# Introduction

Today, I’m going to be looking at adding test automation to a Unity project and how to setup the tests as part of a continuous integration (CI) pipeline.

I’m going to be doing this using [AltUnity](https://altom.gitlab.io/altunity/altunitytester/pages/overview.html) for UI test automation. The goal of this exercise is to write some simple UI tests and have them run on a CI server to form part of a CI build for a Unity project. Once this is working, we can write more complex tests until our heart’s content.

I will be constructing the CI build using Azure DevOps pipelines, but this can be done in any CI tool of your choice. In this article, I’ll be using a voxel game I created, but you can substitute this for a game of your choosing.

# AltUnity Overview

I thought I’d start by looking at how AltUnity actually works. You can read this section purely for information – we will be going through setting up a project step-by-step later in the article.

AltUnity is “an open-source UI driven test automation tool that helps you find objects in your Unity game and interact with them using tests written in C#, Python or Java”.

I will spare going into too much detail, as the [AltUnity site](https://altom.gitlab.io/altunity/altunitytester/pages/overview.html#how-it-works) does a great job of explaining how it works, but at a high level, you instrument your code using the AltUnity Server module. Instrumenting your game code with this module starts a TCP connection on the device running the game and provides an instance of the AltUnity Driver with access to your GameObjects. Your tests can then access the game objects via the AltUnity Driver module.

You can install AltUnity into your Unity project via the asset store. Once imported, you will gain an ‘AltUnity Tools’ option in your toolbar, from which you can launch the ‘Tester Editor’

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text

Description automatically generated

As per the [AltUnity tutorial](https://www.youtube.com/watch?v=L4yAgv8Jc8s), we can create tests that will show in the editor by creating our test scripts under a folder at AltUnityTester>Editor>Tests (creating the test folder ourselves). We can create a basic AltUnity test file via the create menu. This will create an Nunit test file.

Graphical user interface, application

Description automatically generated

Tests created here will then show up in the AltUnity Tester Editor window in Unity, and we can execute them using the AltUnity Tester Editor GUI. The tests work by referencing an AltUnityDriver object which interacts with the AltUnity Server that is added to our game. This is one way of writing out tests with AltUnity.

A picture containing text, monitor, screenshot, computer

Description automatically generated

If we want to run our game with an instance of AltUnity Server, so we can run our tests, we can do so by clicking on ‘Play in Editor’ within the AltUnity Editor. When it runs, you’ll see a popup over your game indicating that the AltUnity Server is running. We can then run our tests. When we do, we should see the game change state as per the code in our test.

A screenshot of a computer

Description automatically generated with medium confidence

# The CI Process

Ok, so we’ve had a look at how AltUnity works, and how we can create a simple test file via the GUI. But how do we get this all to run as part of a CI build? Let’s first draw out how we want this to work.

Diagram

Description automatically generated

The above illustrates an example CI process for our game. Once a developer checks in some changes, we checkout the code, build the game and execute the tests. If this all passes, we will pass the CI build and merge the changes.

To construct this process, we will need to take what we’ve already looked at and do it all via the command line. This will enable us to execute it remotely from a CI build pipeline.

First, we will need a script to build the game and instrument it with AltUnity Server that can be ran from the command line. Second, we will need a test project that will use AltUnity Driver to communicate with the AltUnity Server and execute our tests via a running instance of the game. Once we have these components working, we can combine them and execute them remotely as part of a CI build.

# Creating the Build Script

Let’s start off simple by getting a custom build script working on our machine. Thankfully the [AltUnity docs](https://altom.gitlab.io/altunity/altunitytester/pages/advanced-usage.html#build-games-from-the-command-line) has a guide on building games and instrumenting them with AltUnity from the command line. We will follow the guidance here with some minor changes to setup a build script for our game that builds a development version of our game for the 64 bit windows platform, that is instrumented with AltUnity Server.

Within the projects Assets folder, create a folder named ‘Editor’ if it does not already exist. Editor is a [special folder name in Unity](https://docs.unity3d.com/Manual/SpecialFolders.html) that makes the functions visible in the editor – in our case, via the toolbar menu for testing if we need it (though we will mostly be executing this from the command line).

Within the Editor folder, create a C# script. Call this whatever you like. This will be your build script. We will call ours ‘CustomBuildScript’. I recommend following the guidance in the [AltUnity guide](https://altom.gitlab.io/altunity/altunitytester/pages/advanced-usage.html) to create the build function, but you can use the below as a basis and simply change the scenes array and locationPathName values. You can also change the platform as you desire.

using UnityEngine;

using UnityEditor;

using System;

using Altom.Editor;

public class CustomBuild

{

     [MenuItem("Game Build/Build and Run")]

    static void BuildAndRun()

    {

        try

        {

            BuildPlayerOptions buildPlayerOptions = new BuildPlayerOptions();

            buildPlayerOptions.scenes = new string[] {

                "Assets/Scenes/MainMenu.unity",

                "Assets/Scenes/MainLevel.unity"

            };

            buildPlayerOptions.locationPathName = "./builds/VoxelGame.exe";

            buildPlayerOptions.target = BuildTarget.StandaloneWindows64;

            buildPlayerOptions.options = BuildOptions.Development | BuildOptions.AutoRunPlayer;

            PlayerSettings.SetApiCompatibilityLevel(BuildTargetGroup.Standalone, ApiCompatibilityLevel.NET\_4\_6);

            //  Setup for AltUnity

            var buildTargetGroup = BuildTargetGroup.Standalone;

            AltUnityBuilder.AddAltUnityTesterInScritpingDefineSymbolsGroup(buildTargetGroup);

            if (buildTargetGroup == UnityEditor.BuildTargetGroup.Standalone)

                AltUnityBuilder.CreateJsonFileForInputMappingOfAxis();

            AltUnityBuilder.InsertAltUnityInScene(buildPlayerOptions.scenes[0]);

            var results = BuildPipeline.BuildPlayer(buildPlayerOptions);

            AltUnityBuilder.RemoveAltUnityTesterFromScriptingDefineSymbols(BuildTargetGroup.Standalone);

        }

        catch (Exception exception)

        {

            Debug.LogException(exception);

        }

    }

}

Here, we basically take what is explained in the [AltUnity docs](https://altom.gitlab.io/altunity/altunitytester/pages/advanced-usage.html#build-games-from-the-command-line) and change it to be a standalone windows build. We add our scenes and configure the build via BuildPlayerOptions. We then add the AltUnityBuilder code to instrument the build with AltUnity server and then build the player.

Note: It is important that we set the API compatibility to .NET\_4\_6 when running on standalone windows. Without this line, we will encounter compatibility errors when running in standalone mode.

Also, note how we add “BuildOptions.AutoRunPlayer” to our buildPlayerOptions. This causes the game to launch on build. We can create a separate method without this if we want to simply build the game player file and not execute it.

To run this from the command line and build your game, run the following (replace placeholders accordingly):

"<path to Unity.exe>" -quit -batchmode -projectPath "<Your project dir>" -executeMethod CustomBuild.BuildAndRun

Note: this command will error if you also have your game open in Unity.

If the above has ran successfully, your game should launch, and you should see the AltUnity Server popup running over the top (if you’re not running on a windows admin account, you may get a firewall prompt also).

Graphical user interface, diagram, application

Description automatically generated

This means our AltUnity server is running within our game code, and we can now call it from our tests. Now all we need to do is write some tests!

Note: AltUnity Server will only be enabled in development mode. AltUnity does not recommend enabling AltUnity in your production builds. Though you can toggle this if you desire.

# Creating the Test Project

Let’s create a simple test project and get it running against our instrumented game.

Start by opening Visual Studio and creating a test project. For this example, we will be using XUnit for .NET core, though feel free to select the test framework of your choice.

Once created, we need to add the AltUnityDriver library to our test project. Recall that the AltUnity Driver is used to communicate with the AltUnity Server.

Right click on your test project and select ‘Manage Nuget Packages’. Search for ‘AltUnityDriver’ and install the package (by Altom).

Once installed, rename the default test class (or create one if you don’t have one) and add a basic test similar to the below that will work for your game. The test below will load the MainMenu scene, find a GameObject by the name of “StartText”, click it and wait for the scene to be set to Main.

 public class MainMenuTests : IDisposable

    {

        AltUnityDriver altUnityDriver;

        public void Dispose()

        {

            altUnityDriver.Stop();

        }

        public MainMenuTests()

        {

            altUnityDriver = new AltUnityDriver();

            altUnityDriver.LoadScene("MainMenu");

        }

        [Fact]

        public void StartGame()

        {

            altUnityDriver.FindObject(By.NAME, "StartText").Click();

            altUnityDriver.WaitForCurrentSceneToBe("MainLevel");

        }

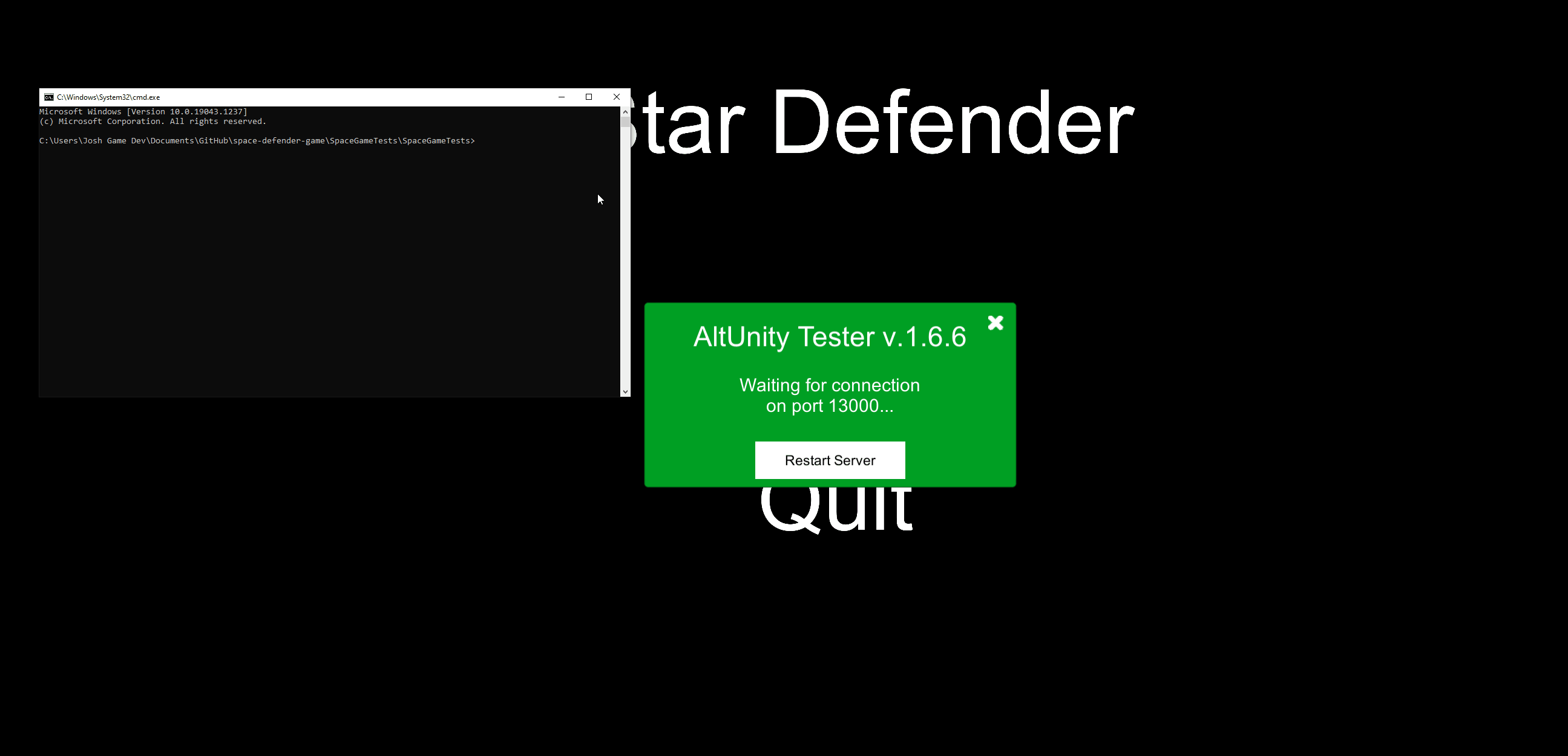
    }

This test will create an instance of the AltUnityDriver (using the default configuration) and execute a simple test that finds a GameObject by the name of ‘StartText’ and clicks it, then verifies the ‘Main’ scene has successfully opened. This is based on the sample game provided. If you’re using your own game, you can write your own tests within here.

We can now execute these tests with the instrumented game running. The driver will connect to the AltUnity server and we should see our test execute. You can do this via visual studio, but to prepare for the CI build, we will do this using dotnet test via the command line.

Start a fresh instance of your game, either by running the build script created previously, or by simply running the .exe file. When you have the AltUnity Tester popup indicating that AltUnity is waiting for a connection, open a command prompt in your test project directory and run your tests by executing:

‘dotnet test’



You should see the game instance be controlled by your test code, and the results of your tests output to the test window.

# Bringing it Together

The last thing we will do before creating a CI pipeline, is combine our build and tests into a simple shell script. You are welcome to skip this step as we do not need to do it, but it will help understand how the CI build will work and make running things locally far more efficient. It will also give us a reference to work with when we’re building our CI pipeline.

If you don’t have git installed on your machine, before completing this step, install it from the [git website.](https://git-scm.com/downloads)

Git comes with Git Bash that enables you to run bash command on your windows machine.

From a directory of your choosing, create a file called ‘test.sh’ and add the following lines, replacing the placeholders accordingly:

echo "== Building and starting game =="

"path to unity.exe " -quit -batchmode -projectPath "your project path " -executeMethod CustomBuild.BuildAndRun

echo "== Running tests =="

cd "testing directory"

dotnet test -- xunit.parallelizeAssembly=false

echo "== Tests completed"

echo "==> Killing game instance"

taskkill //PID $(tasklist | grep VoxelGame.exe | awk '{print $2}') //T //F

This runs the two commands we’ve manually ran previously and then kills the game instance once complete. Now all we have to do is right click in the directory containing the script, select ‘Git Bash here’, once the terminal opens, type ‘sh test.sh’ and you should see the game build and run along with the tests being executed and finally, the game being automatically closed.

Now we’ve technically automated our game build and testing locally. Let’s get it working in a CI build.

# Creating the CI Pipeline

For this, we’re going to be using Azure DevOps. Navigate to your DevOps organisation and create a project if you don’t have one already. Under the project, navigate to project settings > agent pools.

Graphical user interface, text, application, email

Description automatically generated

Now you’re going to need a machine that you can use as a build agent that has Unity installed. For me, I’m using my local desktop for the sake of this exercise. In a real scenario, you’d want to use a dedicated build agent.

Once you have identified which machine to use as a build agent, select the ‘Default’ agent pool in your Azure DevOps project settings and select ‘New Agent’. From the pop-up window, follow the instructions to download and configure the build agent to your machine. This will run you through downloading the agent and running a PowerShell script to configure/run it.

Once complete, in your Agent Pool > Agents, you should see your build agent listed.

Graphical user interface, text, application, email

Description automatically generated

Now, make sure all of your local changes to your game project, including the custom build script, are published to a repository, in my example, I’m using GitHub. Once that’s done, create a new pipeline by navigating to ‘Pipelines’ in the left project menu. Choose your repository type and work through the wizard accordingly. If you’re using the example GitHub project, you will get prompted to authenticate and setup permissions. Just follow the pages as prompted to set this up.

The next step will ask you to configure your pipeline. We’ll start with the starter template which gives us a default template that uses a Microsoft hosted agent. We want to change this to be our own build agent we setup previously. Then we’ll just need to run the build script via cmd and execute our tests via the DotNetCoreCLI. To do this, configure something similar to the following YAML – replacing the paths accordingly for your project and be sure to correct any indentation errors:

# Add steps that build, run tests, deploy, and more:

# https://aka.ms/yaml

trigger:

- main

pool:

  name: Default

steps:

- task: CmdLine@2

  inputs:

    script: |

      "<path to your unity.exe file on build machine" -quit -batchmode -projectPath ".\<game folder name>" -executeMethod CustomBuild.BuildAndRun

  displayName: "Building Game"

- task: DotNetCoreCLI@2

  inputs:

    command: test

    projects: '<path to test project>'

    arguments: '--configuration $(buildConfiguration)'

  displayName: "Execute Tests"

Once you’ve created your pipeline, make sure your build agent is running. During setup, you’ll have configured it to run either as a service or manually. If configured to run manually, you can run it by executing the run.cmd command in an admin PowerShell window, within the build agent folder you downloaded earlier. If it’s running, you should see something like the following:

Text

Description automatically generated

Within Azure DevOps > Project Settings > Agent Pools, if you open your agent pool, you should be able to see your agent as ‘Online’ under the ‘Agents’ tab.

Graphical user interface, text, application

Description automatically generated

Now you’re setup. Run your pipeline. If all executes correctly, you should see the job be picked up by your agent in your PowerShell window. After a short delay, it will execute the steps and you should see your game launch in the background on your build machine.

Text

Description automatically generated

Note: make sure you run the agent using an account with admin access on the machine, otherwise you may get prompted to accept windows firewall.

Now you’re set. You can add more to this, such as unit tests and more complex UI tests, but hopefully this gives you a base working CI build to work from.

# Conclusion

In this article, we’ve looked at how to implement UI tests into Unity using AltUnity. We’ve explored how this works and how to use it. We then took it one step further and looked at setting up a continuous integration pipeline that built our game and executed our tests automatically.

# Final Notes

The one drawback of this is that it still requires a Unity License for your build machine, so if you’re a solo developer, you might have to think about licensing, or perhaps review when you want to run your automated builds. Other licensing options include [Unity Build Server](https://unity.com/products/unity-build-server) which may be worth exploring for scaled projects.