Last Updated: 3/3/2025

Developer Documentation

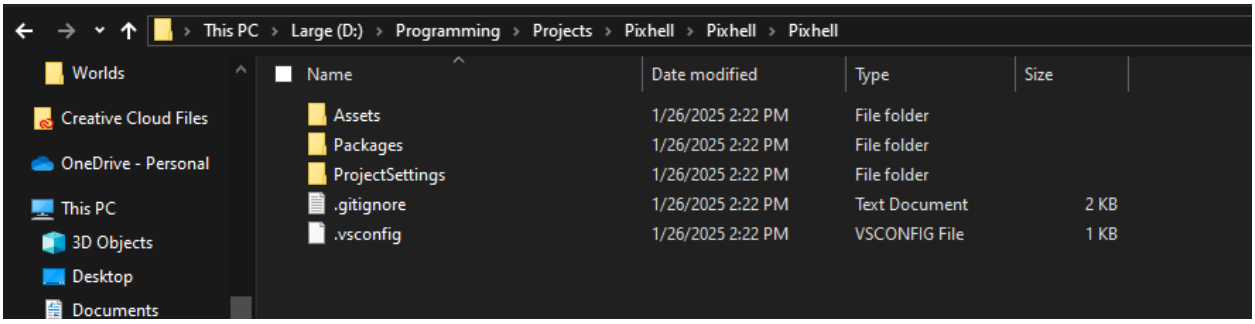
Source Code

The source code is located in the Pixhell folder located in the repository found here. Inside this folder, the folder with most of the code is the Assets folder, which contains various folders related to sprites, animations, scenes, and scripts. This folder also contains the StreamingAssets folder, which is home to information such as play run information and item data. The other folders located in the Pixhell folder contain information related to Unity settings and are generally automatically generated by Unity.

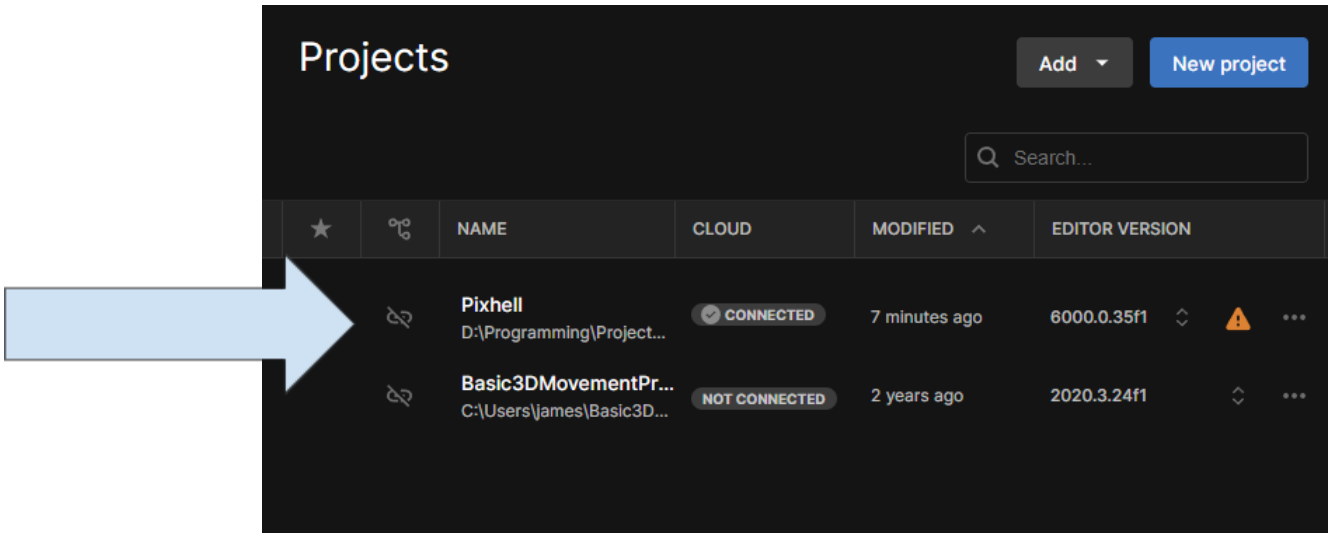
The documentation, weekly reports, and other general information are all located outside of the Pixhell folder and are in the source folder of the repository found here. The reports are found in the "/reports" folder, and the documentation is found in the "/documentation" folder.

In order to open the source code and test the project:

1. Download Github Desktop - Download GitHub Desktop | GitHub Desktop
2. Clone the repository to an easily findable folder - Use this link: https://github.com/CS362-Team12/Pixhell.git
3. Download Unity Hub for desktop - Start Your Creative Projects and Download the Unity Hub | Unity
4. In Projects - click the add button -> Add project from disk
5. Navigate to the Repository folder
6. Select the Unity folder, which is the folder that has Assets, Packages, and Project Settings



7. Download any needed updates (correct Unity version, etc)
8. Open the Pixhell project



Layout

Pixhell/

This is the root of our repository, and contains folders related to documentation, the source code, presentations, reports, and our living document

Pixhell/documentation

Contains developer and user documentation

Pixhell/presentations and /reports

Contains our project proposal, midterm, and weekly reports respectively

Pixhell/Pixhell

This folder is the root of our source code. All of the project's assets, and unity settings are contained within this folder.

Pixhell/Pixhell/Assets

This folder contains the majority of our work on this project, including sprites, animations, and scripts.

.../Assets/Animations, /Hero and Opponents, /Evil Wizard 3

These folders contain the majority of the animation and animation controllers for the game.

…/Assets/PNGs

This folder contains the majority of the UI elements, including the health and XP bar sprites and the dash ability icon sprite.

…/Assets/Prefabs and /Resources

These folders contains all of the pre-configured objects that are used throughout the game, such as the player characters, enemies, triggers, damage text, among other things. Any prefabs that are accessed through script are located in the /Resources folder.

…/Assets/Scene

This folder contains all of the scenes for levels and game environments, including our global scene, menus, lobby (Limbo) and arenas that are used by Unity.

…/Assets/Scripts

This folder contains all of our scripting and behavior control, which includes character control, inventory, enemy spawning, sound control, and many others.

…/Assets/StreamingAssets

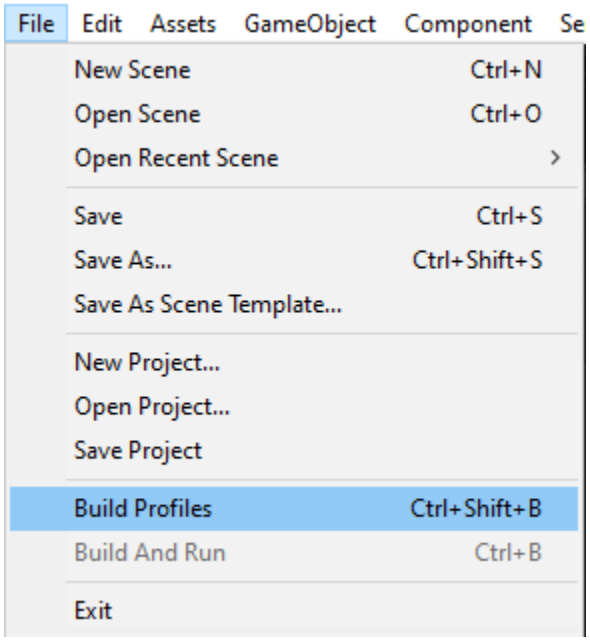
This folder stores essential game data during runtime, including player data (such as items and money), and arena data (such as enemy wave patterns).

Building

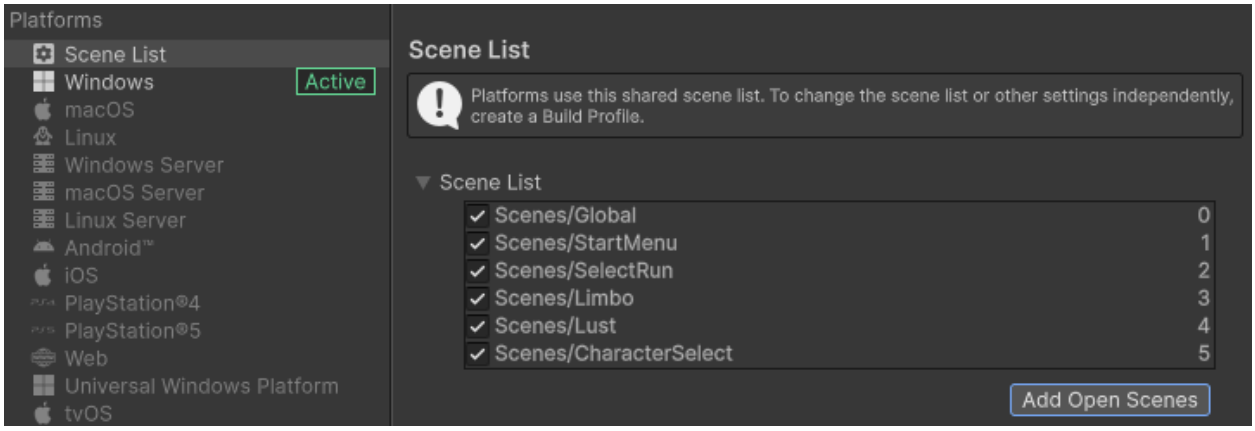
The build process is mostly automated by Unity, with the custom build scripts located within the project when obtaining the source code. Here are the process steps for building a new version:

Building:

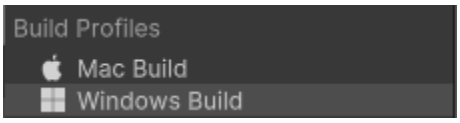
1. Open Pixhell in Unity
2. Ensure that in `scripts/GameConstants.cs` DEBUG is set to `false`. Leaving DEBUG on can cause some debug effects, such as increased drop rates and forced upgrades, to be on. Changing this variable to `false` will force these effects to be off.
3. For official builds (non beta), any save files should also be deleted from the local machine. To do this, navigate to Assets/StreamingAssets/Runs and delete all files there. Run the project in Unity to ensure that no runs are saved.
4. Go to File > Build Profiles



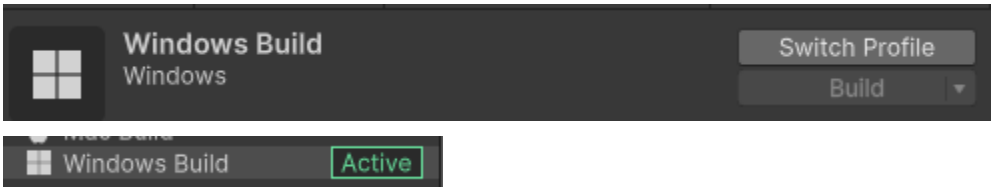
5. Check that the Scene List contains all the scenes included in the game. If necessary, click “Add Open Scenes” to include all the open scenes. See the image below for scenes from the last update.



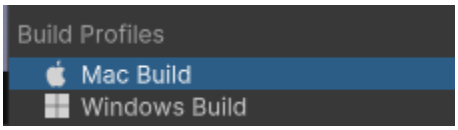
6. Windows:
    a. Find “Windows Build”



    b. Click “Switch Profile” to make the Windows Build active

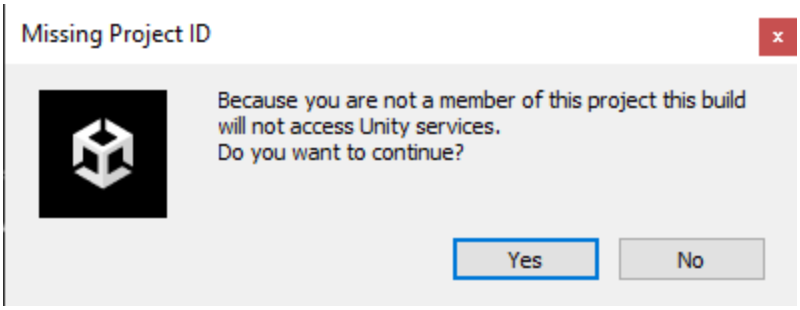


7. Mac:
    a. Find “Mac Build”



    b. Click “Switch Profile” to make the Mac Build active

8. Click Build on your chosen operating system, and select a folder that is easy to find.
    a. If you get the error pictured below, click “Yes” to continue. The project does not rely on Unity services and the build should continue without it.



9. Navigate to the folder the game got built (in some cases, the folder will automatically open). Run pixhell.exe to ensure it works.

Releasing on Github:

10. Navigate to https://github.com/CS362-Team12/Pixhell/releases
11. Utilize Github best practices for releases: https://docs.github.com/en/repositories/releasing-projects-on-github/managing-releases-in-a-repository
    a. Choose a proper tag based on version history
12. Attach a zip file of the mac build and windows build from the previous steps. Name them Mac-<TAG>.zip and Windows-<TAG>.zip, where you replace <TAG> with the name of the tag used.
13. Select “Set as a pre-release” for beta and draft versions.
14. Click “Publish release”.

Tests

Black box testing currently is done through debugging statements and console logs while playing. For instance, when getting hit as a player, the health will be displayed in the console each time the player is hit, and so comparing the values to the expected output is the test for that. Automatic white box testing is currently being implemented and will take the form of a single script with a list of tests that test old functions. Running this script will be useful between large updates to ensure no prior functionality is broken, or if it is, then identify where new issues are occurring quickly.

Adding new tests will be simple; either adding debugging statements throughout the code to test functionality live or adding more functions to the automatic testing script.

Items

Assets/Scripts/Items/ and Assets/StreamingAssets/Items/GlobalItems.csv

Adding Items:

Adding items can be done through the GlobalItems.csv file, by simply adding a new row to the file. All numbers are percentage based (so 100 = 100% increase). A sprite file location will need to be added (all of which are relative to the StreamingAssets folder) for display in the shop.

Once items are added, they should automatically show up in the shop in the lobby (Limbo) to be purchased.

Enemies

Assets/Scripts/Combat/Enemies/

Base Enemy Class:

States:

Enemies have a state table which determines **what** actions they perform. Eventually, these states will probably be additionally overridden by certain conditions to fulfill our smarter enemy AI requirements.
Use the following game constants to create an array of the states you want the enemy to be in.
GameConstants: MOVING, ATTACKING, IDLING

You can add more GameConstants and more states, just update the Enemy.cs file to include 1. A condition checking for that state in the Update() function and 2. A function call to a public, virtual function. This should not affect any other enemies. Then, add the constant to GameConstants.cs in the scripts folder using the next available integer.

For example:
states = [MOVING, IDLING, ATTACKING, IDLING] will have an enemy move around, then idle, then attack, then idle again, before looping back to the start of the array.

Timers

Enemies have a timer table which determines **how long** they perform their actions. This should be a float for how many seconds you wish for that state to run.
All timers are scaled with a multiplier of 0.8 to 1.2. This ensures that enemies spawned around the same time don't perform on the exact same schedule, and adds variability to timings. If needed, a way to surpass this randomness could be added.

Restrictions:
The timer array should be the same length as the states array
Any state which is ATTACKING should be of length 0: Any timer of length zero will run for exactly one frame. If you wish for the enemy to attack multiple times, or for a length of time, either add a new state, or have the enemy be in the ATTACKING state multiple times in a row. i.e. [MOVING, ATTACKING, ATTACKING, IDLE].

For example, using
states = [MOVING, IDLING, ATTACKING, IDLING], and
timers = [3f, 0.75f, 0f, 0.75f],
Will make the enemy move for 3 seconds, idle for ¾ of a second, attack, and then idle for ¾ of a second.