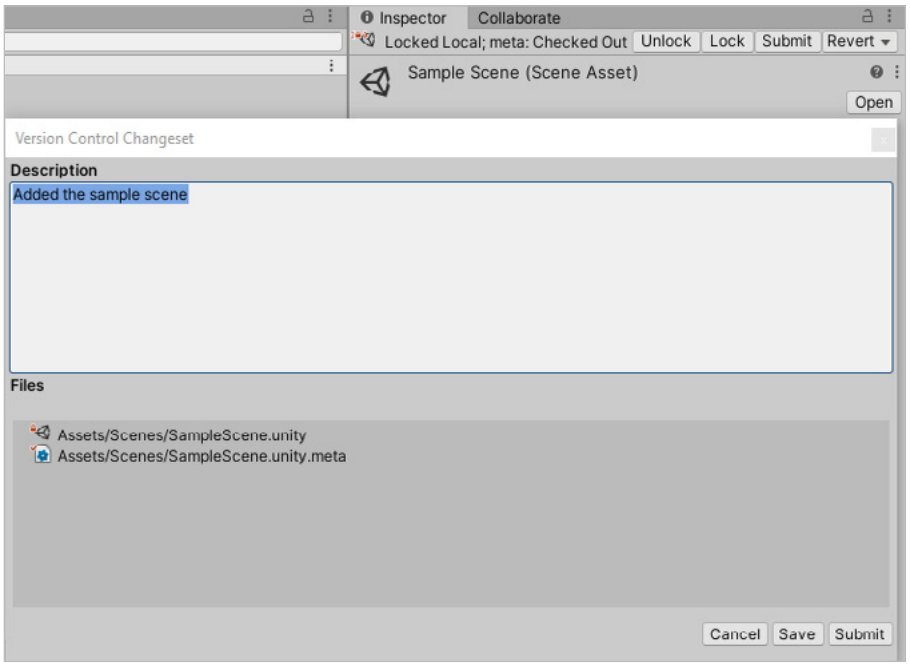
Setting up Perforce Helix Control for a project

Once this is enabled, you will see that files are now considered “Under Version Control,” with the option to check them out.



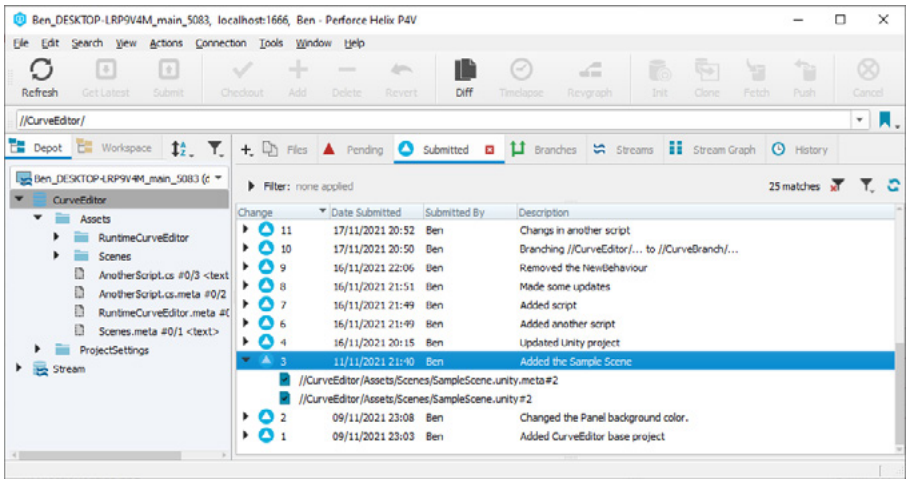
Files under version control

Once a file is checked out, you can lock, unlock, submit, or revert the file. Choosing to submit will bring up a changeset dialog for you to add your commit message before submitting it into the repository.



Changeset dialog box

Use the Helix P4V interface to view the project history.



View the project history

For more on getting started with Perforce Helix Core and Unity, check out the Perforce blog.

Plastic SCM

Plastic SCM is available built into Unity with any of the below editor versions:

— 2019.4.32f1 or later

— 2020.3.20f1 or later

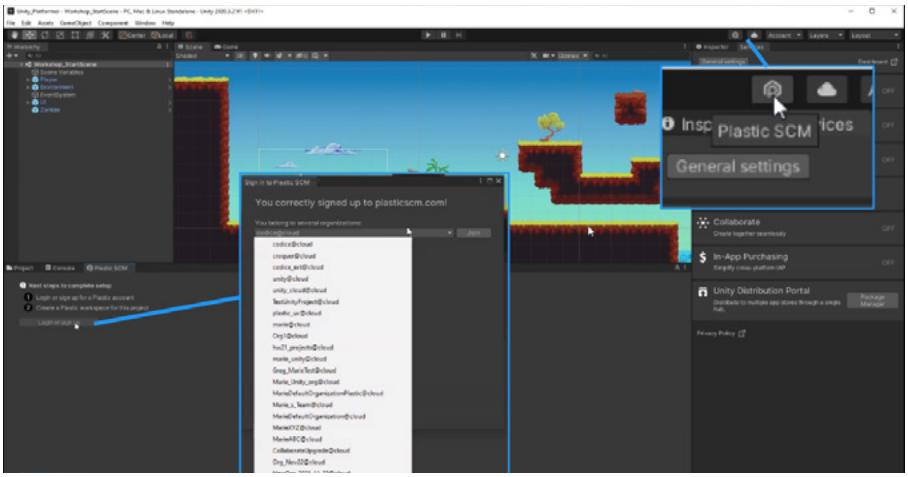
— 2021.1.25f1 or later

— 2021.2.0b16 or later

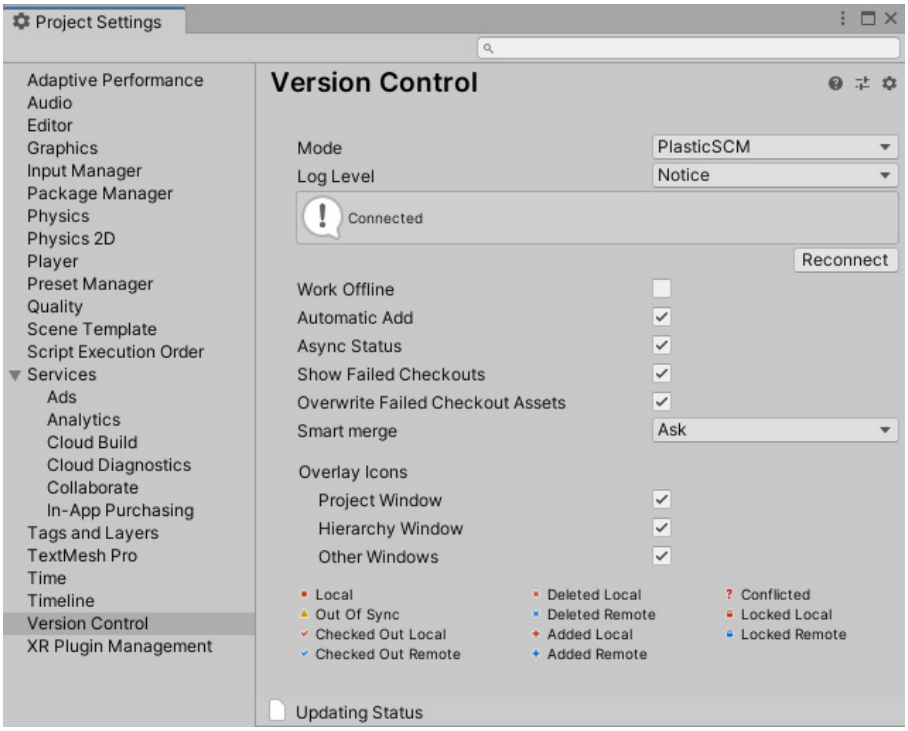
— 2022.1.0a12 or later

You can enable this by clicking the Plastic SCM icon in the toolbar on the top right, then complete your set up by connecting Plastic SCM to your Unity ID, joining or creating an organization, set up or join a new repository, and create your workspace. You can find more detailed step-by-step instructions using using this guide.

Alternatively, you enable this via **Edit > Project Settings > Version Control** in Unity 2020 LTS, then set the Mode to PlasticSCM.

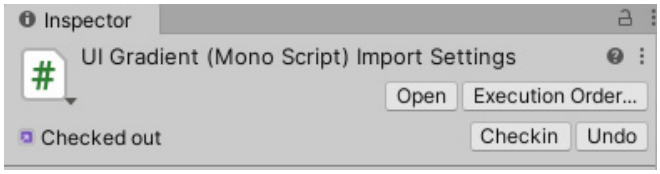


Using Plastic SCM with Unity

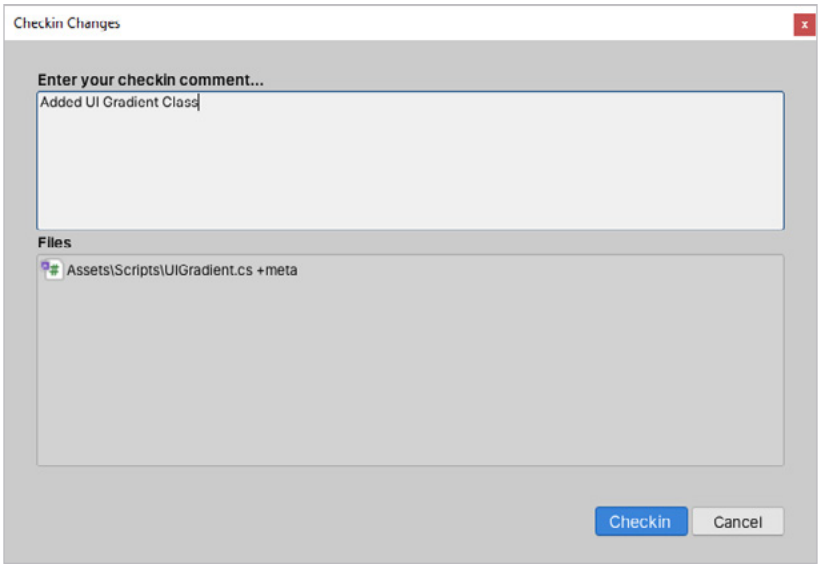


Using Plastic SCM with Unity

The interface is very similar to the Perforce option. Files can be added, checked out, reverted, checked in, or submitted, directly from the Editor.

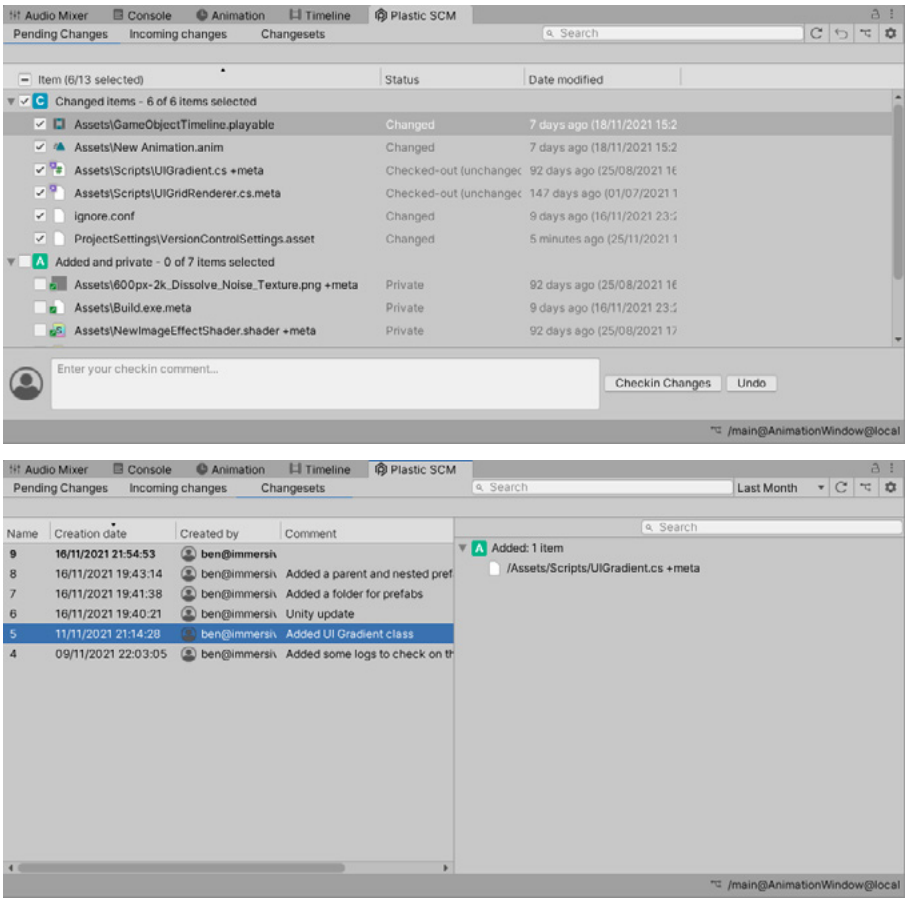


Working with files in Plastic SCM from the Unity Editor



Checking in a file

Plastic SCM also has the advantage of having a Changeset window available in the Unity Editor via **Window > Plastic SCM**.



Changeset window

For more information on setting up Version Control in Unity, check out the official documentation.

Git and other solutions

For all other VCS, open the **Edit > Project Settings > Version Control** window, and select Visible Meta Files from the dropdown menu. There are no other options here, but meta files must be visible in order for version control systems to detect them.

What to ignore

When working with a Unity project, or any project for that matter, only files that cannot be generated should be placed under version control.

For Unity projects, that means only files in the Assets and Project Settings folders should be committed to your repository. Unity can automatically recreate all the other folders. Under no circumstance should you commit the Library folder, since this folder can get very large and Unity will recreate it when launching the Editor if it doesn't exist.

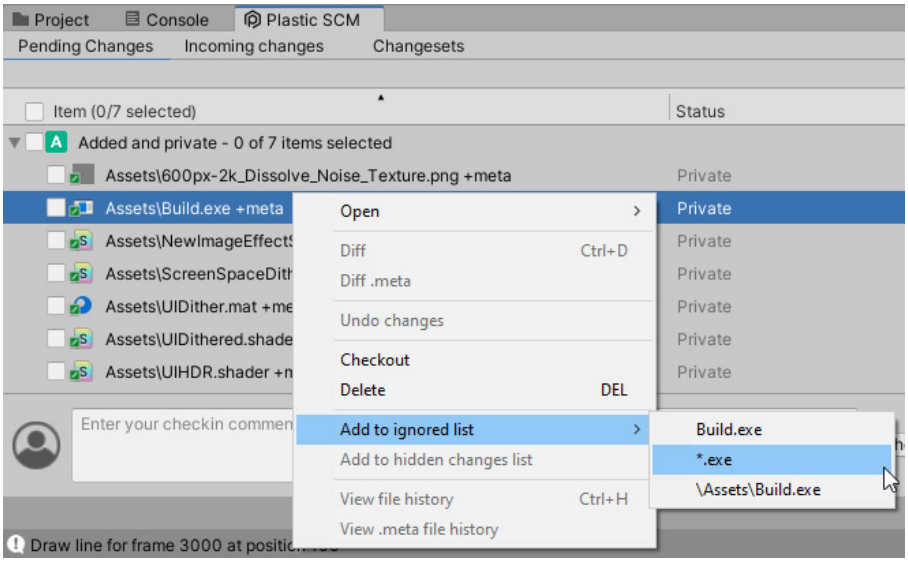
— With Perforce, you need to explicitly add the Assets and Project Settings folders to your depot.

— Plastic SCM automatically selects the appropriate folders and files to place under version control when set up from the Unity Editor. There is a list that is saved in the `ignore.conf` file at the root of the project that describes which files are ignored. To learn more about setting up the "ignore.conf" file, check out this blog post.

— Git requires a **.gitignore** file to indicate what files should never be included. Depending on your Git GUI client, you can select a template when creating a repository, or this can be done through GitHub if you set the hosting up first. Alternatively, a template can be downloaded here.

You should also avoid committing things like .exe or .apk files. Additionally, gradle and xcode projects built from your Unity project should not be added to the repository.

A small exception to this rule is if you were to set up automated build processes for your gradle or xcode projects, but then they would be typically committed to a repository of their own.



Files can be added to the ignored list directly from the Unity Editor when using Plastic SCM.

Working with large files

Unity projects are made up of a lot more than just code. In fact, scripts can often be heavily outnumbered by other asset files in a Unity project. These assets are stored as binary files: Textures, models, Prefabs, audio clips, timelines, and so on. This results in two things:

— They can be hard to compare between revisions.

— The diff cannot be described, so the whole file is written when a change is pushed to the repo.

Again, in a distributed environment, the entire project history is available on a user's local machine. Now if you have a history of large files that have had many changes over a long time, then you will have that many copies of the file stored on your machine. This can quickly consume a large portion of your hard drive space!

It's for this reason that historically, teams prefered a centralized workflow. This way, large historical versions of binary files would only live on a central server, with individual users only accessing the latest version on their machines.

Both Perforce and Plastic SCM are centralized systems that can handle large files well. Plastic SCM also gives you the option to work in a distributed pipeline, and large file sizes is the tradeoff that you need to consider when choosing between these options.

Another feature of Plastic SCM is the Dynamic Workspace, which relies on a virtual filesystem. This means that the Dynamic Workspace downloads files on demand – so, while you see everything in your workspace, in reality not everything is downloaded.

Git, being distributed, can struggle with large files. Be sure to also include Git LFS if you will be working with large files. Git LFS replaces your large files in the .git folder with text pointers while storing the actual asset on a server such as GitHub.

Best practices for version control

Regardless of which VCS you use, many best practices can help your team work more effectively. Every team has different needs, so every practice won't fit every team.

These tips come from the Unity Enterprise Support Team, who are helping to optimize real-world projects for some of the biggest studios out there.

Commit little, commit often

This is by far the easiest change you can make to your workflow, yet it's the one that some developers struggle with the most. When working with other project management tools, it's likely you have already broken down the work into small, manageable tasks. Commits should be exactly the same.

A single commit should only relate to one task or ticket, unless a single line of code magically fixes several bugs. If you are working on a larger feature, break it down into smaller tasks, and make commits for those tasks. We'll dive into feature branches later.

The biggest advantage of using smaller commits is that when something does go wrong, you will find the change much more easily and can revert the negative change without affecting any other positive changes.

Keep commit messages clean

Commit messages describe the history of your project. It's much easier to find the change that added high-score tables to your game if its commit message says "Added high score tables to the menu" and not "bet you can't beat my score on these new tables!"

When working with a task ticketing system like JIRA or GitLab, it's even better to include a ticket number in your commit. Many systems can be set up to work together with smart commits, in which you can actually reference tickets and change their status from your commit message.

For example, the commit "JRA-123 #close #comment task completed" would set JIRA ticket JRA-123 to closed, leaving the comment "task completed" on the ticket.

For more on setting this workflow up, see the documentation in JIRA or the Pivotal Tracker service in GitLab.

Avoid indiscriminate commits

The only time “commit -a” (the git command for “commit all changes”) or any of its counterparts should be used is with the first commit of a project. Usually, this is when the only files in the project are README.md.

A commit should only ever include files that are related to the change you are committing to the repo. Particular care should be taken when working with Unity projects, as some changes may result in several files being marked as changed, such as scenes, prefabs, or sprite atlases, even though you didn’t intend to make any changes to them.

If you accidentally commit a change to a scene that someone else is working on, that could cause a headache for them when they go to commit their changes and find they need to merge your changes first.

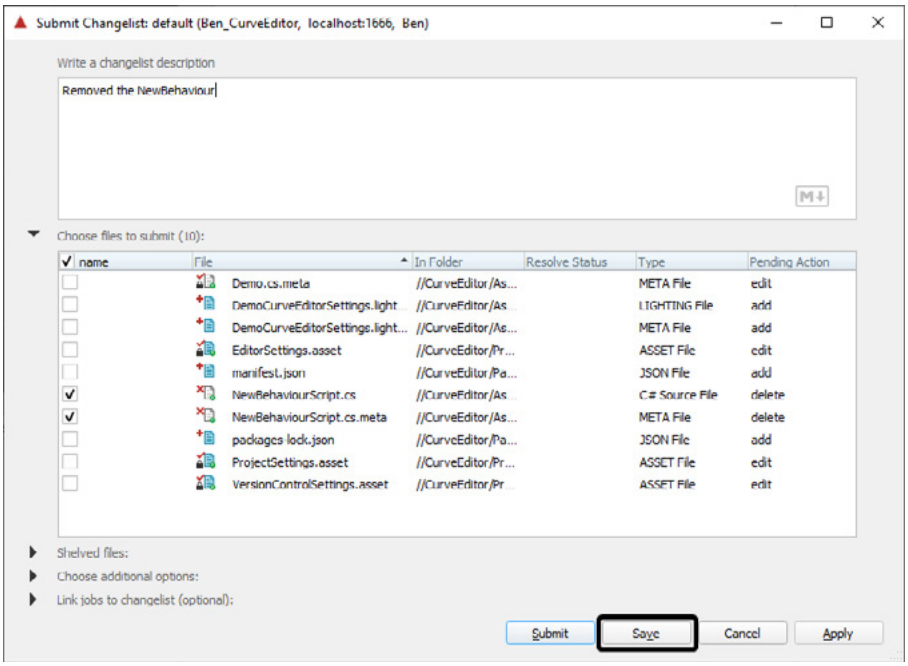
This is one of the most common mistakes that people who are new to version control will make. It’s important to understand that you should only commit what you have changed in the project. To learn more, check out this blog post on how to speed up your workflow.

Get the latest

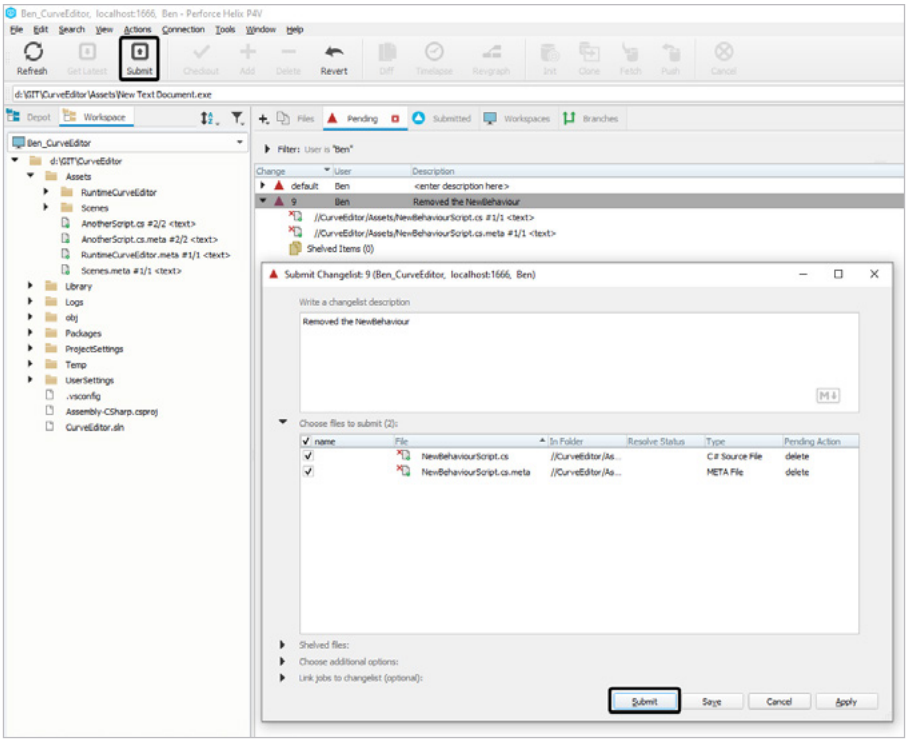
As often as it makes sense, pull the latest changes from the repo into your working copy. It’s not good to work off in isolation, as this only increases the likelihood of merge conflicts. A typical daily workflow in each system would be something like this.

Git	Perforce

— git pull
— Then as many times as you like:
  — Make edits in your working copy
  — git commit your changes
  — git pull the latest changes
— Once you are happy with your change set of commits
  — git pull once more
— git push to send your commits to the repo
	— Get latest
	— Check out files to work on
	— Make edits
	— Submit changes



Saving changes to a new changelist in P4V

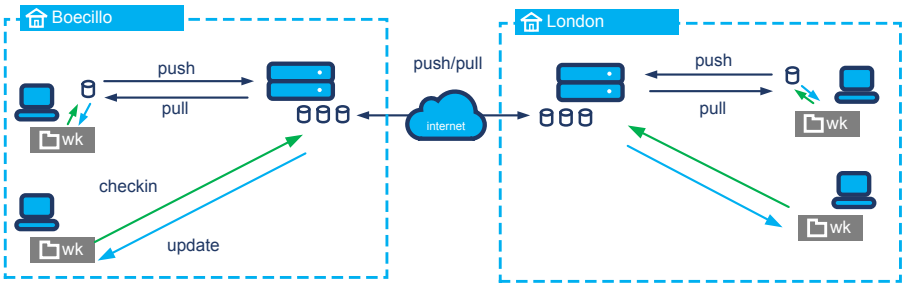


Submitting a changeset in P4V

Plastic SCM (centralized)	Plastic SCM (distributed)	Plastic SCM (multi-site)
— Sync Repositories — Pull visible — Check out files to work on — Make edits — Check in changes — Sync Repositories — Push visible	— Pull changes from the server — Check in changes to your local copy — Pull any new changes — Push your changes back up to the server	— A hybrid of the two, depending on your setup

Plastic SCM workflows are a little different because you can work in centralized, distributed, or multi-site configurations.

Multi-site configurations can be fairly unique, with each user working in either a centralized or distributed workflow.



Multi-site Plastic SCM configuration

Consider the following example:

— Two teams

— Each team has an on-site server

— Team members at both sites check in locally or distributed but benefit from the speed of a close on-site server

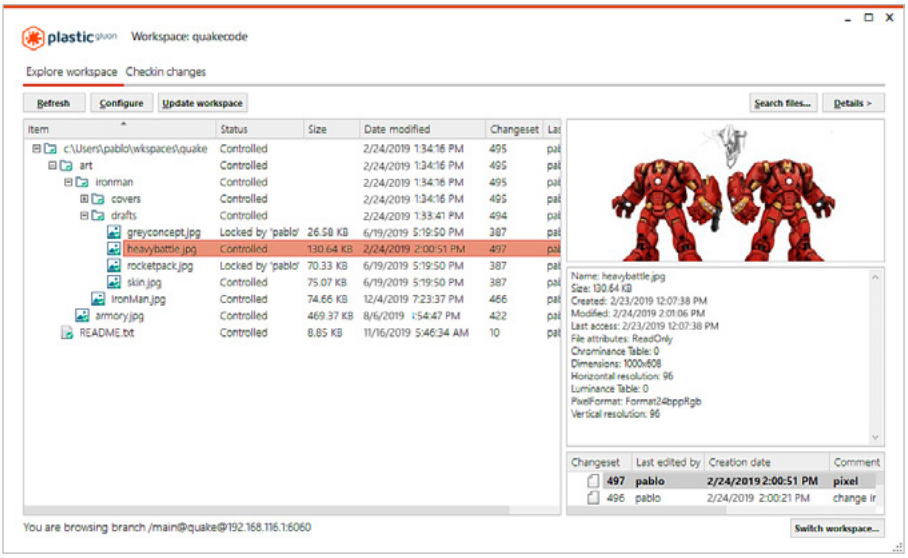
— Servers push/pull between one another to keep fully or partially in sync

Know your toolset

Whichever VCS your team chooses to work with, make sure that the team is comfortable using it and understands the tools at their disposal.

If you're working with Git, not everyone needs to use the same GUI client. But make sure that everyone is comfortable with the commit > pull > push workflow, and that they know how to commit only the files they need.

If you're working with Plastic SCM, let your artists get comfortable using Gluon to simplify their workflow. Gluon lets you decide which files you want to work on and only download those, removing the need to download and manage the entire project. It allows you to lock a file to prevent others from working on it, and, once you're finished, users can submit files back to the repository and unlock them again.

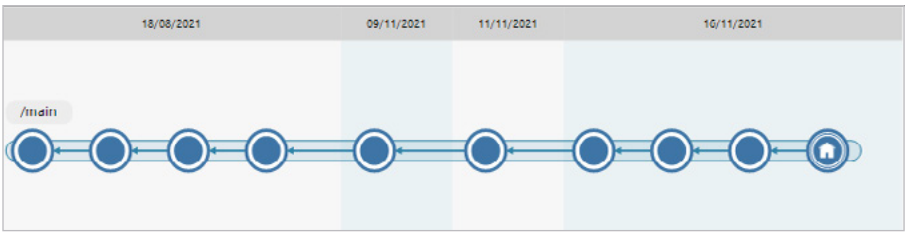


Gluon in Plastic SCM

If you are working with Perforce Helix Core, use the built-in Unity Editor tools for managing version control directly from the Editor. This is incredibly useful, both for artists or for general handling of Unity asset files such as scenes, Prefabs, and so on. You can check out assets for modification in the Editor, make your changes, and then check them back in without even leaving Unity.

Feature branches and Git Flow

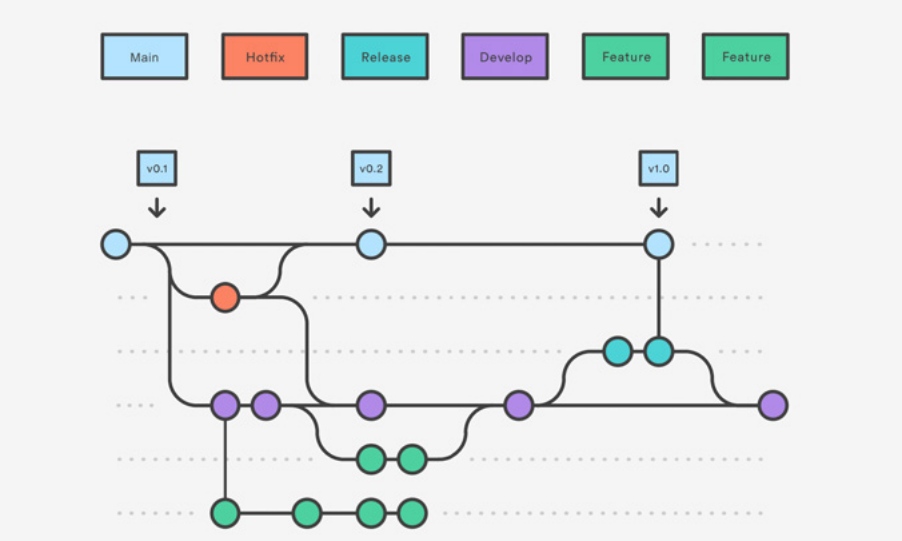
When you're working on a long-standing project with multiple release cycles, feature branching is hugely beneficial to your workflow. Often, teams work out of the same branch of a repo that would likely be called trunk, master, or main. When you do this, your entire project moves along the same timeline.



Development along the main branch in Plastic SCM

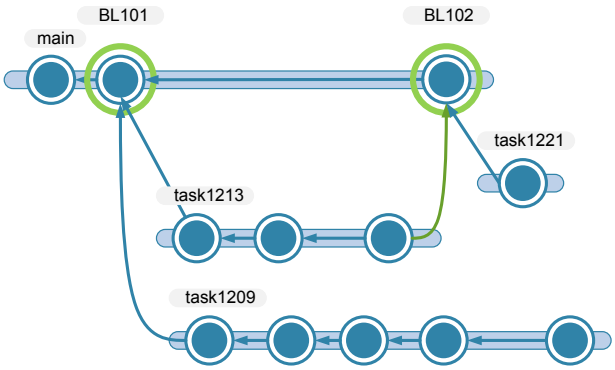
However, it can be beneficial to split the work off into several branches to work more effectively as a team.

In Git, a specific workflow called Git Flow focuses on using different branches for features, bug fixes, and releases. A developer starts out work on a new feature inside an isolated branch, and when they're finished, it's merged back into the main branch. Meanwhile, someone else may have had to do a hotfix on the previous release, fixed a bug, and released a new version safely, without any of the features still under development being included.



A Git Flow workflow allows for easier release management.

Plastic SCM also features task branches. For this pattern, you create a new branch for every task that you track. While in Git Flow, we use feature branches to develop complete, sometimes large, features, task branches in Plastic SCM are meant to be short-lived. If a task takes more than a handful of commits to implement, odds are it could be broken down into smaller tasks.



Plastic SCM branch per task pattern

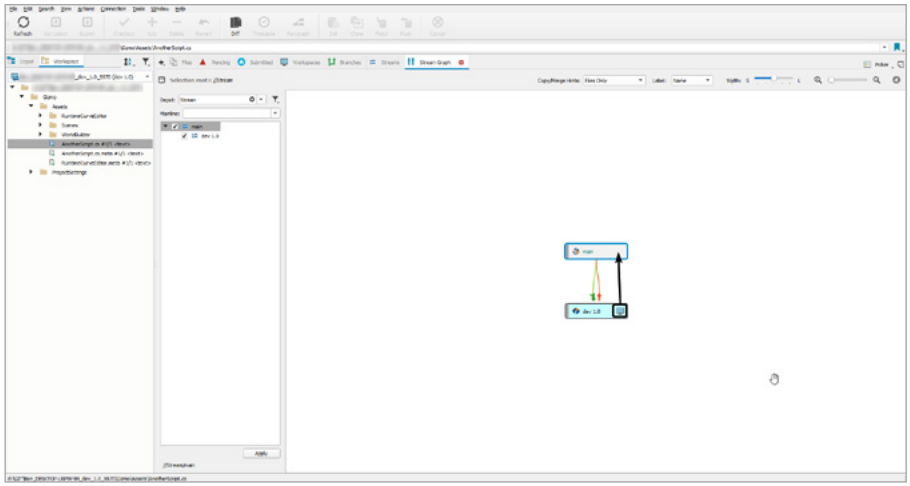
Perforce Helix Core uses a system called Streams to facilitate this style of workflow. When creating a depot to work in, you need to set it up as a stream depot type. Then, you can use the Stream Graph view to create new streams. Every stream other than the mainline stream will need to have a parent stream, so changes can be copied back up-stream.

There are different types of streams for different purposes.

mainline - serves as the base or trunk of a stream system
development - used for long term projects and major new features
release - used for fixing bugs, testing and release distribution
virtual - used to narrow the scope and submit directly to parent
task - creates lightweight branch, used for bug fixes and new features

Options when creating a new stream in Helix Core

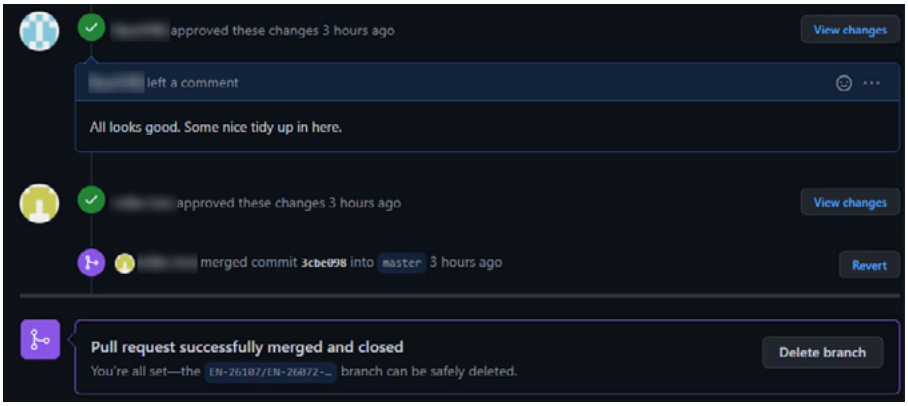
When you switch between streams on your local workstation or copy changes back upstream, only the metadata for changed files gets merged, making the context change quicker.



Perforce Helix Streams workflow. The desktop icon can be dragged between the streams to switch workspaces. The green arrow down from main shows there are changes to be brought into the dev 1.0 stream. The red arrow up to main shows we cannot copy to main until we have the latest changes.

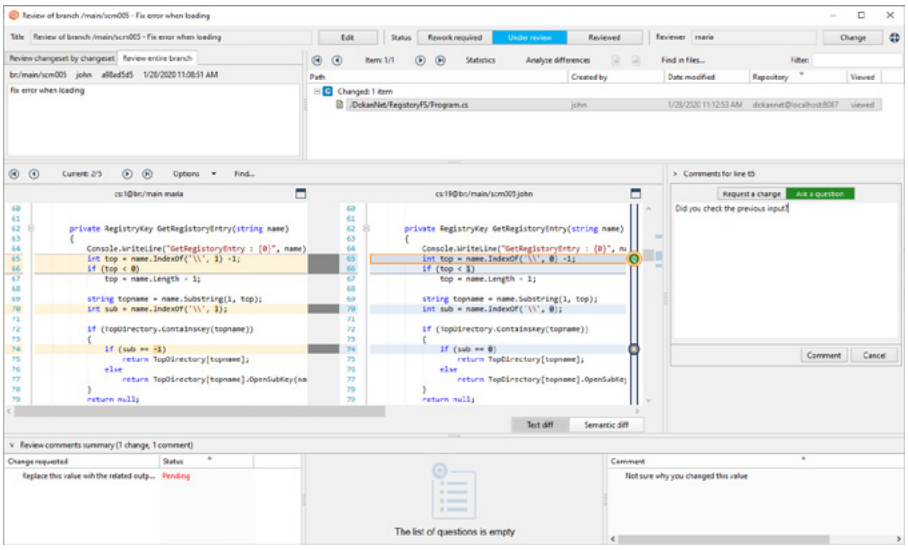
Pull requests

Once you've completed work on a feature branch, it's a good practice to use pull requests to get your changes back into the main stream of the repo. Pull requests are created by the developers of the feature or task, and it's usually the responsibility of a senior developer or DevOps to review the changes before accepting them into the mainline.



A closed pull request on GitHub

Plastic SCM and Perforce both have automated tools to help manage merging branches back into the mainline. Plastic SCM does this with the help of Mergebot, which automatically merges branches of a repo once they've been reviewed and passed validation. Perforce has an additional platform, Helix Swarm, for managing code reviews that can also be set up with automated testing.



Plastic SCM code reviews are included in the GUI.

Summary

Hopefully this book has helped you to feel more comfortable working with version control as part of a team in Unity. Even if you're working on a solo project, the principles of organizing your project and using version control can be really useful.

The biggest takeaway is the importance of clear team communication. As a team, you need to agree on your guidelines: how you should structure your project, which version control system to use, and how your workflow in that system looks. Then, when you start integrating other tools such as JIRA, GitLab, build tools, or automated testing, the work you've already done structuring your project and workflow will really come into its own.

Finally, check out the following resources to find a wealth of information on the various version control systems discussed in the book, plus more tips on setting up your Unity project for success.

Additional resources

Eight factors to consider when choosing a version control system

Introduction to version control, Unite Now 2020

Git Apprentice, by Chris Belanger and Bhagat Singh

Version Control for Games with Unity's Plastic SCM

Plastic SCM product documentation

Mergebot in Plastic SCM

Unity open project with version control

How KO_OP uses Plastic SCM to accelerate production

The hidden productivity costs disrupting your release timelines

Perforce setup

How to Configure Helix Core and Game Engine

Helix Core documentation

SVN setup

Using external version control with Unity


Unity®