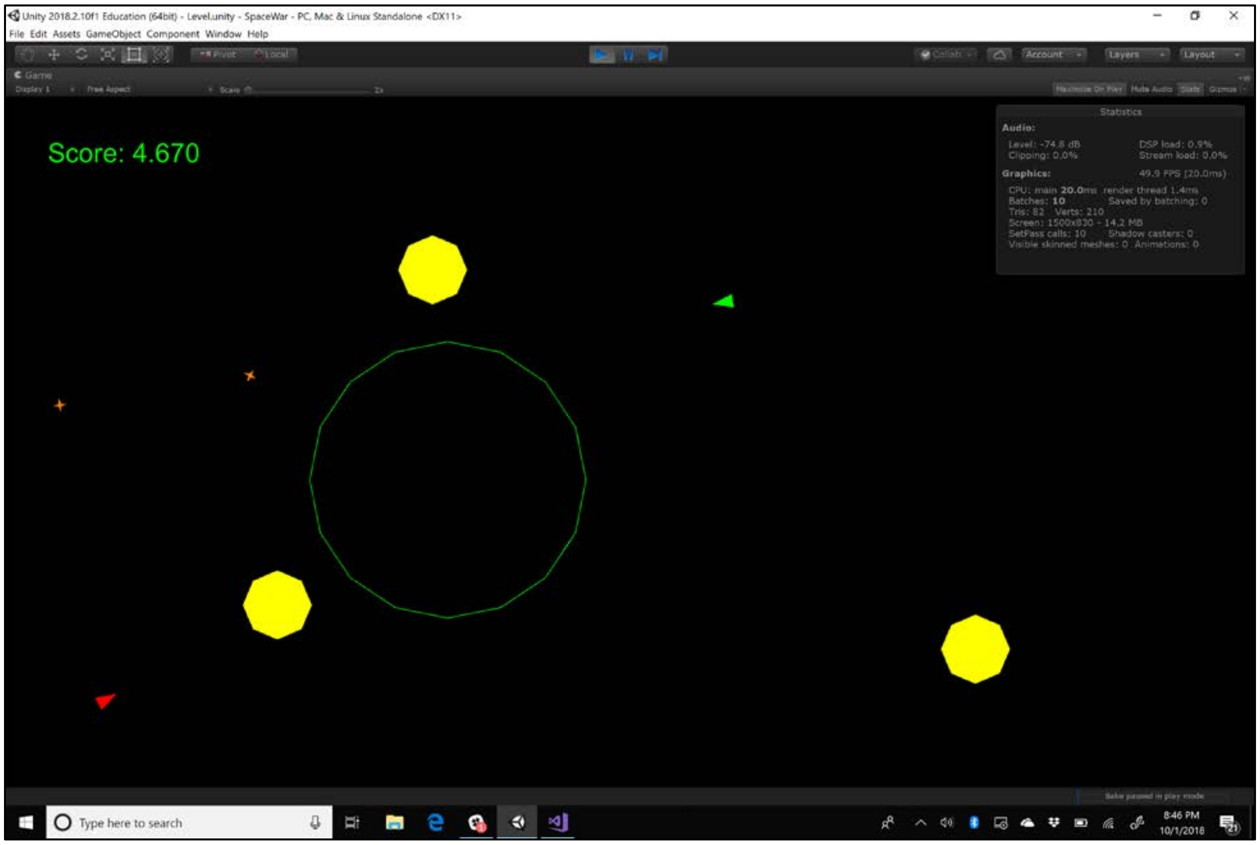
Exercise 1:
Working with the Unity Architecture



```
Unity 2018.2.10f1 Education (64bit) - Level.unity - SpaceWar - PC, Mac & Linux Standalone <DX11>
```

Score: 4.670

```
Statistics
Audio:
  Level: -74.8 dB              DSP load: 0.9%
  Clipping: 0.0%               Stream load: 0.0%
Graphics:                      49.9 FPS (20.0ms)
  CPU: main 20.0ms  render thread 1.4ms
  Batches: 10          Saved by batching: 0
  Tris: 82   Verts: 210
  Screen: 1500x830 - 14.2 MB
  SetPass calls: 10    Shadow casters: 0
  Visible skinned meshes: 0  Animations: 0
```

Overview

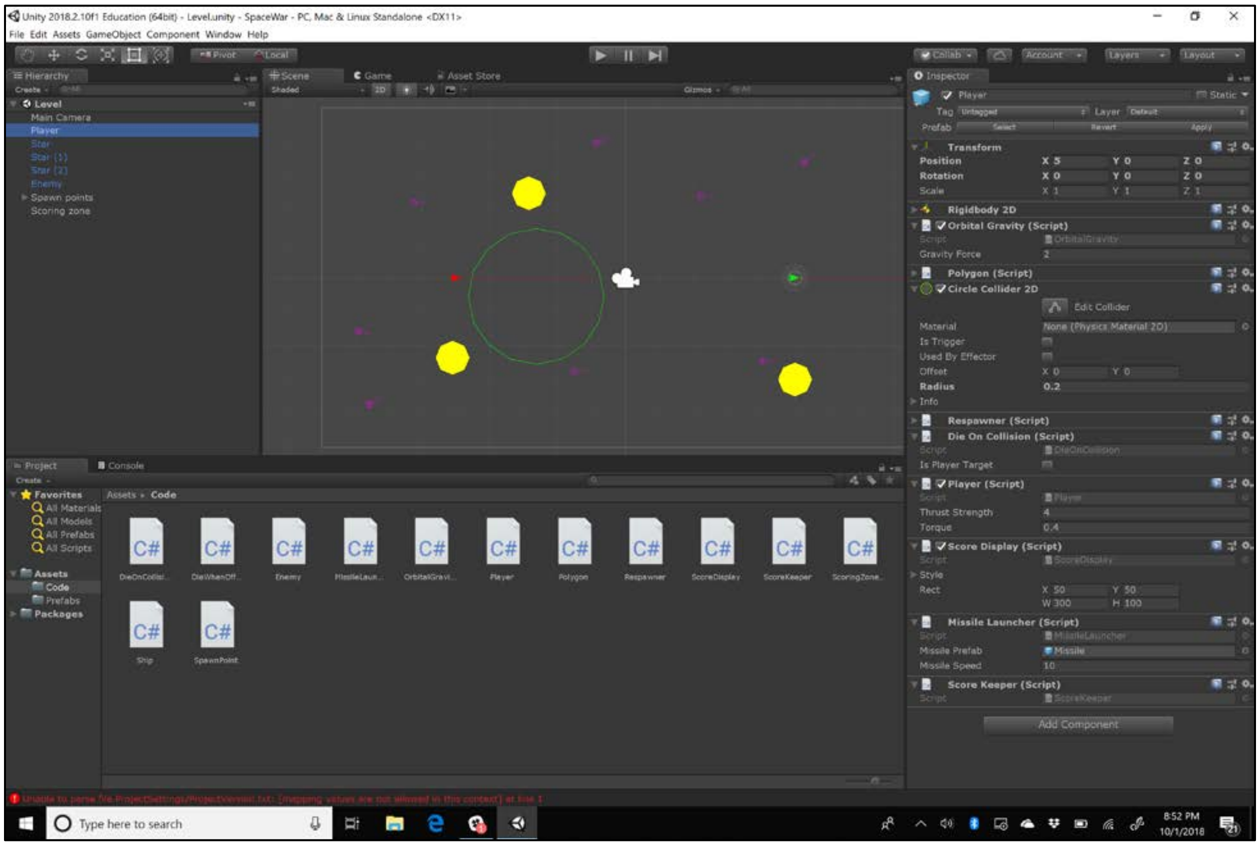
In this assignment, you'll get to know the basics of Unity's architecture by adding functionality to a simple game based on SpaceWar!, one of the first video games ever made. Begin by opening the Level file in the Assets directory, which should launch Unity for you.

How to play

The game is very simple. You're the green ship. You steer with the left and right arrow keys and you accelerate with the up and down arrows. You can fire missiles with the spacebar.

You score points when you shoot the red ship. You can also score points by staying inside the green circle in the middle of the screen, so you can do pacifist play if you prefer. But watch out for the yellow stars, that suck you in with their gravity.

Basic Unity architecture



The main kinds of objects in Unity are:

- **GameObjects**
  Unity primarily represents the running state of the game as a set of GameObjects. In general, anything that appears on screen has a GameObject that corresponds to it (although there can also be GameObjects that don't appear on screen). In general, when Unity wants to do something, it starts by looking through all the game objects.

  GameObjects don't have any real behavior by themselves. They're really just lists of Components (that do all the real work), and optionally other GameObjects (children). Since GameObjects can be in other GameObjects, they form a tree.

  The tree appears on the left in the Unity editor (the Hierarchy view). When you select a GameObject on the left, its components appear in the view on the right (the Inspector).

- **Components**
  Components are objects that add some functionality to a GameObject. By implementing different kinds of functionality as separate components, the designer can mix and match them within a given GameObject as they like, adding only the components they need. They can also easily make several GameObjects that are almost alike, but where one might have some extra behavior by virtue of having an extra component, without having to create a new subclass.

Most of the Unity API is exposed to programmers through a set of built-in components. For example:

- The **Transform** component
  Represents the position, size, and orientation of the GameObject containing it on the screen.
- The **RigidBody2D** component
  This adds physics behavior to the GameObject, automatically updating the object's Transform based on the object's momentum and any collisions that occur between it and other objects. It also allows code to apply forces to the object to move it.
- The **Collider2D** component
  Tells the physics system about the area of the game world that its GameObject occupies. When the colliders of two GameObjects touch one another, the physics system responds, and all the components of the two GameObjects are informed by calling their OnCollisionEnter2D methods, if they have them. There are different subclasses of Collider2D depending on the shape of the area they specify. In this case, we only use CircleCollider2D objects, which specify circular areas.
- The **AudioSource** component
  This plays sound clips (AudioClip objects) through your sound card.
- The **MonoBehaviour** class (not British spelling)
  This is the base class that Unity requires you to use when you make your own components. Most code files in a game define subclasses of MonoBehaviour.

- **Prefabs**
  These are templates for GameObjects. They contain the data for the desired game object – all its components, and the values of their fields – so that you can rapidly create new game objects based on them. In addition, the Unity editor remembers what Prefabs a given game object is based on so that if you edit the underlying prefab, any GameObjects created from it will change accordingly.

- **Levels** (aka **Scenes**)
  A level is a template for an entire hierarchy of GameObjects. When the game starts, it reads the level file for the initial level, thereby creating the game's initial set of GameObjects, and starts them running. The Unity editor is ultimately an editor for these levels. Note that these are called "levels" in the API documentation but "scenes" in the menus in the Unity editor. Sorry.

- **AudioClips**
  These are audio files that can be fed to an AudioSource component to play, using its Play or PlayOneShot methods.

- **Sprites**
  These are simple images to be displayed on the screen. They aren't used in this game, but

they're mentioned here since most games do use them.

- **Assets**
  These are the code and media files from which the objects above are originally loaded or compiled. They include WAV or MP3 files for audio clips, JPG or BMP files for sprites, .unity files for levels, .prefab files for prefabs, and .cs files for the source code for your MonoBehaviours and other C# code. All these files are placed in the Assets folder of your project.

For example, in the game you'll be working with, there is one level. It contains a set of GameObjects. The ones you'll pay the most attention to are the ones named Player and Enemy, which implement the player's ship and the NPC ship, respectively. The Player object has a number of components, such as:

- Transform
- RigidBody2D
- Player component (defined in Player.cs; it's the keyboard handler for the ship)
- OrbitalGravity (causes the ship to accelerate toward nearby stars), MissileLauncher (code to fire missiles)
- Polygon (renders the triangle representing the ship on screen)
- ScoreKeeper (notices when the player scores or loses points)

The Enemy GameObject is mostly the same, except it has an Enemy component, which implements a trivial AI for the ship, rather than a Player component. The Enemy also doesn't have a ScoreKeeper.

What you should do

Start by looking through the code and getting a rough idea of how the game works. This is an important skill to develop as a programmer, if you aren't used to it already. Don't expect yourself to understand it entirely, or even mostly, at this point. But you can look at the names of types and methods and they tell you a lot.

All you have to do for this assignment is to add sound to the game. In particular:

- When a ship is destroyed it should make some kind of an explosion noise
- When a ship fires a missile, it should make some appropriate noise
- When the player scores a point by blowing up the enemy with a missile, it should make some appropriate noise.
- Feel free to add more sound triggers if you like, but these are the only ones we'll grade you on

To make this work, you need to:

- Add some sound assets to the game.
  There are two big directories of them in Exercise 0. The easiest thing to do is to copy those directories into the Assets directory of this game. But feel free to use other sound files if you prefer. **You must not use big sound files**, however, or Canvas won't be able to accept your submission. Plus, your peer reviewers will get annoyed if you have it play the Brandenburg Concertos every time you fire a missile.

- Add AudioSource components to the Enemy and Player game objects
  Add these to the Prefabs of those game objects (go to the Prefabs directory in the editor and click on the associated prefab, changes you make there will update in the game objects in levels). By making the change to the prefabs, you make it easier to make new levels without having to add the AudioSource over again to the objects in each level. Also, you make it possible to have multiple Enemies.

- Add AudioClip fields to the Player and Enemy componets (in Player.cs and Enemy.cs) to hold the different sounds you want to play. Make sure to make these public fields. The editor will automatically display fields that are public.

- Fill in the new AudioClip fields in the editor
  Again, do this in the prefabs. To set an AudioClip field, select the desired game object on the left. Then you will see its components on the right. Find the Component with the AudioField you want to set. To the right of the field, you'll see a circle with a dot; click it, and the editor will pop up a list of all the AudioClips in your project. Double click one to set the field to it. To listen to the clip, click it in the list, then click its waveform at the bottom of the list.

- Find the places in the code where missiles are fired, ships are destroyed, and points are scored, and modify them to play the appropriate audio clip when those events occur. You'll want to use the PlayOneShot method of the AudioSource component.

Optional: make your own level

If you like, you can try editing the level. I'd strongly suggest you make a new level, if you do so (just choose Save Scene As in the editor to make sure it's saving it to a new file). Now you can try adding new Enemies, changing how fast the ships move or turn or shoot, move the stars around, add new stars, or remove them entirely. However, this is all entirely optional. You'll only be graded on whether your game makes sounds in situations we specified.

Turning it in

First exit out of Unity. Now make a new directory and copy into it just the Assets, Packages, and ProjectSettings directories of your project (don't include Library, Temp, or obj – they're huge). Now make a ZIP file of the new directory and upload it to canvas!