

# Lab - Version Control

Assessment

Goals

Resources

Developing software in a team

Branching

How's that useful?

What to do

Git

GitLab

SourceTree

Create a new repository

Clone a repository

Add content to the repository

Update files

Pushing changes to GitLab

Adding a new Unity project to the repository

Modifying a scene

Multiple clones

Changing things at the same time

Dealing with conflicts

Assessment



<http://xkcd.com/1597/> (<http://xkcd.com/1597/>)

## Assessment

- This lab is individually assessed.
- Assessment during a lab session.

- Due on Tuesday, Jan 25<sup>th</sup>.

## Goals

- To become familiar with version control.
- To learn how to use Git.

## Resources

- Pro Git book (<http://git-scm.com/book>).

## Developing software in a team

When developing software in a team, there arises a need to manage the project's source code in an organised way. For one, people make mistakes (like deleting things by accident) - it saves a lot of time when these mistakes can be undone quickly. A second issue is that many people will be making changes to the software at the same time. These independently made changes need to be merged into one project and stored somewhere, so that all members of the team have access to the updated project as progress is being made. This is what *version control* is for.

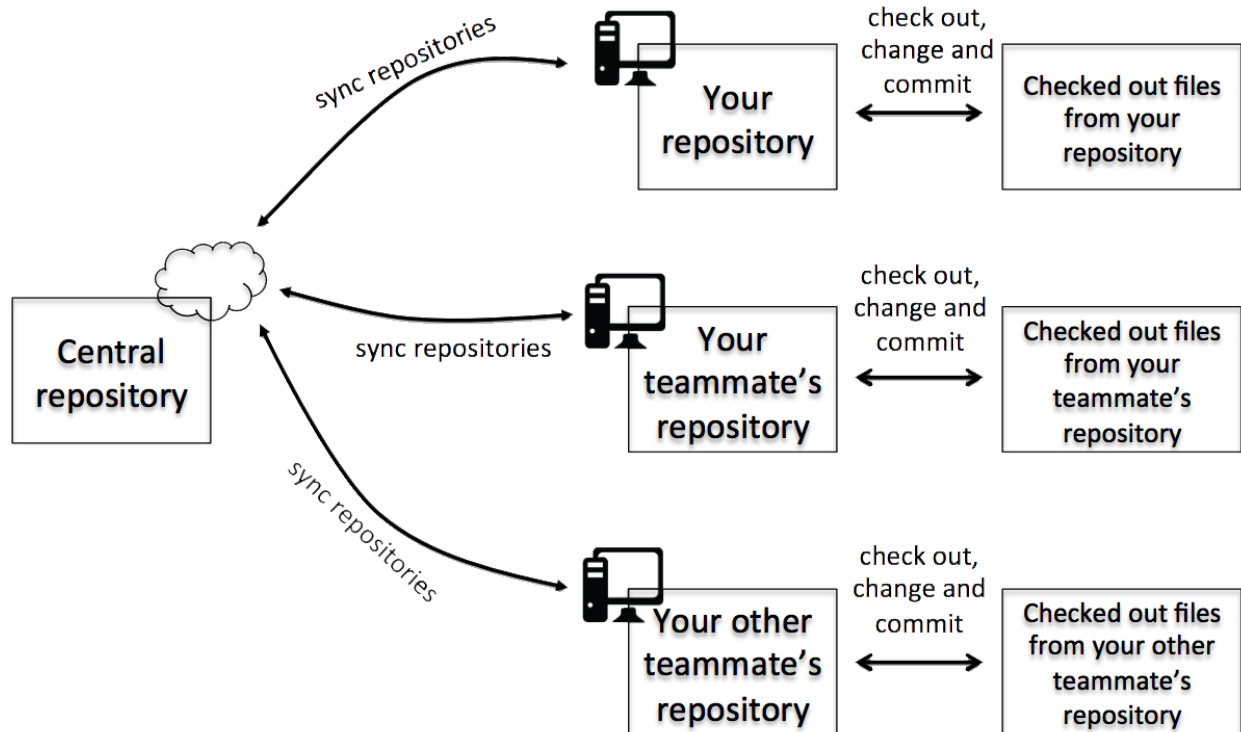
Version control (also referred to as *source control*) is an essential part of professional software development. If it's new to you, it may seem like a bit of extra work. There's a learning curve, but once familiar with version control you'll wonder how you ever got by without it.

There are number of different tools for version control. In this course you'll be using Git (<http://git-scm.com/>).

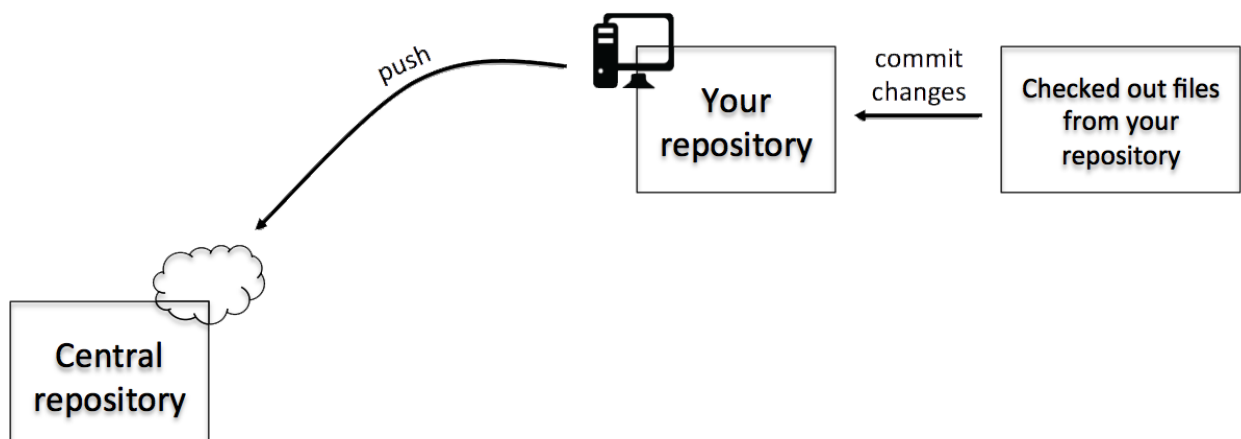
The fundamental element of version control is the *repository* which stores and catalogues files. A repository can be thought of as a database or a library where one can check-in and check-out files related to the project. In your case it will be the Unity project for the game your team is developing, with all its scripts, scenes and game assets. In a distributed version control system, such as Git, every team member has a copy of the repository on their machine. You don't see the repository - it's inside a hidden folder (.git) in the root directory of the project. You can interact with the repository using the tools provided by the version control system (from the command line) or via third-party applications (such as SourceTree). You can check out project files from the repository, make changes, and check the new or updated files back in. The repository keeps track of all the changes made over time. You can check out past versions of the project. This means that if you delete something from a previously checked-in file (or even the entire file), you can retrieve a previous version, thus restoring the deleted content.

Repositories can be cloned to multiple locations and synchronised. For the project in this course, you will set up a repository on a remote server, which provides repository hosting services (GitLab). This *remote* (also called *central*) repository will be cloned by every member of your team, so that each of you will end up with a copy of

the repository on your computer. Typically, when cloning, the version control tools will check out the latest version of whatever is stored inside. So, the files that you see after cloning (remember that repository itself is hidden) are the checked out files.

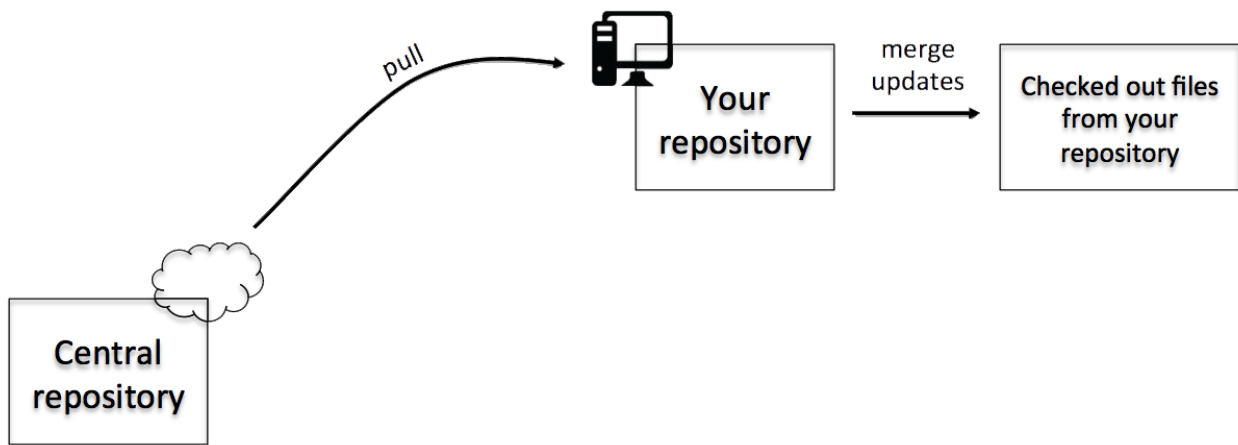


Each team member can make changes to the project and commit modified code and assets to their local repository. This enables concurrent work on different aspects of the project. You can commit your local repository as many times as you like. When you have something that is ready to be shared with the team, you *push* the changes from your local repository to the central one - the two repositories will sync up. Other team members can *pull* from the central repository, and thus have your changes incorporated (automatically) into their projects.



A *pull* from the central repository will also merge the changes into your checked out files (unless you're on a different branch, which we'll talk about in a bit). These merges are non-destructive - they will not erase any of the changes made by you to your checked out version (regardless of whether these changes have been

committed to your local repository or not).



This is more or less the basic scenario for how to use version control in order to coordinate team development of a single project.

## WARNING!



Version control is generally there to make your life easier, but there are few things to be mindful of:

- Version control **will overwrite and amend files** on your machine. Normally this is exactly what you want, and it's very unlikely that you will lose work throughout this process. Still, if you're not sure what's going on, make a copy of your work first. It can't do any harm!
- Always make sure that any files open for editing (Visual Studio Code (VSC), MonoDevelop, Inkscape etc) are saved before doing any operations with version control. Otherwise, you may find yourself in a world of unhappiness!
- Version control systems are very good at merging different versions of text files. Even when two people make changes to the same file, they can usually be merged without an issue...unless the two parties are changing exactly the same lines of code - then the system might get confused and force you to do the merge manually.
- Merging binary files (images, audio, etc) is more or less impossible. Whenever you have binary files in source control, ensure that only one person at a time is working on a given binary file.
- **Don't forget to commit newly created files** into your repository. When you create a new file, it needs to be added to the repository (through the version control tool), otherwise it won't get checked in alongside your other changes. It is not uncommon for a developer to push changes to the central repository that make the project dependent on new files that didn't get checked in. That developer won't notice the problem, because everything will compile fine on his/her machine (the new files exist on his/her hard-drive). However, when other team members sync up, their project gets updates with the changes that may require the new files, but they don't receive the new files. As a result everyone else (except the forgetful developer) gets flooded with compile errors and cannot build the project.

## Branching

The workflow for team development described above is very basic - it assumes everyone is working on the same branch of the repository. Branches are somewhat analogous to alternate realities. Did you ever see a sci-fi movie where the timeline can split, leading to alternate realities? Imagine if you could split a timeline into two alternate realities with the same past but the future events continue to evolve independently and differently for each branch.

That's more or less what branching is for repositories. You can split a project into separate branches and make independent changes in each branch. You can check out the code from a specific branch, make changes, and check files in - the repository will keep track of different changes in different branches. However, in contrast to alternate realities, the branches can be merged together as you wish.

## How's that useful?

Imagine you're half way through development of a new feature. In the meantime a critical bug is found and it needs to be fixed immediately - perhaps it's holding up the work of the other members of the team, and you're the best person to fix it. You cannot just implement the fix in your currently checked out code because the new feature you've been working on is not ready for a push to the central repository. Besides, who knows how many bugs the new feature introduces - you don't want to introduce new bugs while rushing to fix an old one. The best thing to do in this situation is to commit your current changes (unfinished feature) into a branch. Switch back to the main branch that does not know about your new feature (this will usually be the *master* branch) and implement the fix. Check everything in and push to the remote repository. Your teammates can sync up, getting the version of the code with the bug fixed. Afterwards you can switch back to the branch with your new feature and continue with your work. You can do this as many times as you want. When your work is complete, merge the feature branch back into the *master* and push everything to the remote repository.

There are other scenarios where branching is useful. Branches can be also pushed to the central repository. This way someone else can check it out and contribute to the branch without affecting other members of the team.

## What to do

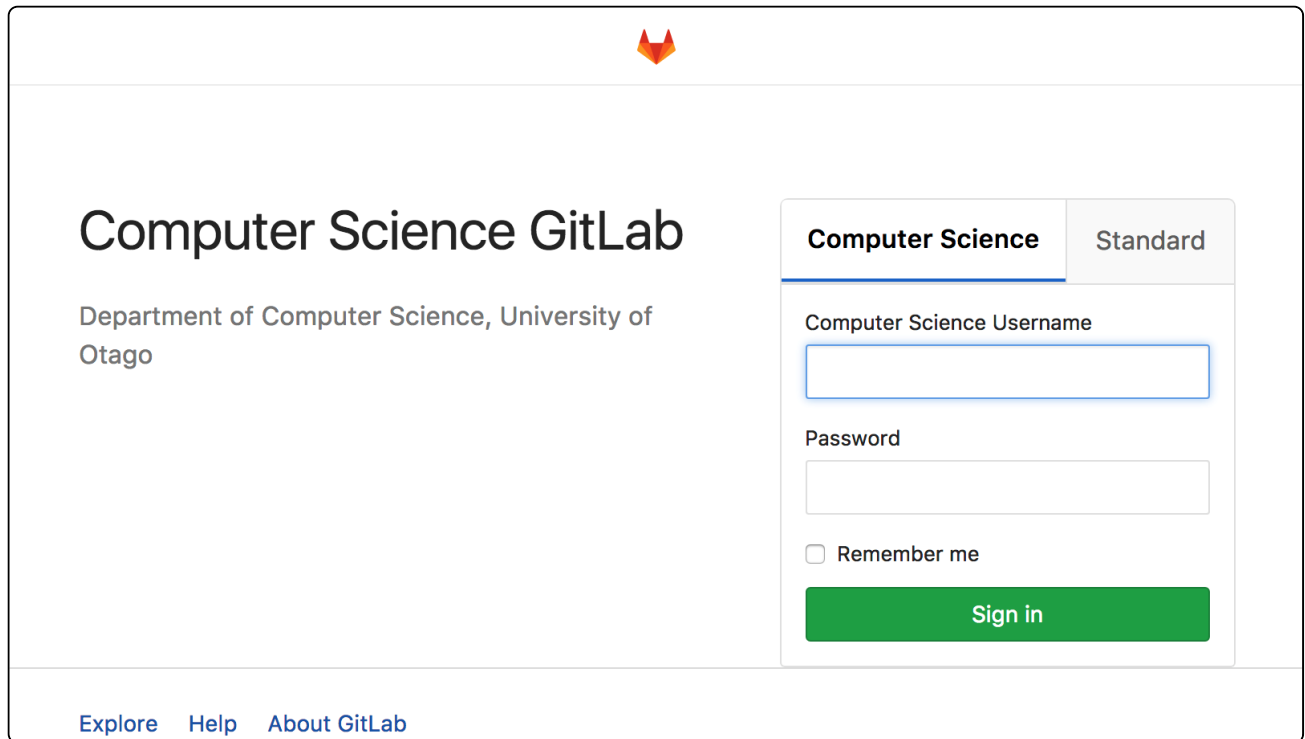
Hopefully after this lengthy introduction, you have an idea of what version control is about, and how useful it can be for your game development. Now it's time to learn how to use the version control tools that let you do all of this (and much more if you care to get good at it).


## Git

Git is the version control system you'll be using in this course. Though a brief introduction to version control has been given in this lab, it's a good idea to read through a brief introduction to version control (<https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control>) and Git basics (<https://git-scm.com/book/en/v2/Getting-Started-Git-Basics>).

## GitLab

GitLab is a hosting service for git repositories. The CS department hosts a GitLab server at <https://altitude.otago.ac.nz/> (<https://altitude.otago.ac.nz/>) - you can login and use it with your CS account credentials (select the LDAP login option). You have used this service already, in the previous lab, to get the SpacInvaders repository. The CS GitLab server is where your labs, as well as the main development project for this course will reside. Once you login to GitLab (<https://altitude.otago.ac.nz/>), you can create new projects and fork from existing ones. You can also control access to your project.





# Computer Science GitLab

Department of Computer Science, University of Otago

Computer Science

Standard

Computer Science Username

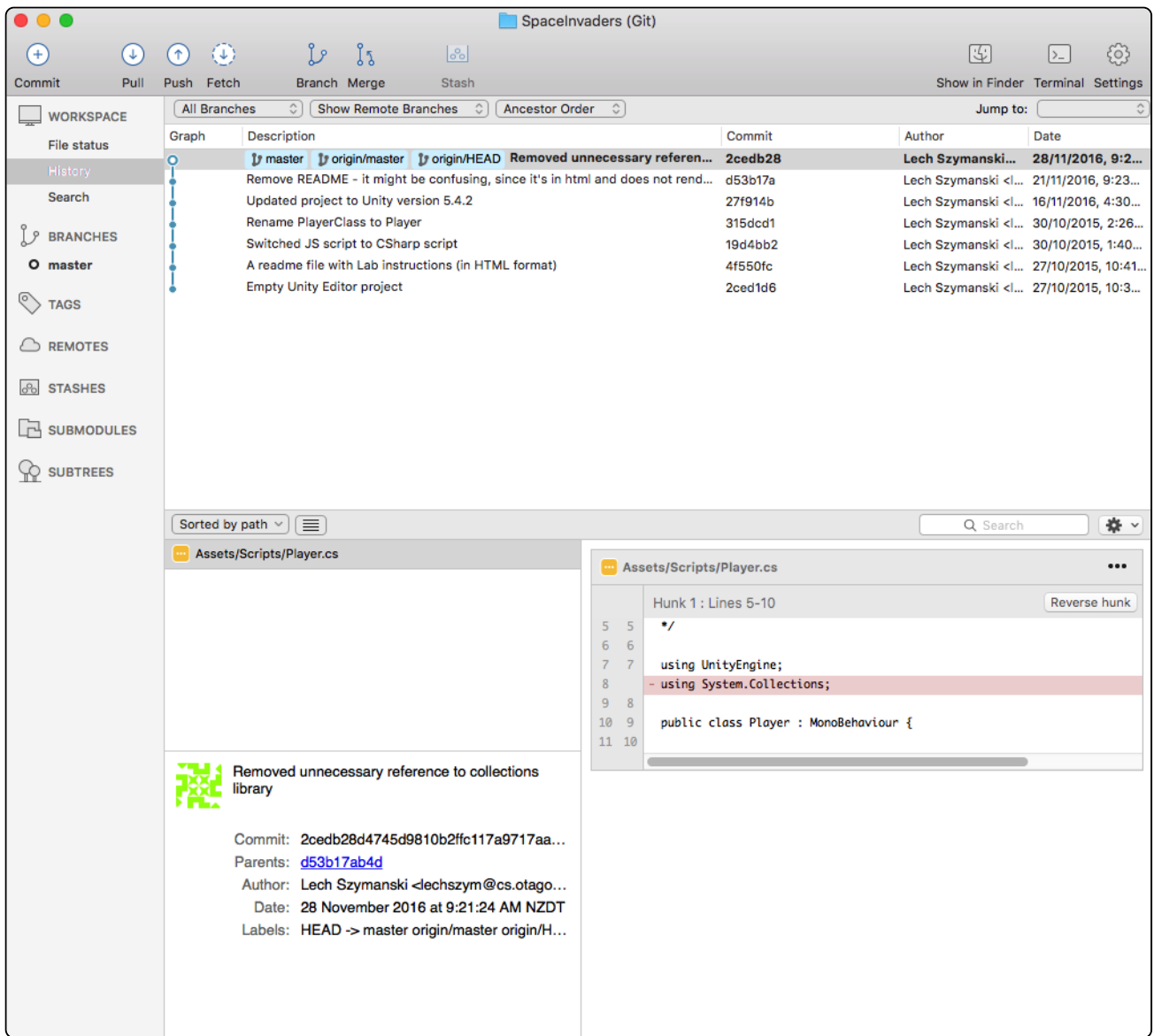
Password

☐ Remember me

Sign in

## SourceTree

SourceTree is the application you will be using to manage the local clones of the GitLab repositories. Using SourceTree you can clone repositories from GitLab, commit changes (to your local repository), push your changes (to GitLab), pull changes (from GitLab), create branches, etc. You can do all those things from the command line as well using the 'git' tool. However, SourceTree is a bit easier to use if you're just starting with version control.



## Create a new repository

- . Login to Gitlab: <https://altitude.otago.ac.nz/> (<https://altitude.otago.ac.nz/>).
- . Once you're logged in, you should be presented with the "Projects" page. Click on the "New Project" button and select the "New project/repository" option. On the page that opens select the "Create blank project" option. Name the new project "VCLab". Make sure the "Visibility Level" is set to "Private", disable the "Initialize repository with a README" option, and press "Create Project".

GitLab Menu Search GitLab

New project > Create blank project

**Create blank project**  
Create a blank project to house your files, plan your work, and collaborate on code, among other things.

**Project name**  
VCLab

**Project URL**  
https://altitude.otago.ac.nz/ lechszym

**Project slug**  
VCLab

Want to house several dependent projects under the same namespace? [Create a group.](#)

**Project description (optional)**  
Description format

**Visibility Level** ⓘ

☒ Private  
Project access must be granted explicitly to each user. If this project is part of a group, access will be granted to members of the group.

☐ Internal  
The project can be accessed by any logged in user except external users.

☐ Public  
The project can be accessed without any authentication.

**Project Configuration**

☐ Initialize repository with a README  
Allows you to immediately clone this project's repository. Skip this if you plan to push up an existing repository.

Create project Cancel

- . A new repository/project will be created on GitLab and you should be taken to its main page. You should see a message saying the repository is empty and a list of command line instructions.
- . Right now, the repository is empty. In order to add files to it, you need to clone the repository to your hard-drive.

## Clone a repository

Cloning a repository refers to making a a local copy of a repository on your hard-drive. You get a local copy of the entire repository, with the entire history of the development.

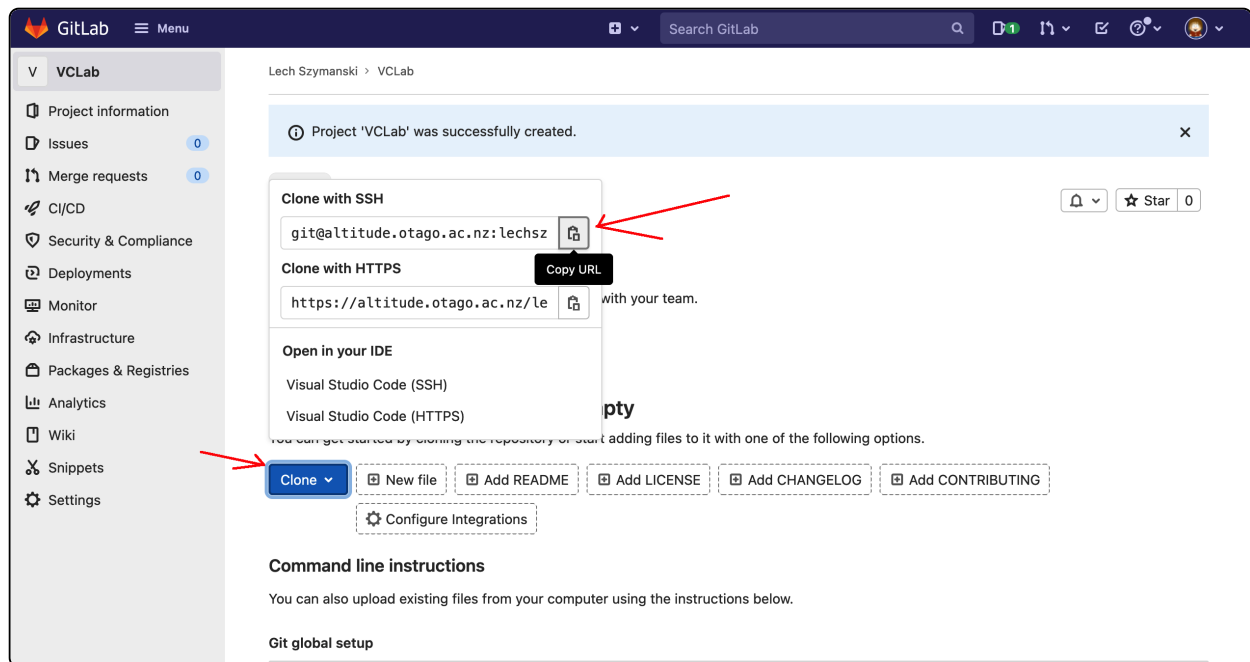
### IMPORTANT!



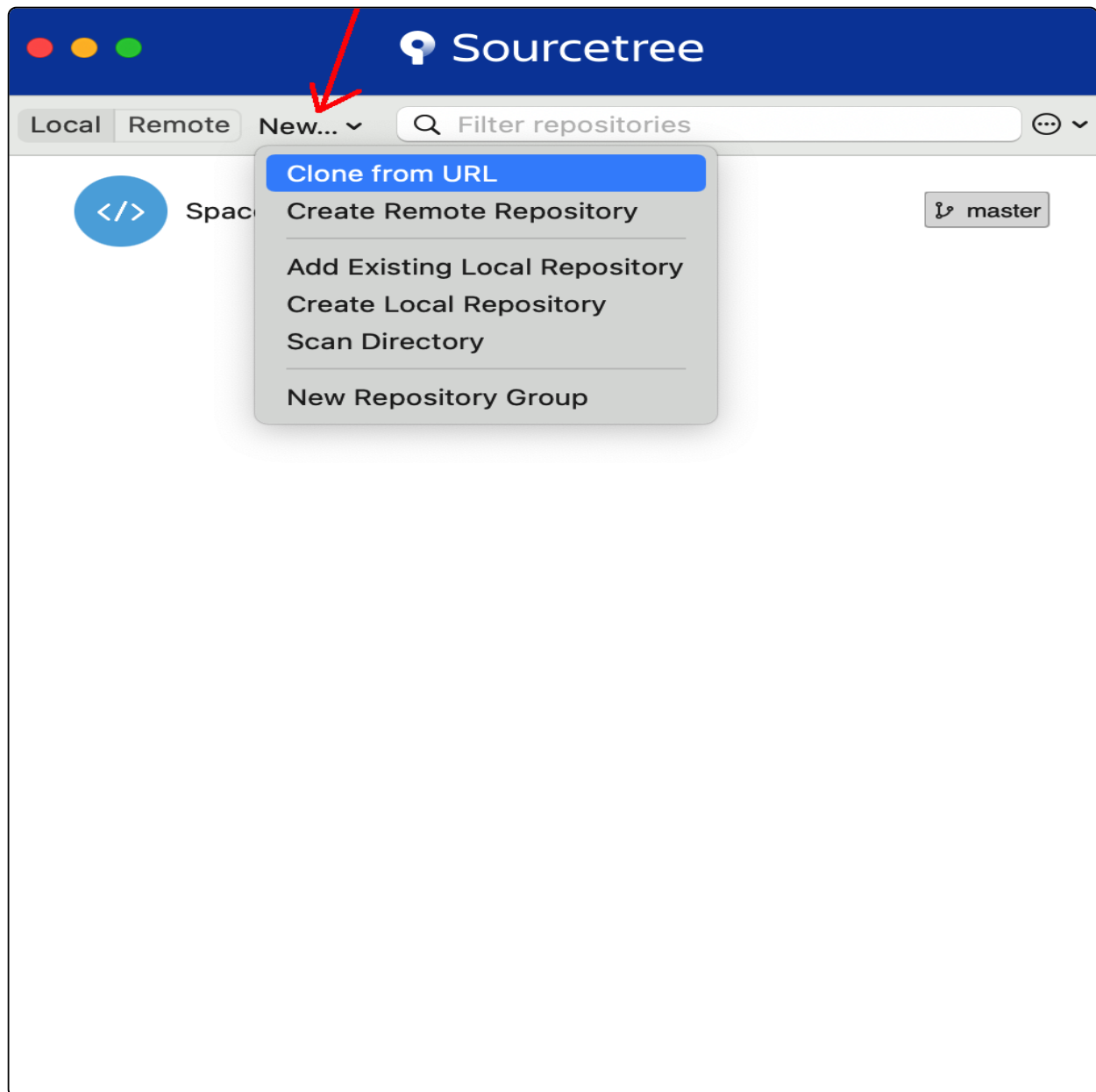
If you haven't gone through the ssh key setup on gitlab, you will have to do that first. Follow the steps 7 through 13 of the Space Invaders Lab (../lab02/index) to setup the ssh access to GitLab.

- . On GitLab, in the "Project" window of your "VCLab" repository, click on the "Clone" button. You'll see URLs to your repository - select the SSH one and copy it to the clipboard.





Open SourceTree. It's likely to open the window with the "SpaceInvaders" repo that you worked on in the previous lab. Close that window and you should be left with the main SourceTree window, where you can select, or add repositories.



- . Click on the "New..." button and select "Clone from URL".
- . In the "Clone a repository" window: paste the SSH link you got from GitLab in the "Source URL", set the destination path to somewhere on `/scratch` (you'll be creating a Unity project in this location, so can't put it on a home/networked drive). Click the "Clone" button.

### Clone a repository

Source URL:

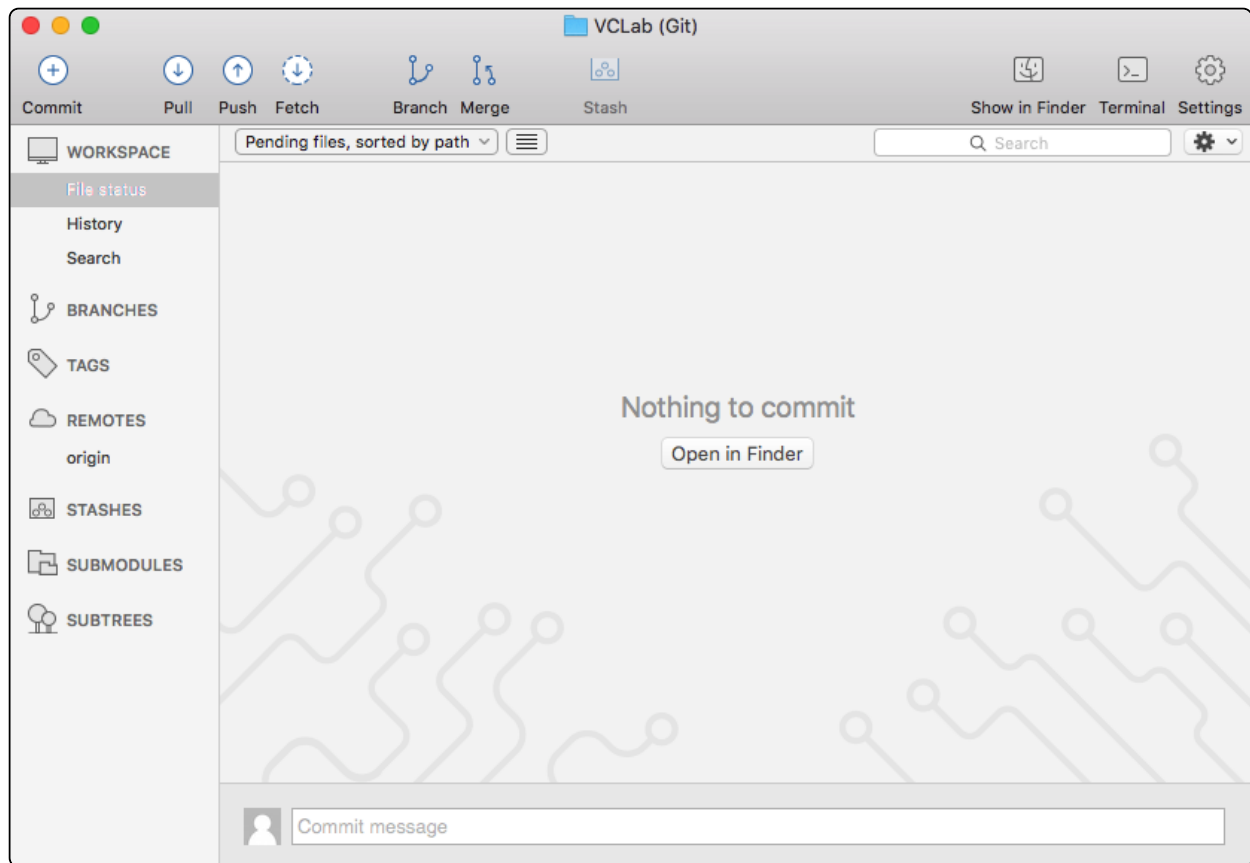
Destination Path:  ...

Name:

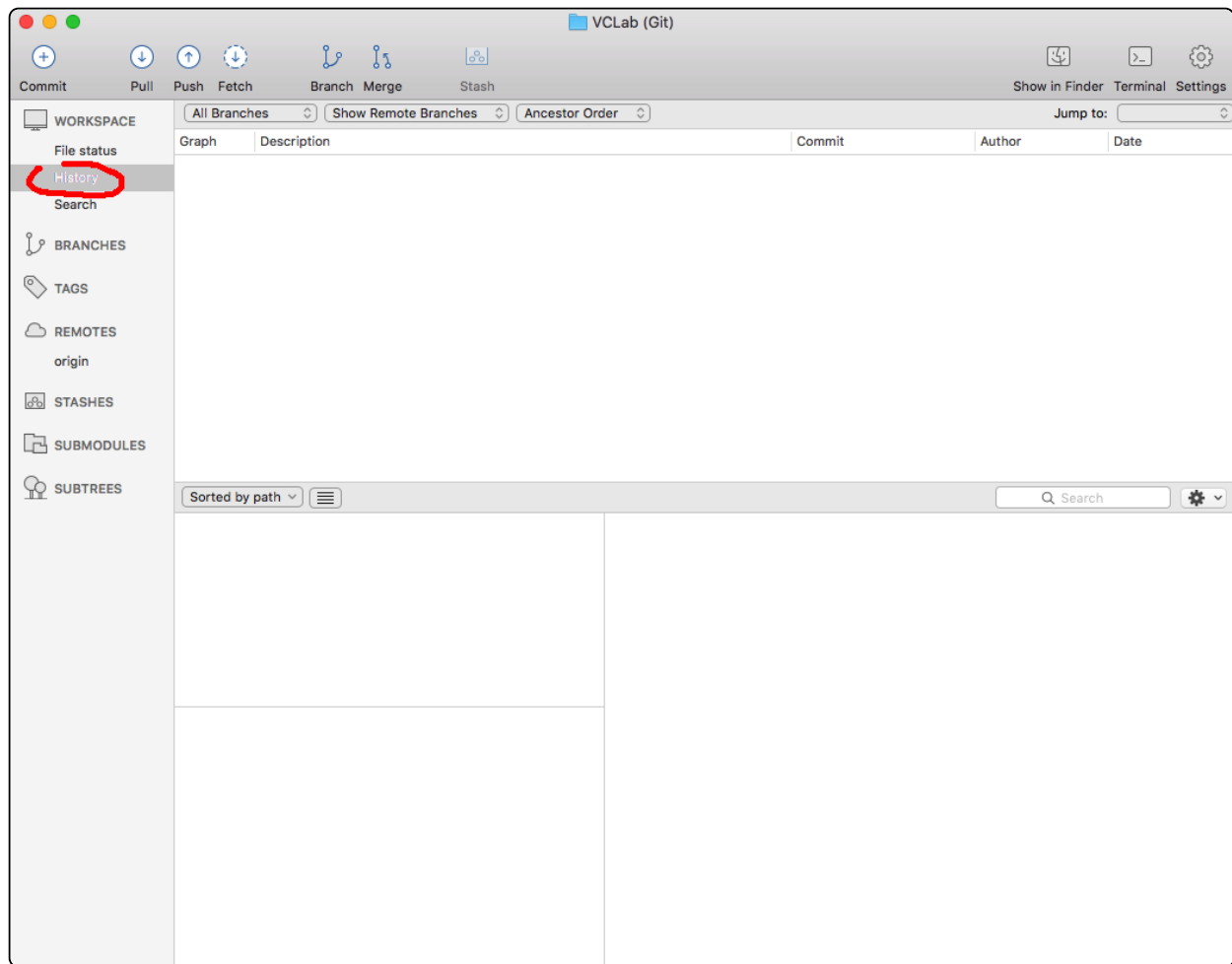
> Advanced Options

This is a Git repository

- SourceTree will make a clone of the repository and open a window showing its history. It should be empty, because the repository is empty.

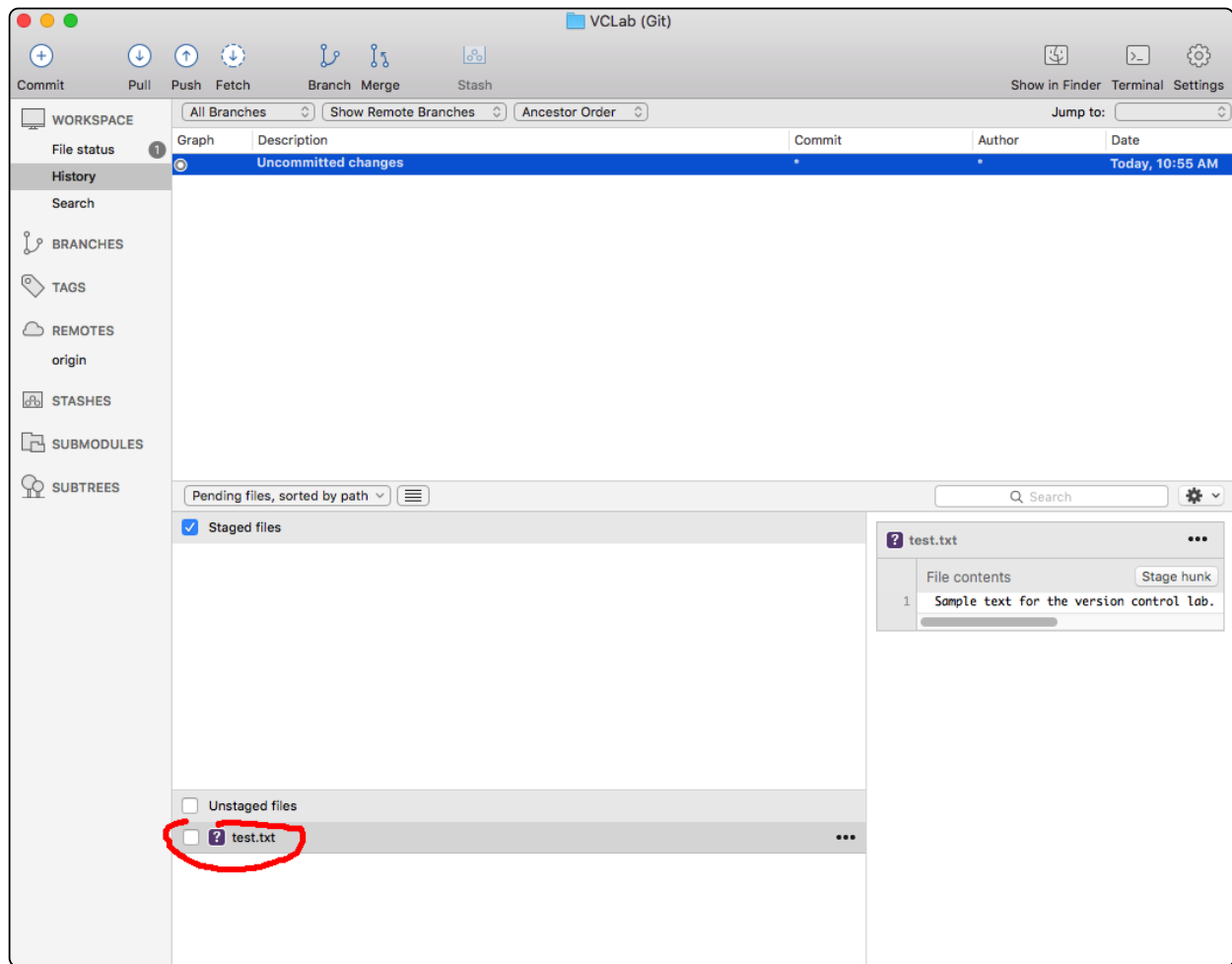


- You can click on the "Open In Finder" button and see an empty folder. In fact, the folder is not empty, it contains ".git" directory, which is hidden on macOS.
- In SourceTree, select the "History" view (see the screenshot below) to see the history of commits. Nothing should be there, as the repository is empty.

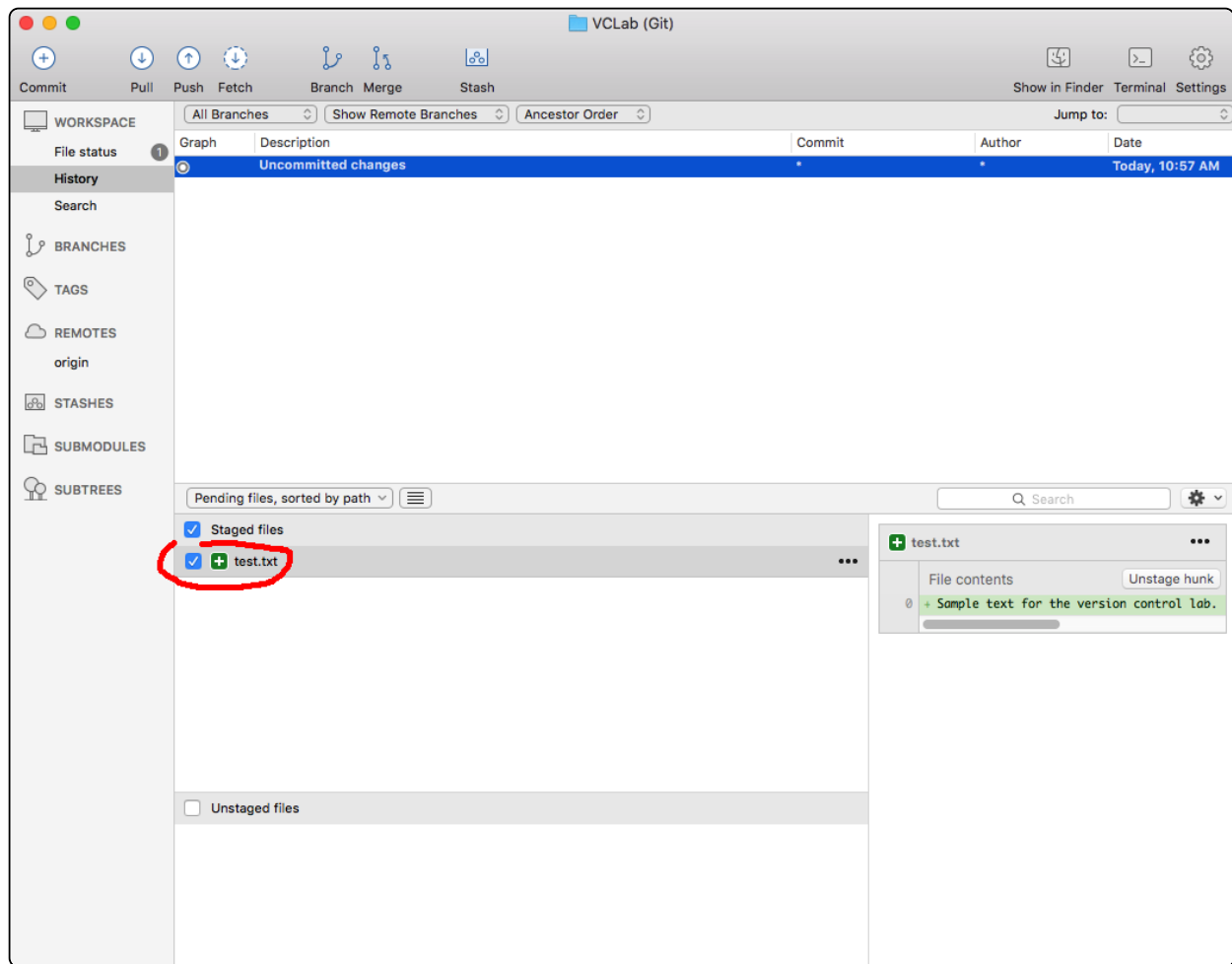


## Add content to the repository

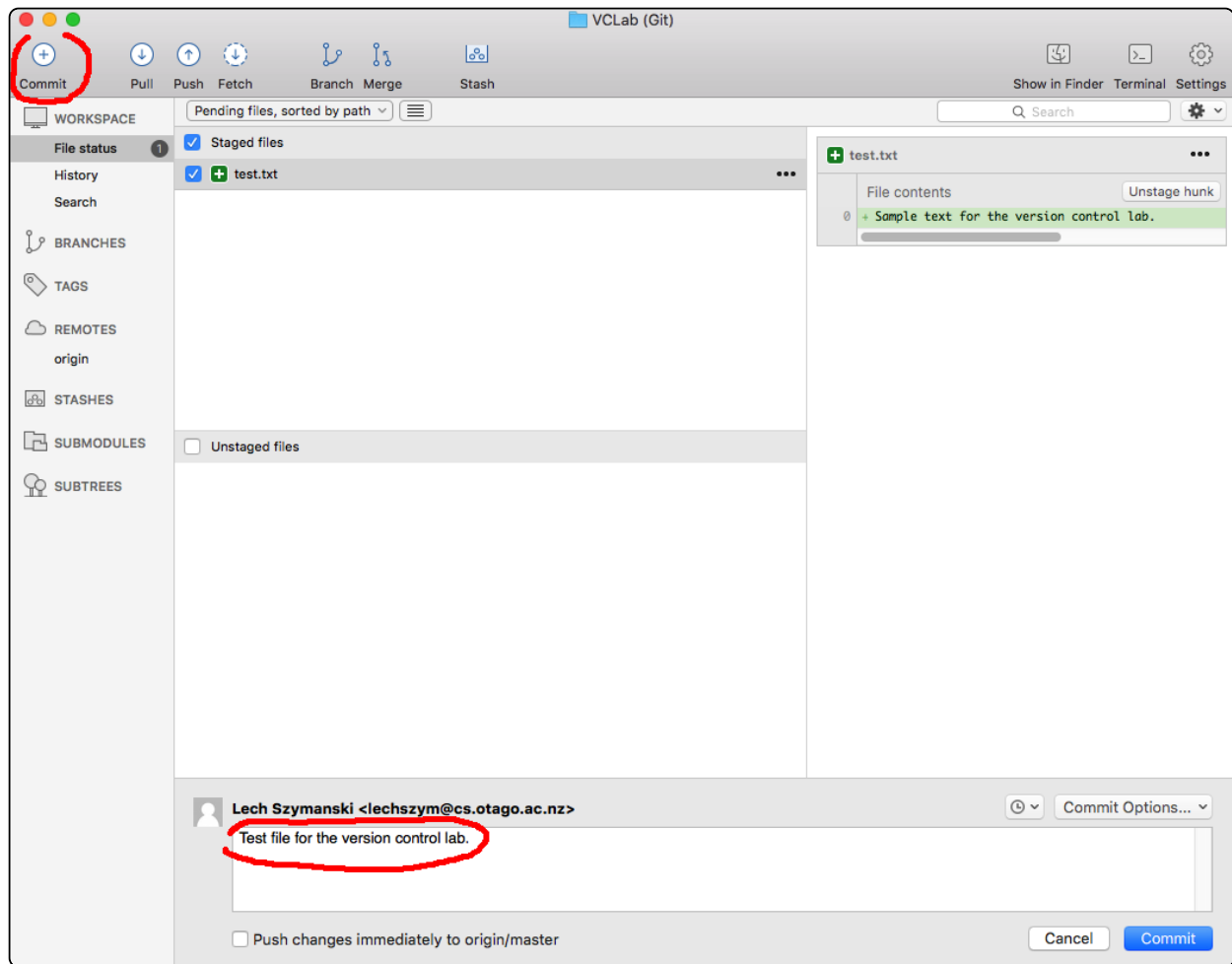
1. Create a text file in your favourite text editor. Put some text inside the file and save it into the "VCLab" directory (the directory where your repository was cloned) as "test.txt".
2. Back in SourceTree, you should see the newly created "test.txt" under "Unstaged files". The purple-question mark icon means it's a new file, not tracked by the repository at all.



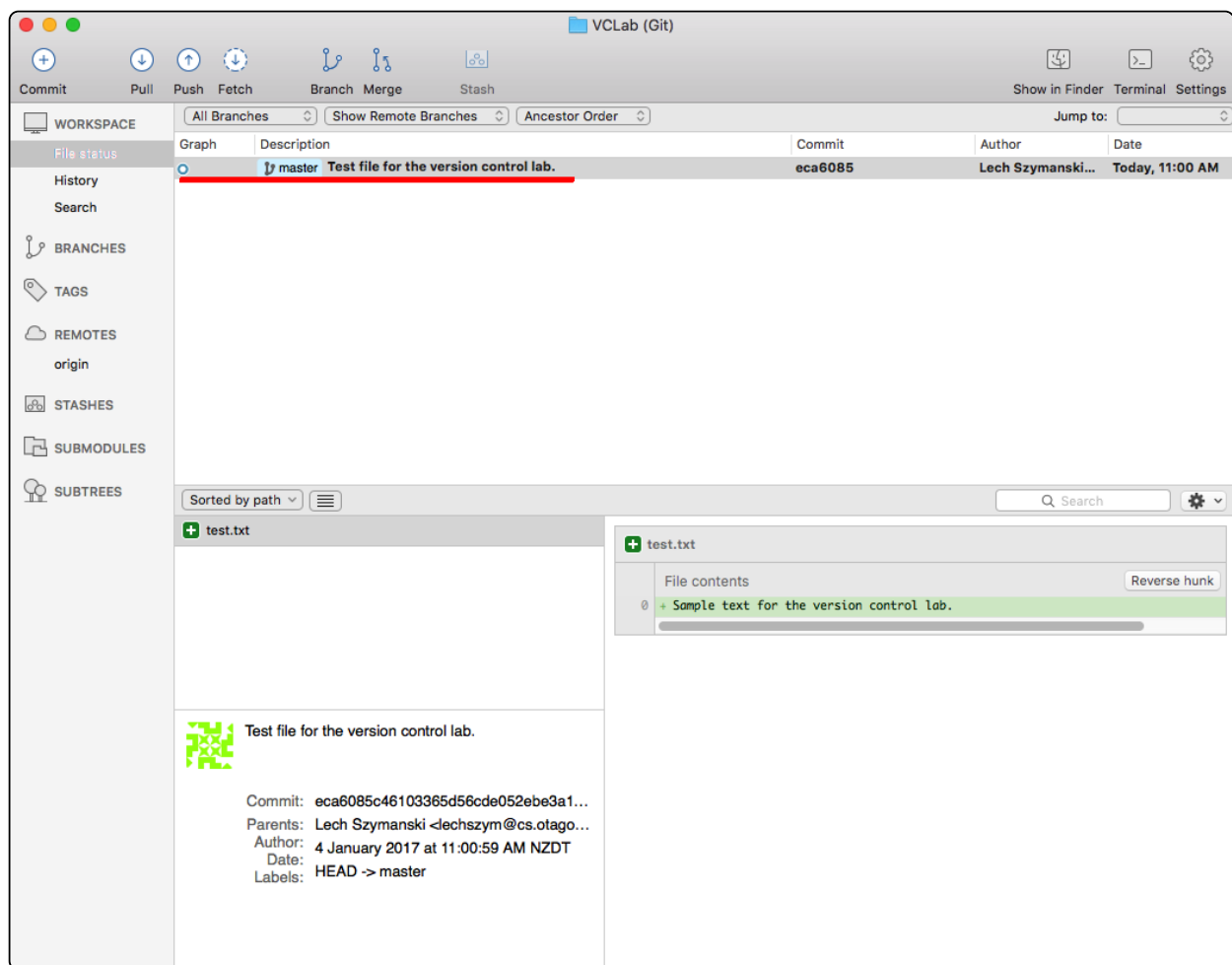
1. In Git, you need to *stage* files to prepare them for a commit to the repository (whether the file is completely new, or just modified). In other words, you have to select files that will go into the repository with the next commit. Right now, you want to commit that "test.txt" file, so that git starts tracking it. Click on the check-box next to it. The file should move to the "Staged files" pane.



5. Time to commit the staged file. Click on "Commit" in the main toolbar (top-left-hand side). A pane will open at the bottom where you can put a comment. A comment must accompany a commit. A good practice is to make these comments brief and informative.



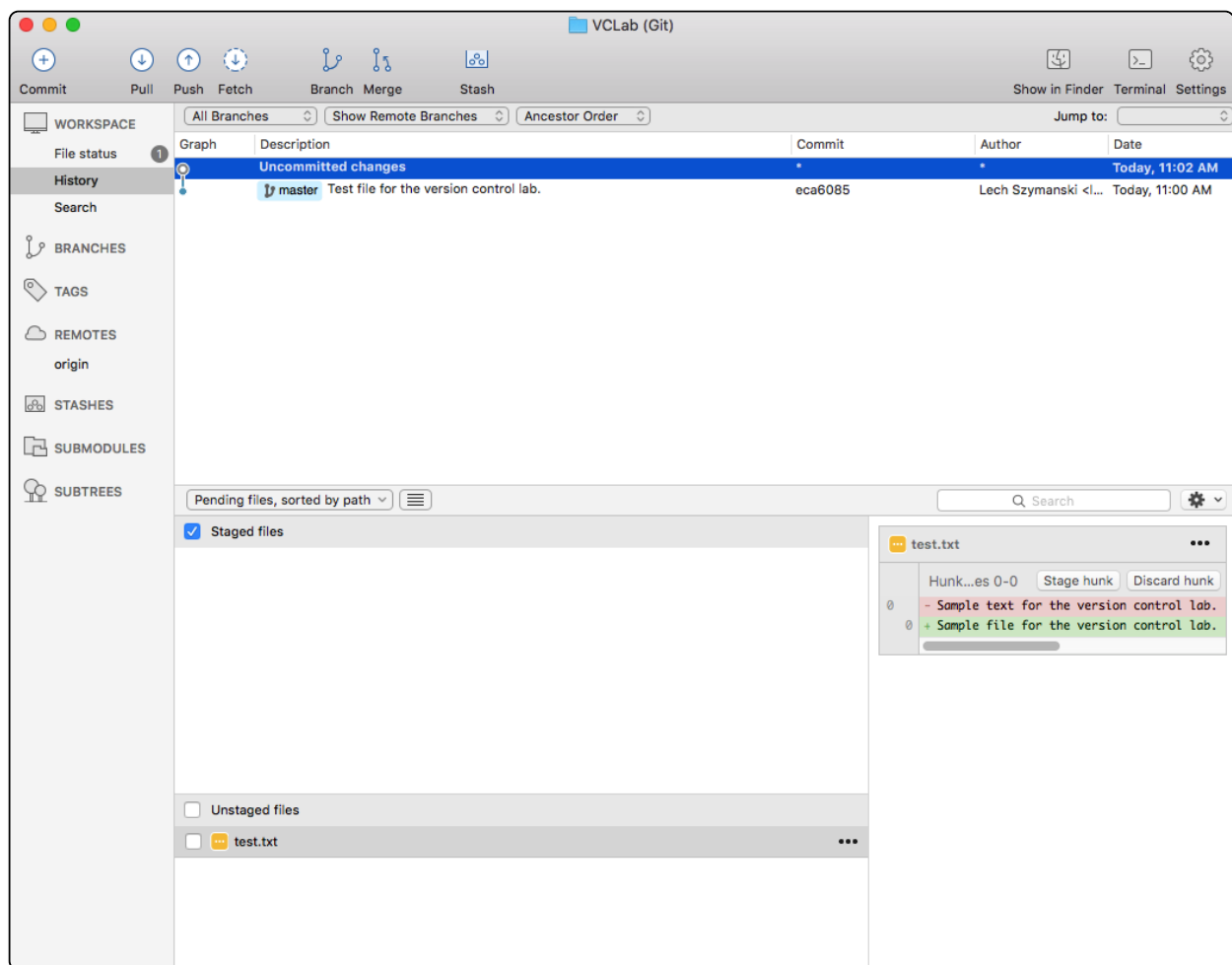
3. Press the "Commit" button (bottom-right-hand side). SourceTree should return to the timeline view and show the commit. You've committed the new file to the "master" branch, which is indicated by the tag preceding the commit message. There is also a commit number - that's a unique ID associated with this commit. You can always revert the repository to a given commit using that ID number.



## Update files

1. Modify the contents of "test.txt" and save the file. In the "History" view SourceTree should indicate that there are "Uncommitted changes", meaning you've created new files or modified the ones that are being tracked by the repository (the latter in this case).
3. In the "Unstaged files" you'll see the modified file - the orange icon means it's a file that is already tracked by the repository. You can even see the changes you made in the window on the right-hand side. Stage the file.





3. Commit the staged file.

---

4. Once you commit, another entry to the timeline is added. Not only can you see the changes, you can always come back to a version of a file from a previous commit - it's surprising how often such a need arises.

---

## Pushing changes to GitLab

1. The repository you've been committing to so far is located on your hard-drive. In order for others to see those changes, and perhaps to allow them to incorporate those changes into their work, you need to push your changes to a central repository - the one hosted on GitLab. The GitLab repository, that you cloned from, is referred to as "origin". The cloning process creates a link to the origin, so you can push back your changes. At the moment, the origin is still empty, whereas your local repository contains two commits - one that added "test.txt" to the repository, and one that changed the contents of "test.txt". The figure below shows a visualisation of the commit histories of the two repositories.

---

2. If you take a look at the VCLab project on GitLab, it should still maintain that the project is empty.

---

3. In SourceTree, in the window that is tracking the history of your local VCLab repository, click on the "Push" button.
- 

4. A window will pop up with the details of the push. You're pushing to "origin", which is the repository you cloned from. Also, make sure to select "master" as the branch to push to. A repository may have multiple branches - "master" is the default main branch, which typically everyone will synch up with. Press "OK"
- 

5. Once the push is finished in SourceTree, go back to GitLab's VCLab project page in your browser. It should now specify two commits and 1 branch. The repository is not empty anymore - it's been synched with your local one. The history of the two repositories is now identical.
- 
- 

6. Take a look at your GitLab project's *Repository->Files* - "test.txt" should be there. If someone clones VCLab now, they will get the latest version of the "test.txt" file that you committed to the repository.
- 

7. Take a look at your GitLab project's *Repository->Commits* - your two commits with commit messages should be listed there.
- 

8. Take a look at your GitLab project's *Repository->Graph* - the history tree of your commits should be listed there.
- 

## Adding a new Unity project to the repository

To practice a bit more, you're going to create a new Unity Project and add it to the VCLab repository.

9. Open Unity and create a new 2D project. Call the project "UnityGame" and save it to the VCLab directory on your hard-drive - that should be somewhere in the */scratch* directory.

### IMPORTANT!



Check that your new Unity project is configured properly for storing in a repository. You might have to change a number of settings in order to make Unity a bit more git-friendly. It all boils down to how it saves its data. Unity can save scenes in binary mode, which is bad for tracking changes. You need to make sure that Unity is set to save things in text mode. If this setting is not correct you will be in a world of pain when collaborating on the project with other members of your team, as git will be unable to seamlessly merge changes from multiple developers. This configuration has to be checked once. Once it's done, and the project is in the repository, it will remain in this configuration for other team members when they clone the repository and open the project on their computers.

**DON'T FORGET TO CHECK THESE SETTINGS LATER WHEN YOU CREATE THE UNITY PROJECT FOR YOUR GAME!**

- In Unity, from the main menu, select *Edit / Project Settings / Editor*
  - In the **Inspector panel** set the "Version Control | Mode" to "Visible Meta Files" and "Asset Serialization | Mode" to "Force Text".
- 

1. In Unity, create a "Scenes" folder in "Assets" and save the current scene the (with just the "Main Camera" in it) as "Level1".
  2. Take a look at VCLab repository in SourceTree. Notice that there are lots of new files (about 145 or so) in the unstaged window. However, you won't be adding all of them. Some of those files you don't want in the repository - they have to do with personal settings rather than game content. You probably have your own way of working in Unity, and your mates might have a different way - you don't want to have your Unity workspace reconfigured every time you synch your repository.
- 
2. You can tell git to ignore certain files (or directories) - this way SourceTree will stop listing in the unstaged list those files that you don't want to be committed to the repository.
  3. Create file ".gitignore" in the root folder of your Unity game project (if you followed the instructions exactly it will be the *VCLab/UnityGame*).
  4. Paste the following text into ".gitignore" and save it.

```
# This .gitignore file should be placed at the root of your Unity
  project directory
#
# Get latest from
  https://github.com/github/gitignore/blob/master/Unity.gitignore
#
/[Ll]ibrary/
/[Tt]emp/
/[Oo]bj/
/[Bb]uild/
/[Bb]uilds/
/[Ll]ogs/
/[Mm]emoryCaptures/

# Asset meta data should only be ignored when the corresponding asset
  is also ignored
!/[Aa]ssets/**/*.*meta

# Uncomment this line if you wish to ignore the asset store tools
  plugin
# /[Aa]ssets/AssetStoreTools*

# Autogenerated JetBrains Rider plugin
[Aa]ssets/Plugins/Editor/JetBrains*

# Visual Studio cache directory
.vs/

# Gradle cache directory
.gradle/

# Autogenerated VS/MD/Consulo solution and project files
ExportedObj/
.consulo/
*.csproj
*.unityproj
*.sln
*.suo
*.tmp
*.user
*.userprefs
*.pidb
*.booproj
*.svd
*.pdb
*.mdb
*.opendb
*.VC.db
```

```
# Unity3D generated meta files
*.pidb.meta
*.pdb.meta
*.mdb.meta

# Unity3D generated file on crash reports
sysinfo.txt

# Builds
*.apk
*.unitypackage

# Crashlytics generated file
crashlytics-build.properties

# ===== #
# OS generated #
# ===== #
.DS_Store
.DS_Store?
._*
.Spotlight-V100
.Trashes
Icon?
ehthumbs.db
Thumbs.db
.vscode
```

5. The "Unstaged" list in SourceTree should have now about 20 files. Notice that *UnityGame/.gitignore* is among them. You will be committing *.gitignore* too.

---
6. If *.DS\_Store* is still lingering around, right-click on it, select "Ignore" and "Ignore the exact filename". This will create another *.gitignore* in the root folder of your repository and *.DS\_Store* should disappear (and another *.gitignore* will show up).

---
7. To stage all those new files, click on the checkbox next to "Unstaged files". Type in a commit message and commit.

---
8. Notice the "1" that appeared over the "Push" button in SourceTree. It signifies that your local repository has 1 commit that is not in the origin.

## Modifying a scene

9. Back in Unity modify the "Level1" scene. For instance, you can create a folder called "Sprites" in "Assets", drag *planet.png* (*images/planet.png*) and use it for a sprite game object in the scene. Save the scene.

---

2. In SourceTree, you should now see a set of "Uncommitted changes". Three of them have to do with creation of new folder in "Assets" and addition of a new file, "planet.png". Since "Level1.unity" has been committed previously, it should show as a modified file.
- 

1. Stage all the files and commit them to the repository. Notice that the "Push" button has a number "2" over it - your local repository has two commits that have not been pushed to the origin.
- 

## Multiple clones

Before you push the changes to the origin, let's see what would happen if someone else cloned VCLab from GitLab at this moment.

2. Close the VCLab project in SourceTree - you should see the main SourceTree window. Select *New Repository / Clone From URL*.
  3. In the window that pops up, paste the SSH URL of the VCLab project from GitLab for the "Source URL". For the "Destination Path" specify a location somewhere in the */scratch* directory. However, make sure the last part of the "Destination Path" says "VCLab2" - it's going to be a completely separate copy of the VCLab repository.
- 

4. Once the repository is cloned, SourceTree should indicate that there is a "VCLab" as well as "VCLab2" repository on your hard-drive. They are two separate clones of the "VCLab" repository from GitLab. Normally, there is no point in doing this, but we are simulating a situation when someone else clones the same repository.
- 

5. In SourceTree, open the "VCLab" and "VCLab2" repositories side by side. "VCLab" should have 4 commits and lots of Unity files, whereas "VCLab2" only 2 commits and only "test.txt". That's because the last two commits from "VCLab" have not been pushed to the origin, and so the new clone does not have them.
- 

6. Push the changes from "VCLab" to the origin. Now, the Unity files are in the repository on GitLab, but not in "VCLab2", because "VCLab2" has not been synched with the origin yet.
- 

7. To synch up "VLab2" you need to "Pull" from the origin. The "Pull" command fetches all the changes from the origin and updates your files on the hard-drive.
- 

8. Now "VCLab2" should show 4 commits, and all the Unity files are there in the "VCLab2" directory.
- 

## Changing things at the same time

Let's imagine "VCLab" and "VCLab2" are actually on different computers, and two developers are working on the project at the same time.

3. Get back to the "UnityGame" project from "VCLab" in Unity. Create a folder called "Scripts" in "Assets" and create "Planet.cs" there with the following content.

## Planet.cs

```
01: using UnityEngine;
02: using System.Collections;
03:
04: public class Planet : MonoBehaviour {
05:
06:     // Rate of the 'bob' movement
07:     public float bobRate;
08:
09:     // Scale of the 'bob' movement
10:     public float bobScale;
11:
12:     // Update is called once per frame
13:     void Update () {
14:         // Change in vertical distance
15:         float dy = bobScale * Mathf.Sin(bobRate * Time.time);
16:
17:         // Move the game object on the vertical axis
18:         transform.Translate(new Vector3(0, dy, 0));
19:     }
20: }
```

4. Add "Planet.cs" as the script component to the "planet" game object in the "Level1" scene. Set the "Bob Rate" to 0.8 and "Bob Scale" to 0.005. Save the scene.
- 

1. If you run the game, the planet will bob up and down.
  2. In SourceTree commit all the new and changed files to the "VCLab" repository.
- 

3. Close the current project in Unity and open "UnityGame" from "VCLab2".
  4. Load "Level1" scene.
  5. Drag star.png (images/star.png) to *Assets / Sprites* and add a star sprite game object to the scene. Do not modify the *planet* game object.
- 

6. If you play the scene, the new star object will be there, but the planet will not move - remember, the script has been added in a different project, residing in the "VCLab" directory.
7. Save the scene in Unity and in SourceTree, in the "VCLab2" repository, commit all the new and changed files.

- 
3. In SourceTree, push "VCLab" changes to the origin. Right after that, pull the changes in "VCLab2".
- 

3. What happened? "VCLab" changes went to the origin. "VCLab2", after the pull, incorporated those changes into its project. The timeline shows the pulled changes (in pink) as a "origin/master" branch, and the commits made since the last sync, as the "master" branch. However, git was able to merge them back into one timeline, and the result is a new point in history, the "Uncommitted changes" in "VCLab2", which contain all the commits made in "VCLab2" and the changes that came through the merge with "origin/master".
  3. Switch back to Unity. The "UnityGame" from "VCLab2" should be still open. If you're prompted, reload the files. If you play the game now, the planet bobs up and down. All the changes from "VCLab" have been incorporated into this project.
  1. In SourceTree, in "VCLab2", commit the changes due to merging of the changes pulled from the origin. Because these changes are due to a merge, you don't need to stage anything, just go ahead with the commit. There even should be a default commit message describing the merge - you can leave it there if you like. Commit.
  2. The history of the merge should be visible in "VCLab2"'s timeline. Push "VCLab2" changes to the origin.
- 

3. In SourceTree, pull the changes in "VCLab". It's should get all the changes pushed from "VCLab2" - including the merge.
- 

4. At this point you can also open the "UnityGame" project from "VCLab" and it will have the star added in "VCLab2". Both projects are synced.

## When does this work and when does it not?



In this example, despite the fact that the same scene was changed independently in two separate projects, the synchronisation worked perfectly and the merge was seamless. This doesn't always work out so nicely. Why did it work this time? It worked because the changes were made on independent parts of the scene. If, for instance, the *planet* game object, that's already in the scene was changed in "VCLab" and "VCLab2", git would get confused. Merging is pretty easy when different aspect of the file are changing - in this case, different assets are added to the project, and different parts of the scene are modified. However, if the same game object in the scene was modified independently, there would be no way for git to know how to reconcile those changes. To avoid this problem:

- Do not modify the same aspects of the project. Working on the same scene may be fine, but modifying the same game object in the scene is not advisable.
- Do not modify the same asset at the same time. Most assets are saved in binary format, and there is no way to reconcile the changes.
- Do not modify the same lines of code in a script at the same time. Git will cope well with independent changes in the same file, but not the same lines.



- **Always commit to your local repo before pulling!** If changes come from the origin, even if they overwrite your work, you'll be always able to revert to a previous commit.
- **Always pull before pushing!** In fact, git will not allow a push if the origin is ahead of your local repo. Basically, it forces you to do a merge of the changes from the origin before you can commit. Of course, if no one has changed anything in the origin, the pull will not result in any changes.
- Even when the automatic merge doesn't work out, you can still recover. First of all, if all your work has been committed to the local repository, if the worst comes to the worst, you'll be able to retrieve it. But it might not be necessary - quite often, when git cannot do the merge, you can still go over the files, scan for conflicts and fix them manually.

## Dealing with conflicts

The merge above went very smoothly. Unfortunately, this doesn't always happen. Quite often people will make independent changes to the same lines of code, or introduce conflicting changes to a scene, or prefabs or other assets. This may create conflicts which git is unable to resolve automatically, thus requiring user intervention. The following is a conflict-resolving exercise to give you an idea how and what goes into conflict resolution of a merge.

5. The "UnityGame" project from "VCLab" should be still open in your Unity Editor. If it's not, then open it.
  3. Let's assume you're working on a set of new features - a new look for the star asset and a change so that the planet bobs in a horizontal instead of vertical direction.
  7. Drag sun.png (images/sun.png) to *Assets / Sprites*
  3. Change the "star" object's "Sprite Renderer"'s "Sprite" property to "sun". That sprite is quite large, so set the "Scale" of the object's "Transform" to (0.2,0.2,1).
- 

3. Save the scene and in SourceTree commit all the changes to the "VCLab" repo.
- 

0. Now to change the movement of the planet, modify the "Planet.cs" script. In the **Update ( )** method rename the **dy** variable to **dx** and change the translation so that the object moves in the horizontal direction. VCS editor is git-aware and so you can view the changes you have made with respect to the "HEAD", or the last version in the repository.

The code on the left is the old version (that was committed to the repository) and the right-hand side shows the version with the changes you just made (still not committed to the repository).

1. In SourceTree commit all the changes to the "VCLab" repo.
- 

2. Meanwhile, let's suppose that your teammate is working on a different feature - planet movement that takes place only when spacebar is pressed. In the Unity Editor switch to the "UnityGame" project from the "VCLab2" folder.

3. For whatever reason, for testing perhaps, at some point the "VCLab2" developer changed the size of the Scale of the star object to (2,2,1) and then disabled the object altogether. That object is not essential to the new feature, and so the developer doesn't pay attention to it after those changes were made. Make those changes to the scene.
- 

1. Then, the "VCLab2" developer implemented the spacebar feature, which consists of a change to the "Update()" function of "Planet.cs" script as shown below - left-hand side shows the original code, right-hand side shows the new code. Make the corresponding changes in "Planet.cs" in the "VCLab2" directory.

3. Now switch to the SourceTree window showing the "VCLab2" repository. Check in all the changes. Note that changes to "Level1.unity" (the scaling and disabling of the star in the scene) are not essential for the spacebar feature...and a careful developer would probably not check them in at all. However, in this exercise we want to see a conflict, and so everything gets checked in.
- 

3. Let's assume "VCLab" was the first one to push their changes. In SourceTree *push* the changes to the origin from the "VCLab" repo.
- 

7. Now, because the GitLab repository, or the "origin", has just been changed, "VCLab2" developer will not be able to *push* his/her changes, unless they *pull* first. So, as a "VCLab2" developer, try to *pull* from the origin in SourceTree.

3. The screenshot below shows what happens to "VCLab2" after the pull. The pink time-line is "VCLab2"s history, including the most recent changes, and the blue line are the changes introduced by "VCLab". Git once again tries to merge everything, but it's unable to do it automatically. The files that git cannot merge on its own, "Level1.unity" and "Planet.cs", as marked as being in conflict. "VCLab2" developer must resolve the conflicts before pushing changes to the origin.
- 

3. If you switch to the Unity Editor, which should still have "VCLab2"s project open, it will complain about not being able to load the scene. That's because git stores the information about the conflict inside the conflicted files using a special mark-up that breaks the syntax and Unity doesn't understand it. The conflict must be resolved in order to restore "Level1.unity" to a valid state.

3. Let's suppose that the "VCLab2" developer is aware that his/her changes to the scene are superficial and the changes made by "VCLab" are the ones that configure the scene correctly. In other words, the "VCLab2" developer wants to resolve the conflict in "Level1.unity" by taking the version that's coming from the "origin", discarding all the changes that have been made in his/her repository. In SourceTree right-click on conflicted "Level1.unity" and select "Resolve Conflicts->Resolve Using 'Theirs'".
- 

Take a note of the "Resolve Using 'Mine'" option - it allows "VCLab2" developer to resolve by reversing the direction of the overwrite, ignoring changes that are coming from the "origin" and using the scene as it is in the local repository.

1. After this resolution, the Unity Editor should not complain about the scene anymore...but it will still complain about "Planet.cs". Again, because this file is in conflict, it contains git syntax marking the conflicted areas, which the C# pre-processor doesn't understand. For this conflict you will practice manual resolution. Open "Planet.cs" in the VCS editor.

The markup for the conflicts is as follows. The section starting with "<<<<<< < HEAD" and ending with "=====" shows the section of the file with changes that are coming from the local repo - corresponding to "mine". The following block, starting with the same "=====" and ending with ">>>>>> <sha number>" are the changes that are coming from the "origin" - corresponding to "theirs". There may be more than one block like this in a text file and you have to resolve all those conflicts.

Notice that in this case you cannot just simply take the "mine" or "theirs" block, as git has automatically merged some other changes which rely on the existence of both **dy** and **dx** variables. In order to resolve this conflict, you need to understand what the code does, and make sure that the resolution produces a working code. In this case, it makes sense to change the variable name to **dx** as "VCLab" developer intended (since movement is now horizontal), but keep it initialised to 0, as the "VCLab2" developer intended. The resolved file is shown below. It would probably make sense to change the comments as well, since they refer to vertical instead of the horizontal movement.

2. Now that the git diff syntax is removed, you can test that the script works in Unity. The planet should bob horizontally, but only if the spacebar is pressed.
  3. Though git's syntax has been removed from "Planet.cs", git still needs to be told that conflict has been resolved. You do this by staging the conflicted file. Go ahead and stage "Planet.cs" in SourceTree.
- 
4. Commit the changes to the repository. SourceTree should generate a descriptive message describing the merge and listing the files that were in conflict.
  5. Next you can push everything to "origin".
  3. And now the "VCLab" repo can "pull" from the "origin". Everyone is synced up, gaining the new spacebar feature developed by the "VCLab2" developer as well as the new asset and planet movement done by the "VCLab" developer.

## Sync tips



- If you don't have a lot of experience with git, it's not a good idea to develop completely independently for long periods of time, and then try to merge massive amounts of content when putting the final game together. It always ends in grief, as there are hundreds of conflicts. Instead, try to break the features into smaller chunks that can be merged more often.
- You can't merge binary files - if a binary file is changed independently by two parties, you will have to resolve conflict using "mine" or "theirs", thus losing one set of changes.
- **Always pull before pushing!** - git will merge any changes that are out of sync into your checked out code.

- **Always test that everything works after the merge** before you commit merge changes to your local repository. Run the game in Unity and make sure that all conflicts are resolved and everything runs as it should.
- If you think a pull might produce a complicated merge, make a backup of your repo first - this way you can always abandon a merge that went wrong and try again from before the merge.

## Assessment

Show your work to the demonstrator for assessment.