

Polish and Build

Rishav, Omang

March 10, 2025

Contents

1	Introduction	2
2	Git Work Flow	2
2.1	Creating Repository	2
2.1.1	Initial Setup	3
2.2	Inviting your Team and Cloning the Repository	4
2.3	Creating Feature Branches	5
2.4	Unity Files with Git	6
2.4.1	Scenes and Prefabs	6
2.4.2	C# scripts	7
2.5	Creating Pull Request	7
2.6	Updating From Main	8
2.7	Unity-Git Workflow	10
2.8	Resolving Merge Conflicts	11
3	Features	13
3.1	Overview of the project	14
3.2	Score	14
3.3	Lives & Game Over Screen	15
3.4	Audio Manager & SFX	15
3.5	Particle Effects	15
3.6	Camera Shake	15
4	Building for WebGL	16
4.1	Preparing the Build	16
4.1.1	About Scene List and SceneManager	17
4.1.2	Maintaining Consistent Resolutions for UI	18
4.2	Exporting to Itch	19
5	Conclusion	22

1. Introduction

This studio will be a bit different from the previous ones. You'll be working in your Game Jam teams to implement individual features while collaborating via Git and GitHub. To start, you'll receive a basic prototype of the classic retro game Breakout (See the [git repo here](#) and [play the base prototype here](#)). The goal for this studio is for each person in the team to implement a simple unique feature. For example, a team of three should complete three distinct enhancements. Each feature focuses on enhancing gameplay with "game juice," as outlined in Section 3. Your final submission will be a link to an [itch.io](#) build for the improved game, along with the link to your GitHub repository.

You'll first learn the key steps for managing a Unity project with multiple contributors using Git. **Make sure to properly study this portion of the document to avoid major mishaps for your final submission, which is worth 50% of the grade!!** Next, we'll provide some suggestions for quick upgrades for improving the game feel of the breakout prototype, such as adding particle effects and camera shake. Note that we will not be providing direct implementations anymore. At this stage of the course, you should have a decent idea of the different components of Unity, and be capable of searching for how to implement specific features. We will leave it up to you on which features you wish to implement, including if you want to modify something that's not in the list, as long as it qualifies as adding to the "game juice" (see [here](#), [here](#) and [here](#) for references on what qualifies as game juice).

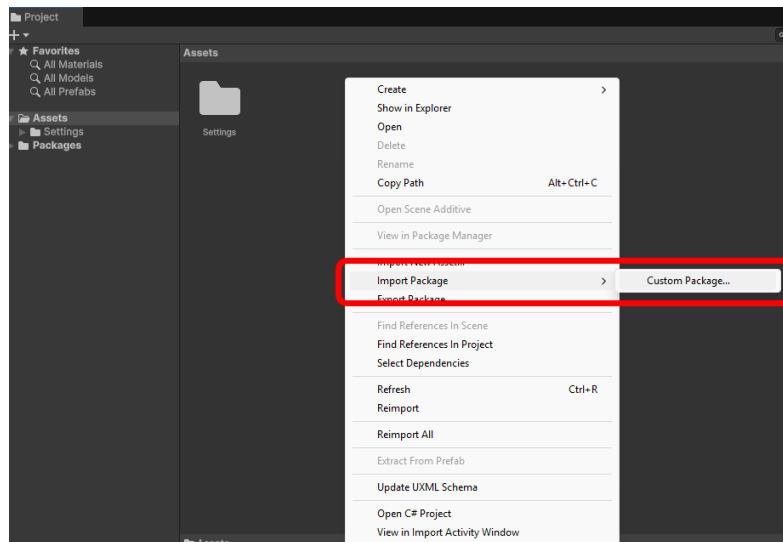
2. Git Work Flow

Before getting started with implementations, everyone in the team should read through this section thoroughly. Once you have a good idea of how to collaborate with your team with Git and GitHub, you can go to Section 3 to focus on implementations, and reference this portion of the document whenever you are ready to push a completed polish upgrade.

2.1. Creating Repository

First, one person in the team will create the base repository. Create a repository for your teammates based on the [New Project Setup Guide](#). **Once the initial commit is complete**, delete every file and folder in it except for the "Settings" folder. Download this [Breakout.unitypackage Unity package](#)¹. Finally, either drag and drop the asset into the project's asset folder, or right click in the **Project** tab > **Import Package** > **Custom Package**, and select the "Breakout.unitypackage" to import.

¹We opted for this approach rather than asking teams to fork the repo since maintaining forks can sometimes introduce strange behaviors when making pull requests unless things are configured properly. Since that's not the focus of this guide, we are instead distributing the base project via a Unity Package.



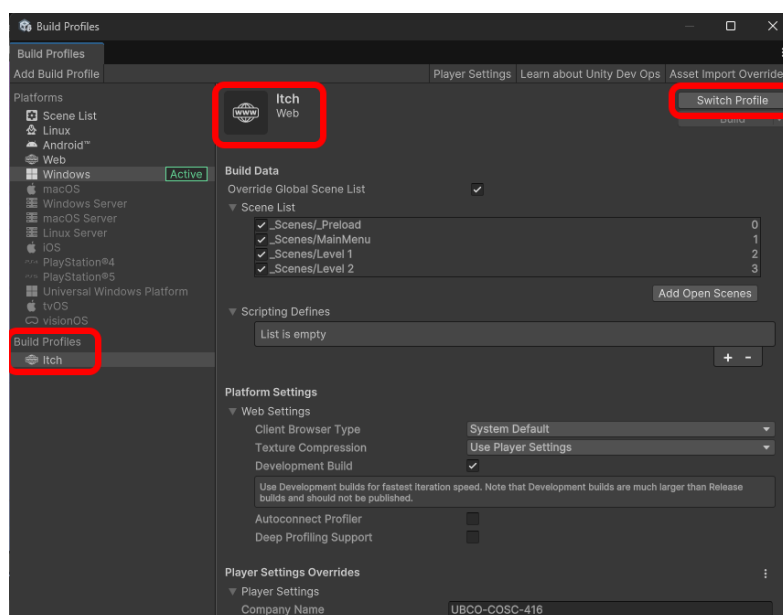
Make a Git commit here, something to effect of “Added Breakout base prototype”.

Tip

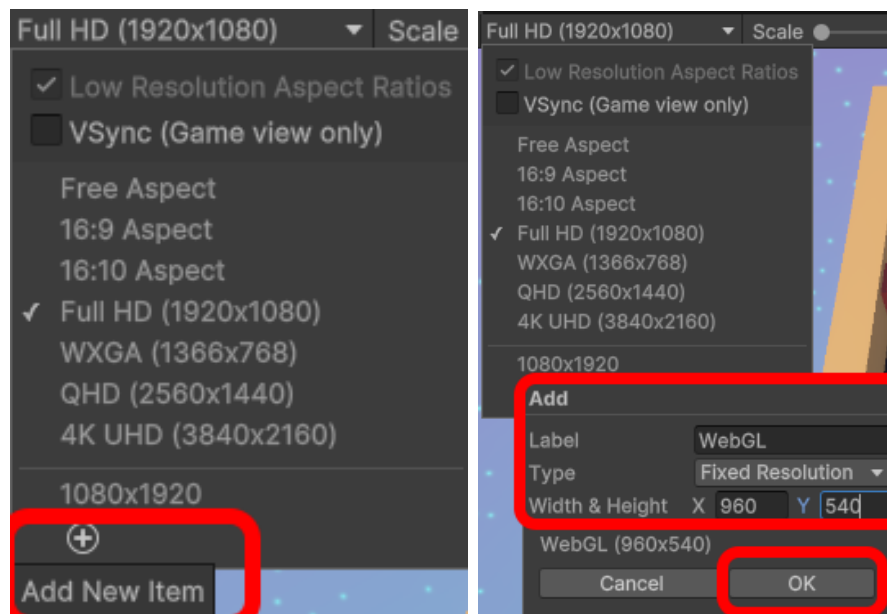
It’s worth noting that it doesn’t really matter who makes the base repository, and this applies for your final project as well. All team members will be contributing, and that will show in the contributors list and commit history. You can also add it to your pinned repositories even if it’s not made on your account for portfolio and showcasing reasons if that’s your jam!

2.1.1 Initial Setup

This project is configured for a WebGL Build. Consequently, a few things need to be configured for the base project to work. First, open **File > Build Profiles**. Under Build profiles, you should see a profile called Itch. Switch to this build profile.



This build profile is configured for the project with the relevant scenes already added to the Scene List, and a few other settings that we will discuss in Section 4. In order to ensure consistent UI, make sure the resolution of your game window is **960 x 540**. You can set a custom resolution on your game window, and name it something convenient (like WebGL or Itch) for reference.



Your final build will also be this resolution, and ensuring this is also something we will discuss in Section 4.

Note

These settings may have to be applied per person in the team. Make sure to check these two settings after cloning the repository on every team member's systems.

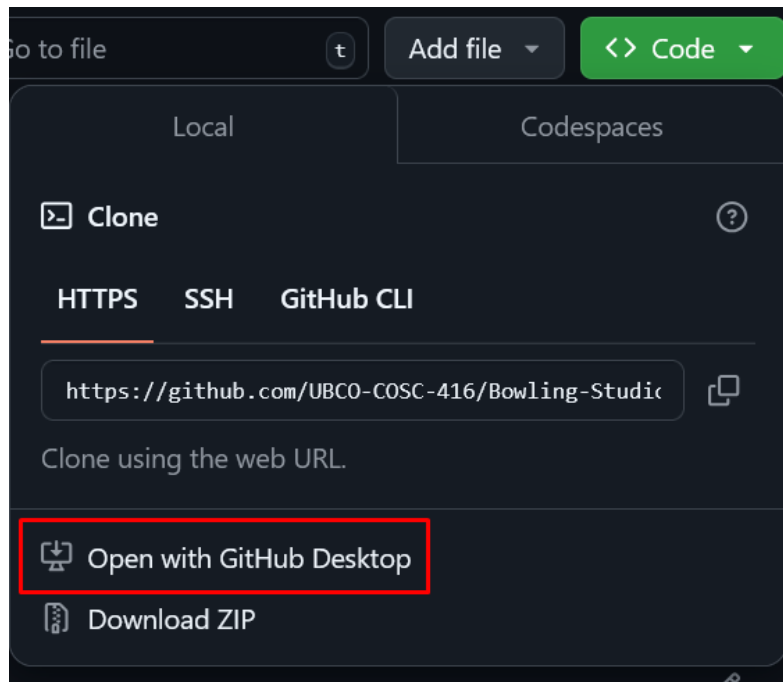
At this point, you can switch to the “_Preload.unity” scene, press play and go through the full game.

Note

If you are getting an error called “You may not pass in objects that are already persistent”, simply select from the top bar **Assets > Reimport All**. This will force Unity to refresh all assets, and the error should stop happening. It's a known Unity bug, and does not affect your game logic or the final build.

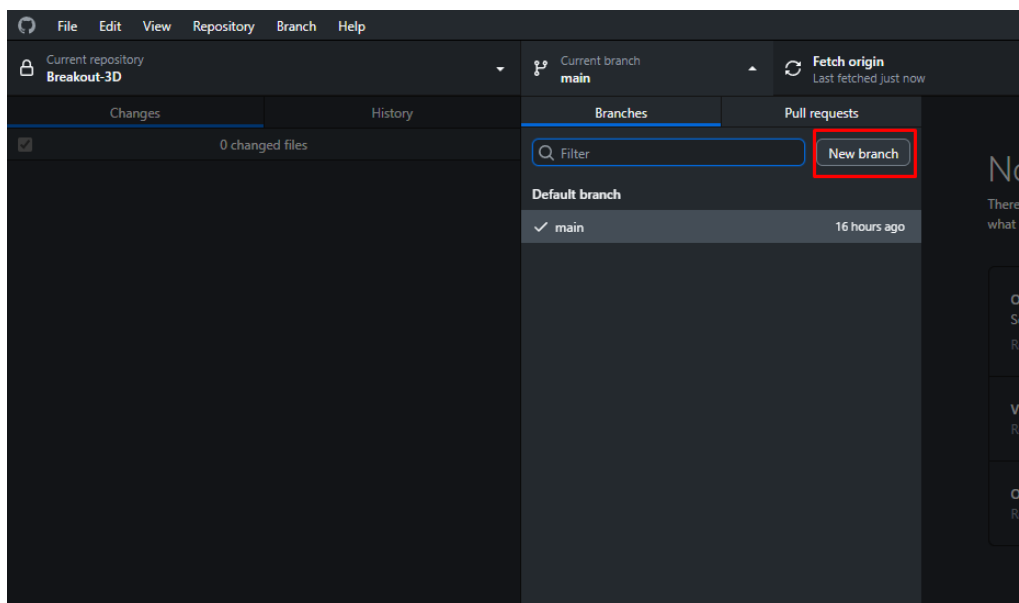
2.2. Inviting your Team and Cloning the Repository

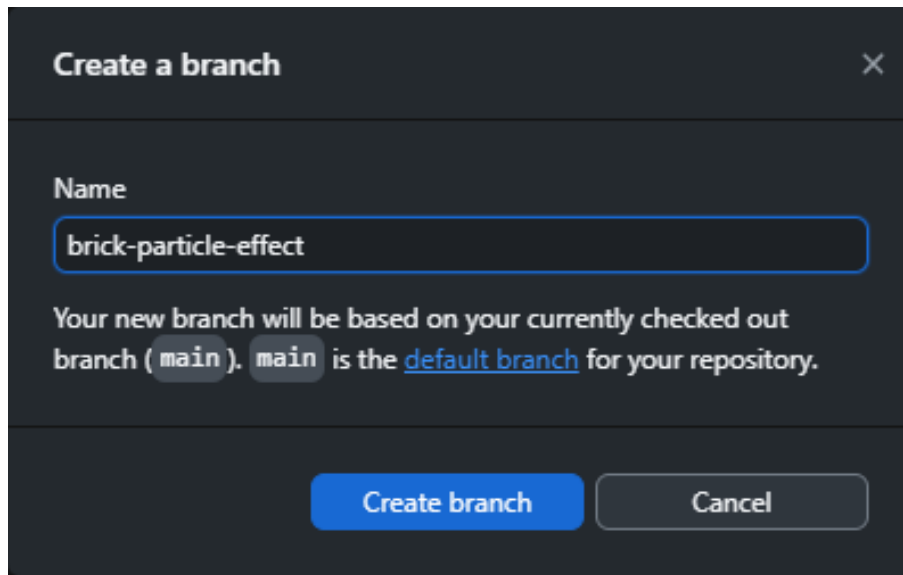
Next, invite the rest of your team members to the git repository. See [this GitHub resource](#) for more details on how to invite collaborators. You will need their GitHub user ID names, and send them invites to the repo. Each of the team members will receive an email associated with their GitHub account to join the repository. For the team members, accept this invite, and then clone the repository using GitHub Desktop.



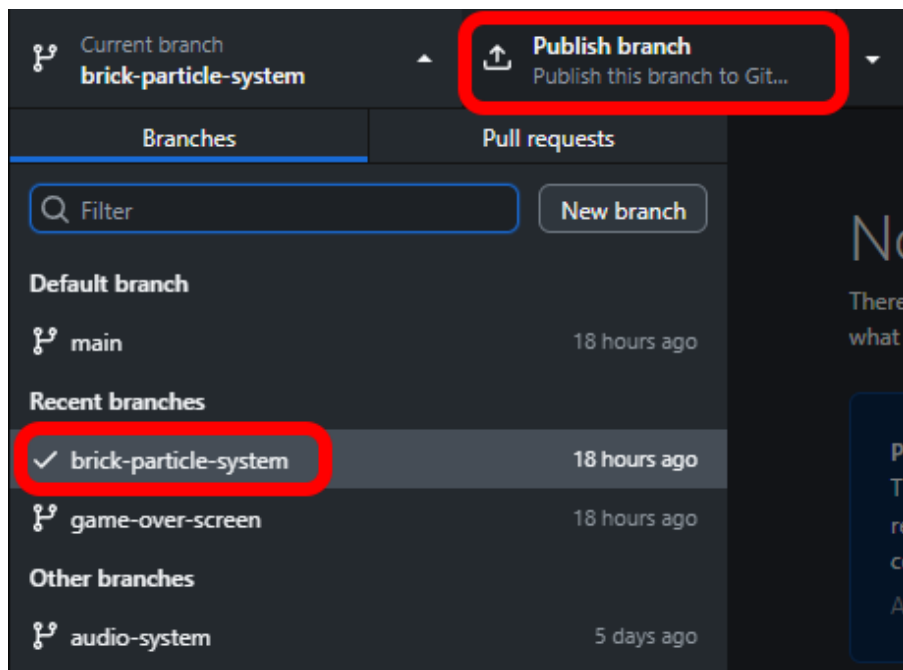
2.3. Creating Feature Branches

Now is the perfect time for your team to decide who will be responsible for each feature. Each teammate should create a separate branch for the features they are responsible for.





Make sure to name it something meaningful and related to the feature you want to implement. Make sure you switch to that branch, publish it, and then start working on the feature.



2.4. Unity Files with Git

2.4.1 Scenes and Prefabs

Unity Scenes (.unity) and prefabs (.prefab) store the scene/prefab configuration in the yaml format. If you open a unity scene in a text editor, this is how it would look like:

```

%YAML 1.1
%TAG !u! tag:unity3d.com,2011:
--- !u!29 &1
OcclusionCullingSettings:
  m_ObjectHideFlags: 0
  serializedVersion: 2
  m_OcclusionBakeSettings:
    smallestOccluder: 5
    smallestHole: 0.25
    backfaceThreshold: 100
  m_SceneGUID: 00000000000000000000000000000000
  m_OcclusionCullingData: {fileID: 0}
--- !u!104 &2
RenderSettings:
  m_ObjectHideFlags: 0
  serializedVersion: 10
  m_Fog: 0
  m_FogColor: {r: 0.5, g: 0.5, b: 0.5, a: 1}
  m_FogMode: 3
  m_FogDensity: 0.01
  m_LinearFogStart: 0
  m_LinearFogEnd: 300
  m_AmbientSkyColor: {r: 0.212, g: 0.227, b: 0.259, a: 1}
  m_AmbientEquatorColor: {r: 0.114, g: 0.125, b: 0.133, a: 1}
  m_AmbientGroundColor: {r: 0.047, g: 0.043, b: 0.035, a: 1}
  m_AmbientIntensity: 1
  m_AmbientMode: 0
  m_SubtractiveShadowColor: {r: 0.42, g: 0.478, b: 0.627, a: 1}
  m_SkyboxMaterial: {fileID: 2100000, guid: 20fca97cf5b8e92459331b3cd74c4be1, type: 2}
  m_HaloStrength: 0.5
  m_FlareStrength: 1
  m_FlareFadeSpeed: 3
  m_HaloTexture: {fileID: 0}
  m_SpotCookie: {fileID: 10001, guid: 00000000000000000000000000000000, type: 0}
  m_DefaultReflectionMode: 0
  m_DefaultReflectionResolution: 128
  m_ReflectionBounces: 1
  m_ReflectionIntensity: 1
  m_CustomReflection: {fileID: 0}

```

These files are not easily readable. Hence, to avoid merge conflicts, ensure that each team member edits **separate scenes and prefabs**. No two people should modify the same scene or prefab simultaneously, as this will lead to merge conflicts.

Note

Before modifying a scene/prefab, always let your team know. After making changes, commit & push immediately to avoid overlapping edits.

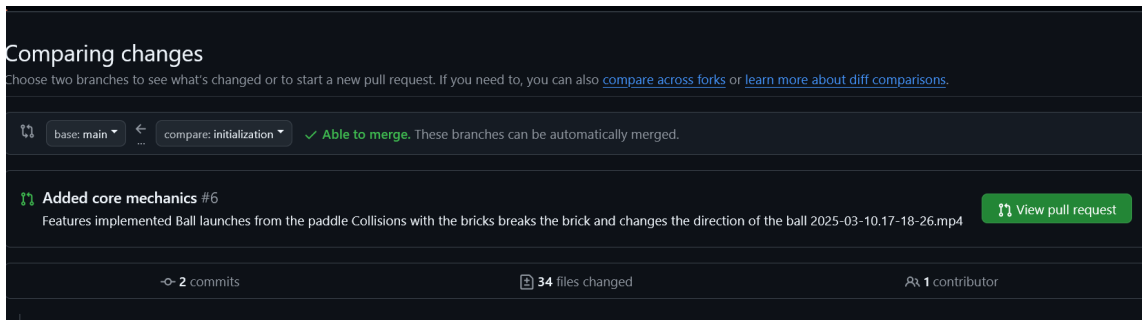
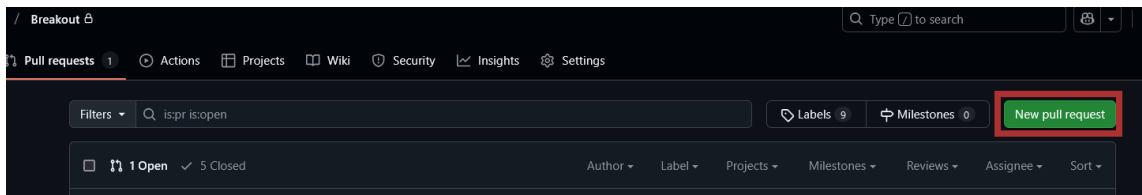
Aim to make all your components and prefabs as modular as possible. Once a feature is complete and ready for integration, add the necessary components to the final scene(s) and make sure all the references have been assigned.

2.4.2 C# scripts

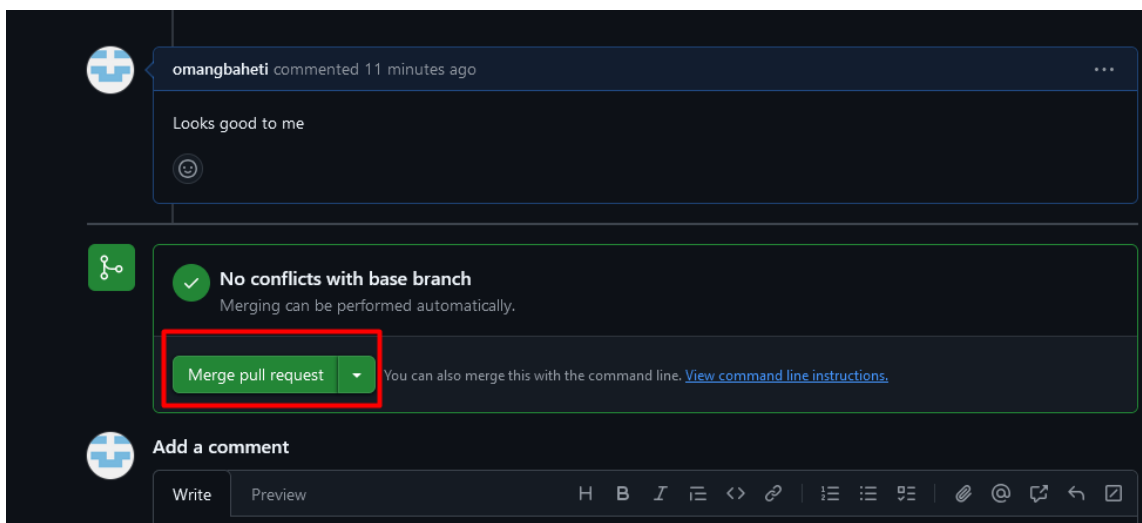
When it comes to C# scripts, multiple people modifying the same script is generally okay if you are working on different parts of the script (that's what git is built for). Even if you mess up, merge conflicts are easier to resolve as you/your teammates are aware of the parts of script that are relevant for the game. Section 2.8 goes into details about how to resolve merge conflicts for .cs files.

2.5. Creating Pull Request

Once a feature is implemented, tested, and integrated into the final scene(s), submit a pull request (PR) from your branch. For example, if a team member develops the core mechanics of the game as a minimal prototype, they should open a PR from their branch.



Another team member can then checkout to the branch containing the PR, test the implemented features, and verify their functionality. If everything works as expected, they can request the PR author to merge the changes into the main branch.



Now that the pull request has been merged, each team member should open GitHub Desktop and update their branch from the main branch, following the steps outlined in the next section of the guide.

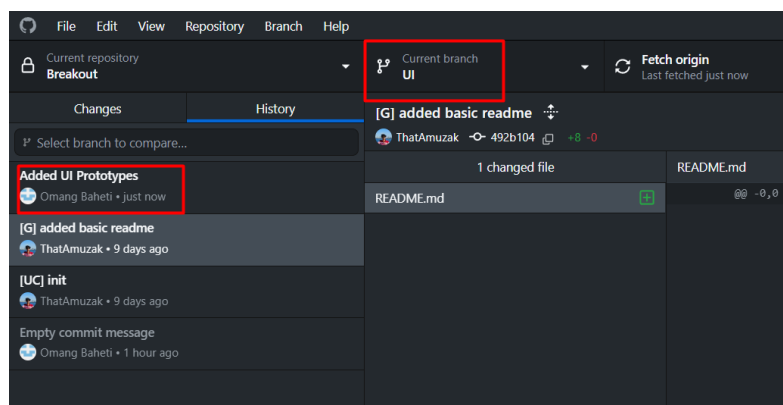
2.6. Updating From Main

After a pull request has been merged into main, all team members should update their branches from main. To ensure that your branch stays up to date with the latest changes from the main branch using GitHub Desktop, follow these steps:

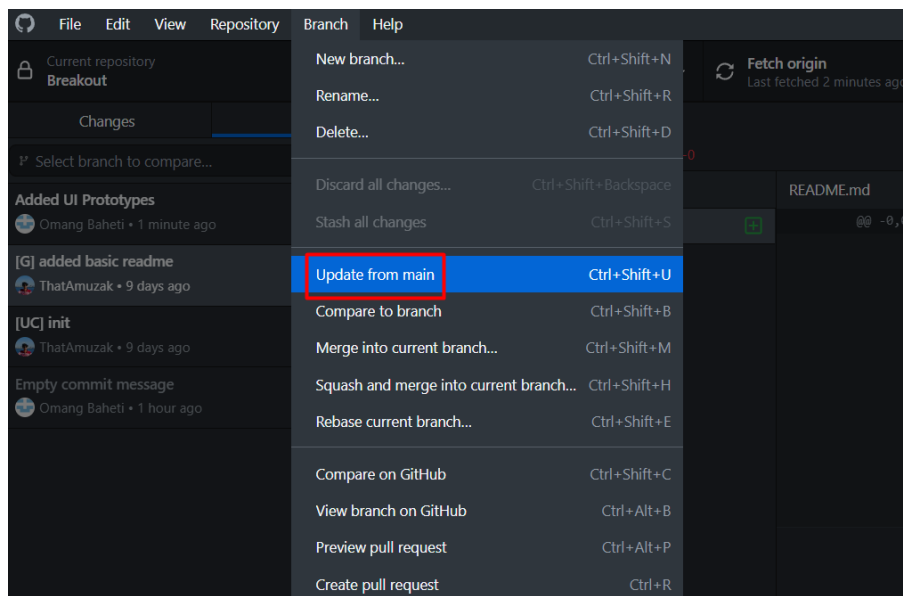
1. Open GitHub Desktop and ensure your repository is selected.
2. Switch to the main branch by selecting it from the branch dropdown.
3. Click "Fetch origin" to check for updates from the remote repository.

4. If updates are available, click "Pull origin" to incorporate the latest changes into your local main branch.
5. Switch back to your feature branch from the branch dropdown.
6. In the top bar go to Repository > Update From Main.
7. If there are any merge conflicts, resolve them in your code editor, commit the changes, and push the updated branch.

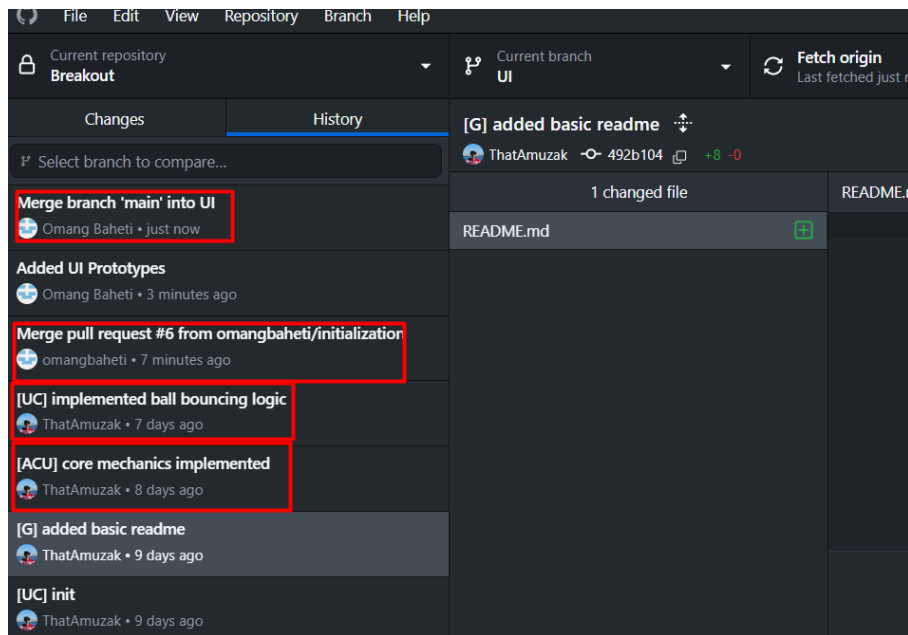
Regularly updating your branch prevents large merge conflicts when integrating features. In this instance, let's say another member is working on implementing the user interface on a different branch called "UI".



Since the Core-Mechanics pull request has been merged into main, the member working on UI should pull from Main so that they are updated with the latest changes

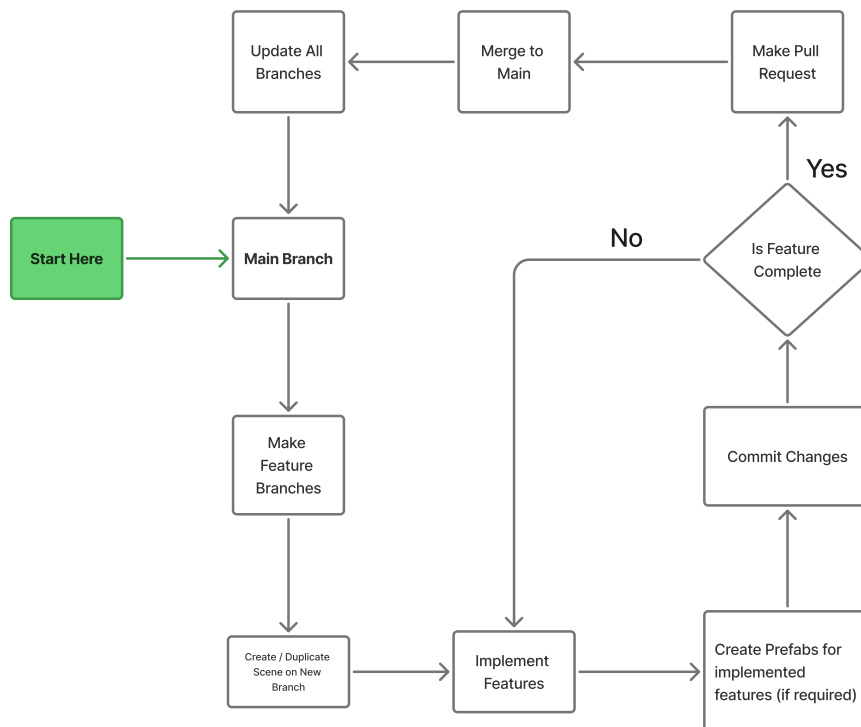


Git will now merge all the commits from main into the "UI" branch and the team member will be updated on all the latest changes from main



2.7. Unity-Git Workflow

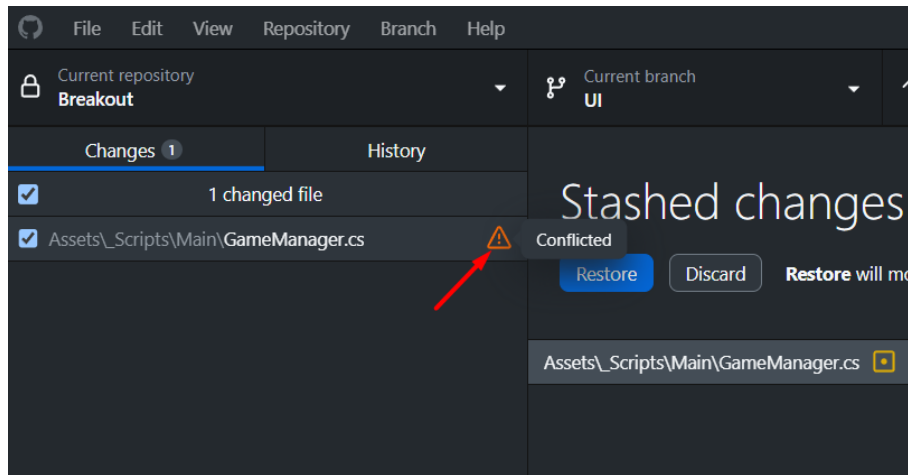
Similar to your other projects, once you have switched to your branch, you can make frequent commits and fixes to your branch. Make sure to commit often, you don't have to commit fully completed features every time. You can split your work into several commits. Once your feature is complete, you can make a pull request, as discussed in Section 2.5



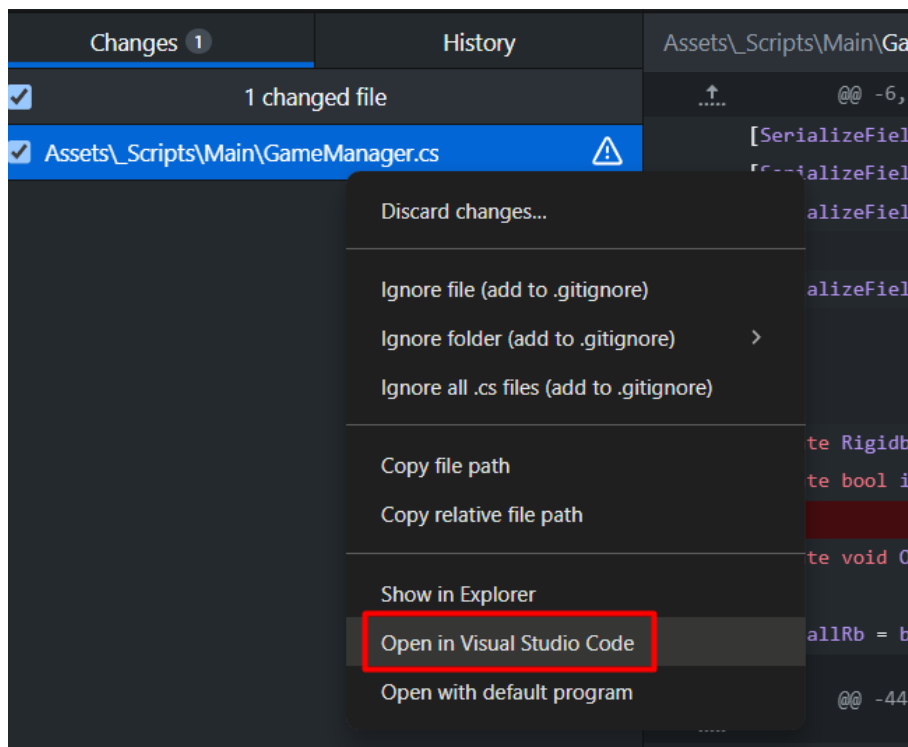
2.8. Resolving Merge Conflicts

If two team-mates are working on the same script or file, there is a chance that you'll run into merge conflicts. You'll most likely run into it when updating from main. These conflicts must be resolved manually before merging changes.

For example, suppose team member A is adding Score-Counter functionality to GameManager.cs, while team member B is simultaneously fixing a bug in PaddleController by resetting it within GameManager.cs. If team member B's pull request is merged before team member A completes their feature, team member A will encounter a merge conflict when updating their branch from the main branch.

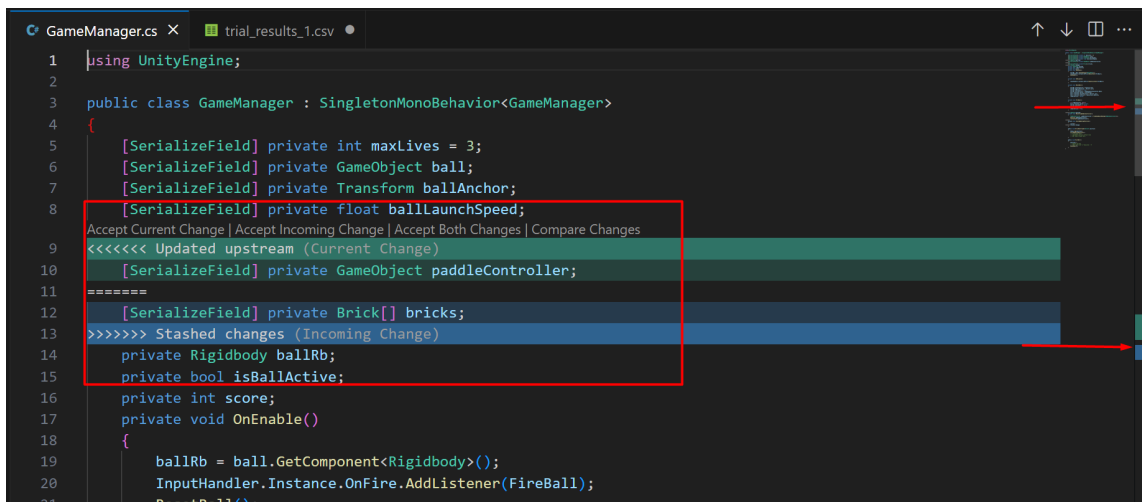


When a merge conflict occurs, Git flags the conflicting file, requiring manual resolution. We recommend using VS Code, as it provides a user-friendly interface for handling conflicts.



To resolve conflicts using VS Code:

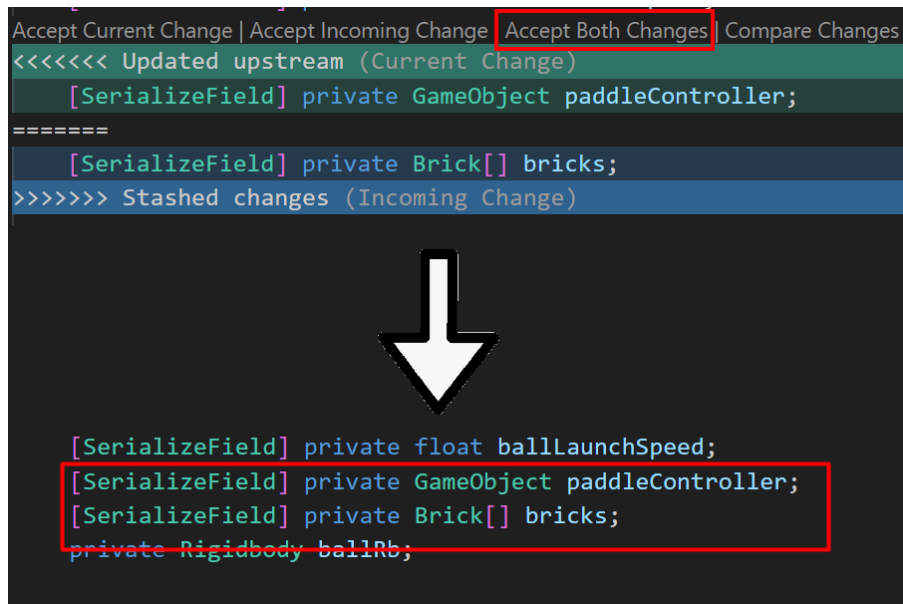
1. Open VS Code and navigate to the file with conflicts.
2. VS Code will highlight conflicting sections with "**Current Change**" and "**Incoming Change**".
3. Choose one of the following options:
 - **Accept Current Change**: Keeps your version.
 - **Accept Incoming Change**: Uses the version from the other branch.
 - **Accept Both Changes**: Combines both versions (requires manual adjustment to ensure correctness).
4. Make sure to save the merged file.
5. Commit the changes to your branch.



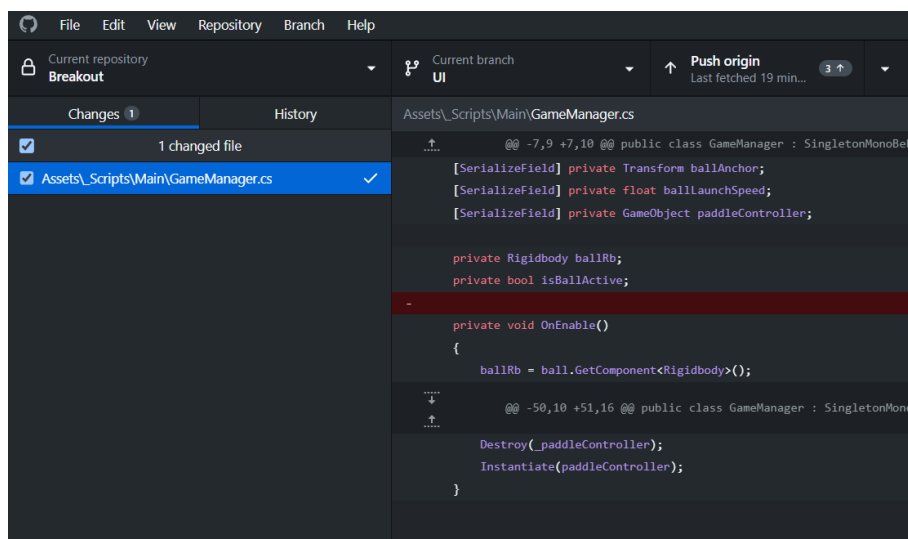
```
1  using UnityEngine;
2
3  public class GameManager : SingletonMonoBehavior<GameManager>
4  {
5      [SerializeField] private int maxLives = 3;
6      [SerializeField] private GameObject ball;
7      [SerializeField] private Transform ballAnchor;
8      [SerializeField] private float ballLaunchSpeed;
9      <<<<<<< Updated upstream (Current Change)
10     [SerializeField] private GameObject paddleController;
11     =====
12     [SerializeField] private Brick[] bricks;
13     >>>>>> Stashed changes (Incoming Change)
14     private Rigidbody ballRb;
15     private bool isBallActive;
16     private int score;
17     private void OnEnable()
18     {
19         ballRb = ball.GetComponent<Rigidbody>();
20         InputHandler.Instance.OnFire.AddListener(FireBall);
21         ResetBall();
```

The VS Code interface visually highlights conflicting sections and provides selection options directly within the editor. Additionally, the scrollbar uses green and blue indicators to mark other conflict locations in the file. To complete the merge, all conflicts must be resolved before proceeding.

In this case we want to preserve both the functionality: The score counter and the solution for the PaddleController bug. Hence for all the merged sections, we will accept both changes.



Similarly, we do so for all relevant parts of the code and save the file. Once the file has no more conflicts we can push our commits to our branch. You can put the commit message for such a situation as “Resolving merge conflict for GameManager”.



For Unity scene and prefab files, it’s best to coordinate with teammates to avoid conflicts altogether as previously covered in Section 2.7. If conflicts occur, check with the author of the conflicting change and manually reconstruct the scene or prefab if necessary.

3. Features

Let’s discuss some features you could consider implementing. As discussed earlier, there needs to be one feature per team member. It needs to be some form of visual addition to the juice, general game feel, or even just conveying the state of the game.

Note

Everyone needs to make a PR and submit at least one feature, including the team member who made the project. Remember, keep the scope of your feature small. The goal of the studio is to get comfortable with the process of working with a team using Unity and Git.

3.1. Overview of the project

The project has the core logic of the game contained in the “_Scripts/Logic”. UI scripts are contained in “_Scripts/UI”. The scene structure of the project is as follows:

1. _Preload.unity scene is the first scene that’s created. It contains our `SceneHandler.cs`, which has the property `DontDestroyOnLoad` using the utility script `DDOL.cs`. This ensures that the entire `GameObject` is preserved through scenes transitions.
2. This also contains a simple `DoTweened` blackout transition screen. It switches scenes using `LoadNextScene` and `LoadMenuScene`.
3. The `GameManager` maintains the state of the game, which for this project is simply tracking the destruction of blocks and when to transition to the next scene.

If you look at the codebase, the `GameManager` has a few comments for places to implement features.

```
public void OnBrickDestroyed(Vector3 position)
{
    // fire audio here
    // implement particle effect here
    // add camera shake here
    currentBrickCount--;
    Debug.Log($"Destroyed Brick at {position}, {currentBrickCount}/{totalBrickCount} remaining");
    if(currentBrickCount == 0) SceneHandler.Instance.LoadNextScene();
}

1 usage ThatAmuzak
public void KillBall()
{
    maxLives--;
    // update lives on HUD here
    // game over UI if maxLives < 0, then exit to main menu after delay
    ball.ResetBall();
}
```

This is where you could consider implementing some simple features for this studio.

3.2. Score

Similar to [Studio 4](#), you can add a simple UI component to show the current score as the number of bricks destroyed, and have a nice score count transition when a brick is destroyed.

3.3. Lives & Game Over Screen

[Studio 4](#) discusses how to implement user interfaces. A basic HUD element showing the current number of lives would be a nice addition. You could also consider implementing a simple feature where once the number of lives are equal to 0, you freeze time, show a UI screen saying game over. Then, using a coroutine, you can transition to the main menu scene after a short delay of 1.5 seconds. You can simply use `SceneManager.Instance.LoadScene()` to tell the scene manager to move to the main menu after `yield return new WaitForSeconds(1.5f)` and setting back the timescale to 1.

3.4. Audio Manager & SFX

Basic audio managers are relatively simple to implement in Unity, and are easy to find out about ([see this for a good example](#)). Additionally, Dr. Jalilvand has also covered implementing audio including SFX and background music extensively in [Week 8 Session 1](#). Having a good audio manager that allows you to quickly inject sound effects throughout your code will be very useful when polishing and upgrading your game.

3.5. Particle Effects

Unity's particle system is a powerful tool to quickly create and prototype nice visuals for your game. The GameManager has a function `OnBrickDestroyed` that receives the position of where the brick was destroyed. You could make a simple particle effect here that exemplifies this action, at the position of the brick.

The particle system component has tons and tons of settings, and it's easier to simply search for specific modifications to your effect rather than look for specific effects. To get started, you can either watch [this great video](#) which shows how to implement three different particle effects, and review Dr. Jalilvand's lecture on the particle system in [Week 7 Session 2](#). Another useful resource is [this video](#) which shows how to implement an explosion effect with Unity's particle system or the VFX graph, which is an extensive powerful tool for all sorts of visual effects in your game, not just limited to particle effects.

3.6. Camera Shake

Camera shake is a nice visual upgrade, and is made trivial with DoTween. You could consider a script similar to the following to implement a simple camera shake effect.

```
1 using DG.Tweening;
2
3 public class CameraShake : SingletonMonoBehavior<CameraShake>
4 {
5     public static void Shake(float duration, float strength)
6     {
7         Instance.OnShake(duration, strength);
8     }
9
10    private void OnShake(float duration, float strength)
```

```

11     {
12         transform.DOShakePosition(duration, strength);
13         transform.DOShakeRotation(duration, strength);
14     }
15 }

```

Attach this to your scene camera, and use `CameraShake.Instance.Shake(duration of your shake, strength of your shake)` to easily implement camera shake effects!

4. Building for WebGL

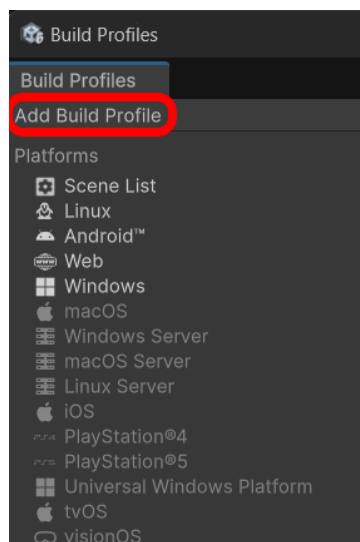
Once you are done with the modifications, and your final project is ready, make sure to play through it completely once in Unity. If everything seems good to go, then you are ready to test out the full build.

Hint

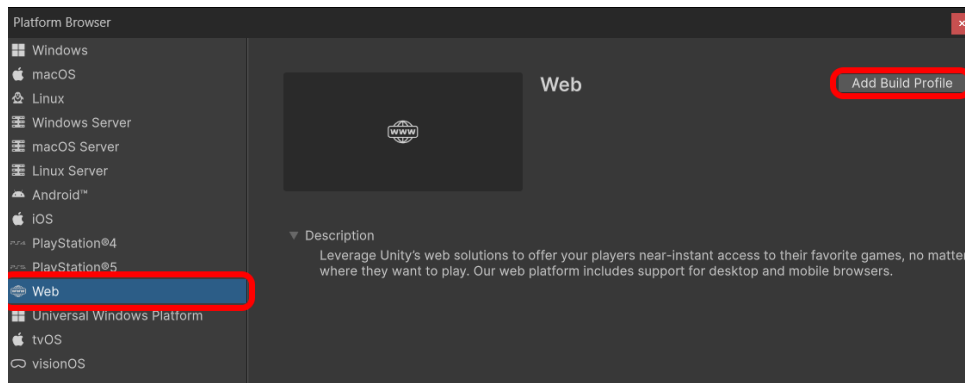
For this assignment, we have already prepared a lot of the elements (such as the build profile configuration), however you will have to set this up for you game. For this assignment, simply verify everything has been configured correctly for your project, so you are aware about what to do for your final project.

4.1. Preparing the Build

To create a build, click from the top bar **File > Build Profiles**. This window shows the different build platforms that you can build for, that you have already installed the modules for. To ensure consistency between your teammates, make a build profile by clicking on “Add Build Profile” on the top right.

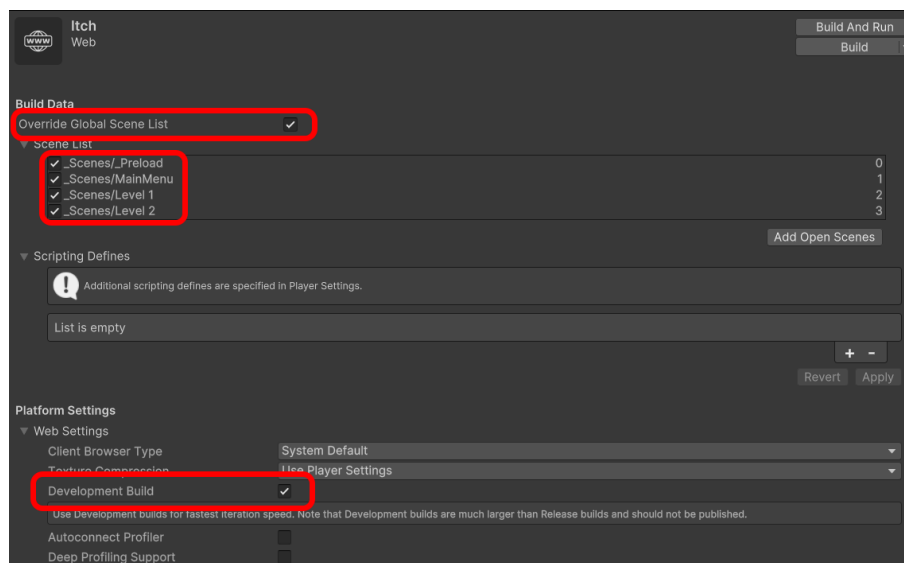


Next, select Web as the build profile and select “Add Build Profile”.



You should now be able to see your new build profile on the left hand side panel. You can right-click to rename it something meaningful (for example, WebGL or Itch). Next, you will want to enable “Override Global Scene List” and put the scenes you have in your game in order of play. In this case, that would be `_Preload` → `MainMenu` → `Level 1` → `Level 2`.

Finally, under platform settings make sure to enable **Development Build**. Release builds can take upwards of 3 hours to make, and are not required for this jam. Remember that first time builds on a machine will also take quite a long time as Unity will be generating files that are critical for the build process for the first time. Subsequent builds should be faster.



4.1.1 About Scene List and SceneManager

The SceneManager API in Unity allows you to load scenes in your build scene list via [SceneManager.LoadScene](#). You can pass the name of the scene as a string you want to load via this function, and it will load that scene, or pass an integer value, which will then load the scene at that index from the Scene List. So for example passing 1 would load MainMenu.

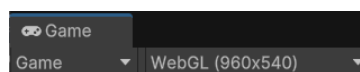
You can check the `SceneHandler` script in this project for a reference on how this is done and potentially automated throughout your project. As discussed earlier, this `SceneHandler` has the `DontDestroyOnLoad` attribute added to it via the DDOL utility

script, so it persists through scene changes and maintains which scene it's in. Feel free to use a similar implementation for your game to handle scene transitions and moving between scenes, or make it more robust by looking at the current scene string to determine the next level!

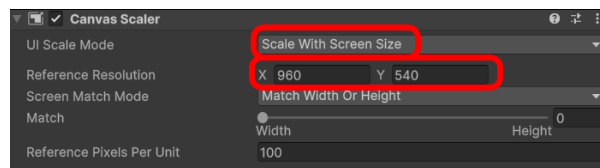
4.1.2 Maintaining Consistent Resolutions for UI

Maintaining a proper resolution from your Unity project to the final build can be tricky. The build process can cause all sorts of small scaling and resolution issues to crop up. To ensure UI is consistent, you want to ensure the UI resolution is **960 x 540** (a standard 21:9 aspect ratio for web games) in the following locations:

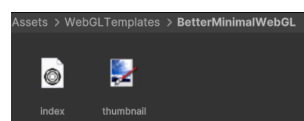
- Your game view window should be **960 x 540**, so what you see is what you get in your build



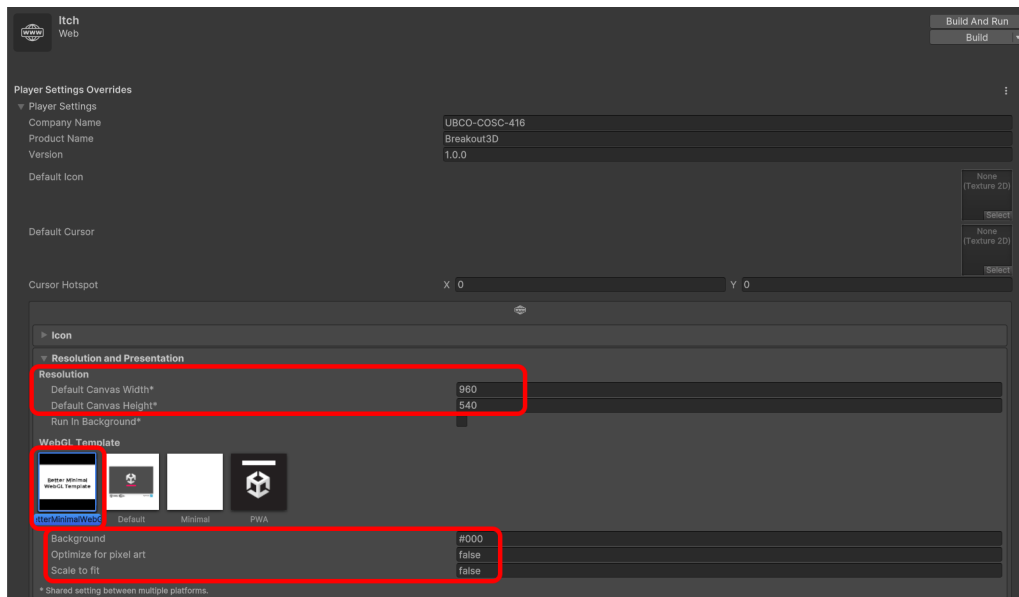
- All your canvases, will have a “Canvas Scaler” component. Make sure this is always set to “Scale with Screen Size”, and the reference resolution is **960 x 540**.



- In your build profiles, you will need a WebGL template that's properly configured for itch. Fortunately, you can download this [Better Minimal WebGL Template](#) which ensures resolutions and aspect ratios are maintained. Download the file for 2020.2 and higher, unzip the folder and place the contents in your asset folder.



Now, in your Build Profiles, you should be able to see and configure this new WebGL template with the following settings, under the Resolution and Presentation settings.



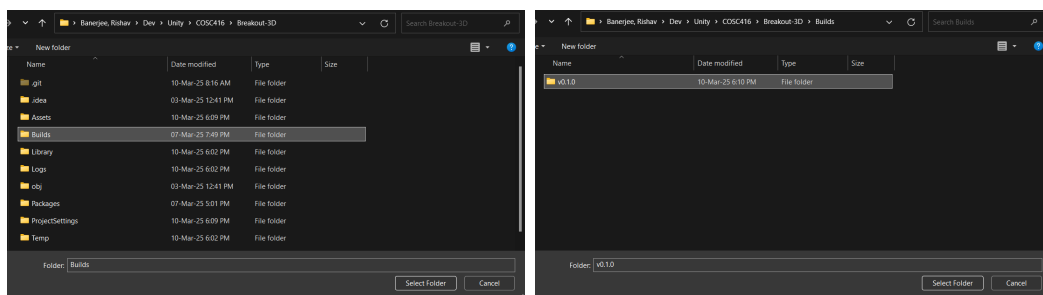
And that should be it! It's a lot to keep track of, but making sure these are followed early on will help avoid issues with UI incorrectly scaling or being out of place in the final build from the expected version within Unity.

4.2. Exporting to Itch

Once your game is complete, you can click on the Build option on the top right of your Build Profiles window.



Make a folder called "Builds" in your Unity project. This will contain all your Builds. Within this, make folders based on your current version, or any other versioning system that your team can agree upon. For example [semantic versioning](#) is easy to follow and popular.

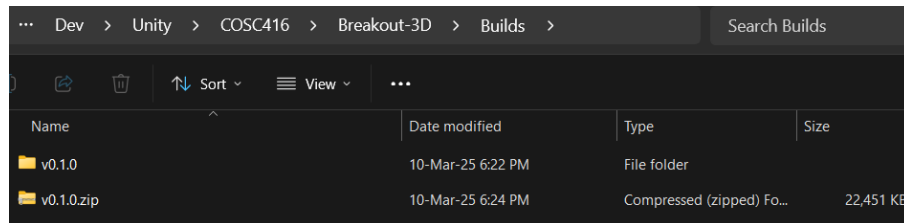


Note

"Builds" is a folder name that is typically gitignored from your project, so that unnecessary files are not uploaded to your git project. If you choose to have another name, then make sure that it's added to your gitignore.

Once you select folder, Unity will start building the project. As discussed, first time builds can take a significant amount of time. Hence, we recommend build early and run tests early. Subsequent builds will always be shorter since a lot of critical files will be generated by Unity after the first build to speed up the process.

Once your build is complete, make a zip file of the build. This is what you will upload to itch.

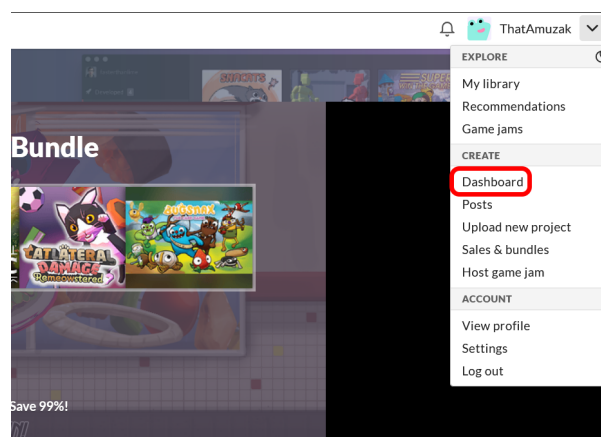


Name	Date modified	Type	Size
v0.1.0	10-Mar-25 6:22 PM	File folder	
v0.1.0.zip	10-Mar-25 6:24 PM	Compressed (zipped) Fo...	22,451 KB

Note

Only one person in the team needs to do this initially. Once the itch page has been set up up, they can the others as project admins via their URLs, which will be discussed at the end of this section.

Now, go to itch.io. Make an account if you are visiting for the first time. Once you have created your account and logged in, open your Dashboard.



Look for the “Create new project” button here.

Create new project

On the new project tab, you can configure your project for a free playable HTML game.

Title
Breakout3D

Project URL
https://thatamuzak.itch.io/breakout3d

Short description or tagline
Shown when we link to your project. Avoid duplicating your project's title.
A simple prototype of Breakout with Unity and WebGL

Classification
What are you uploading?
Games — A piece of software you can play

Kind of project
HTML — You have a ZIP or HTML file that will be played in the browser
You can add additional downloadable files for any of the types above

Release status
Released — Project is complete, but might receive some updates

Pricing
☐ \$0 or donate
☐ Paid
☒ No payments

The project's files will be freely available and no donations can be made

Gameplay video or trailer
Provide a link to YouTube or Vimeo.
eg. https://www.youtube.com/watch?v=5JEaA47sP

Screenshots
Optional but highly recommended. Upload 3 to 5 for best results.
Add screenshots

Next, upload the zip file with the Upload Files button, and enable the check-box that says “This file will be played in the browser”. Finally, set the embed option as “Embed in Page” and “Auto-detect size (Unity HTML only)”. This is using the WebGL template that we set up earlier.

Uploads

Upload a ZIP file containing your game. There must be an `index.html` file in the ZIP. Or upload a `.html` file that contains your entire game. [Learn more](#) →

Any additional files you upload will be made available for download. You can apply a minimum price to the project after uploading additional downloadable files.

v0.1.0.zip · Success [More...](#) [Delete file](#)
22mb · [Change display name](#)
Today at 6:36 PM

☒ This file will be played in the browser

Upload files or [Choose from Dropbox](#) [Add External file](#) ?

File size limit: 1 GB. [Contact us](#) if you need more space

TIP Use **butler** to upload files: it only uploads what's changed, generates patches for the [itch.io app](#), and you can automate it. [Get started!](#)

Embed options

How should your project be run in your page?

Embed in page ▾ Auto-detect size (Unity HTML only) ▾

Finally, at the bottom of this page, set the visibility of your project to public, and set it to unlisted in search and browse if you want to. Hit save and you should be done!

Visibility & access

Use Draft to review your page before making it public. [Learn more about access modes](#)

☐ Draft — Only those who can edit the project can view the page
☐ Restricted — Only owners & authorized people can view the page
☒ Public — Anyone can view the page

Public access settings [×](#)

☐ Disable new downloads & purchases ?
☒ Unlisted in search & browse

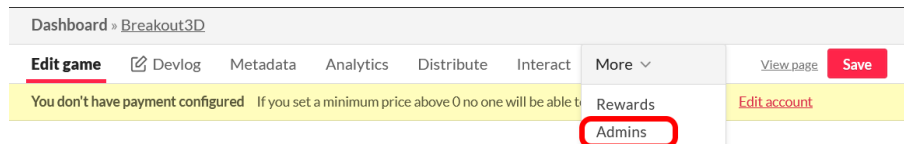
Save [View page](#)

Hint

The public option might be disabled initially. Save and view the page once to enable this option.

If you have done everything correctly, your game should now be playable! Now for future updates, you can just delete the existing uploaded zip, and upload the new zip, set that to be playable in the browser, and save the page.

You can also add your team members as admins to the page, so anyone can configure and update this page as required.



5. Conclusion

That's it for Studio 5, and the guides. For your submission, submit a document with the links to your team's GitHub repo, and the itch.io build. At this point, you should be comfortable implementing different game features with Unity, and learning new aspects of the engine when required. As a final challenge, your final project submission will be a game for the [Retro Twist Game Jam](#). You will put all that you have learned in this course and more to the test, and race against time to make a fully playable experience in just two weeks! Game jams are amazing experiences, and we hope that this course will have prepared you well to conceptualize, prototype, implement and polish your very own video games.