Lab - Space Invaders

Goals

- To build a simple game in Unity2D
- To learn about collisions.

- To learn how to keep the game state.
- To learn how to load different scenes.
- To get familiar with the game base for the first assignment.

Resources

- SpaceInvaders project (https://altitude.otago.ac.nz/cosc360/SpaceInvaders)

Preparation

- Familiarise yourself with C# syntax (https://learn.unity.com/course/beginner-scripting).
- Read about the rigid body (http://docs.unity3d.com/Manual/RigidbodiesOverview.html) component.
- Read about colliders (http://docs.unity3d.com/Manual/CollidersOverview.html).
- Watch video tutorial on 2D Physics (https://learn.unity.com/tutorial/2d-physics)

What to do

This lab is a chance to see how a simple game can be implemented in Unity2D, to become more familiar with C#, and to get a head start on the week's assignment.

About Assignment 1

This lab is also a prerequisite for your first Assignment. The Assignment is a continuation of the game developed in this lab. So, it's especially important to understand how this game works. It's a long lab, but at the end of it, you should know enough to make a complete game in Unity.

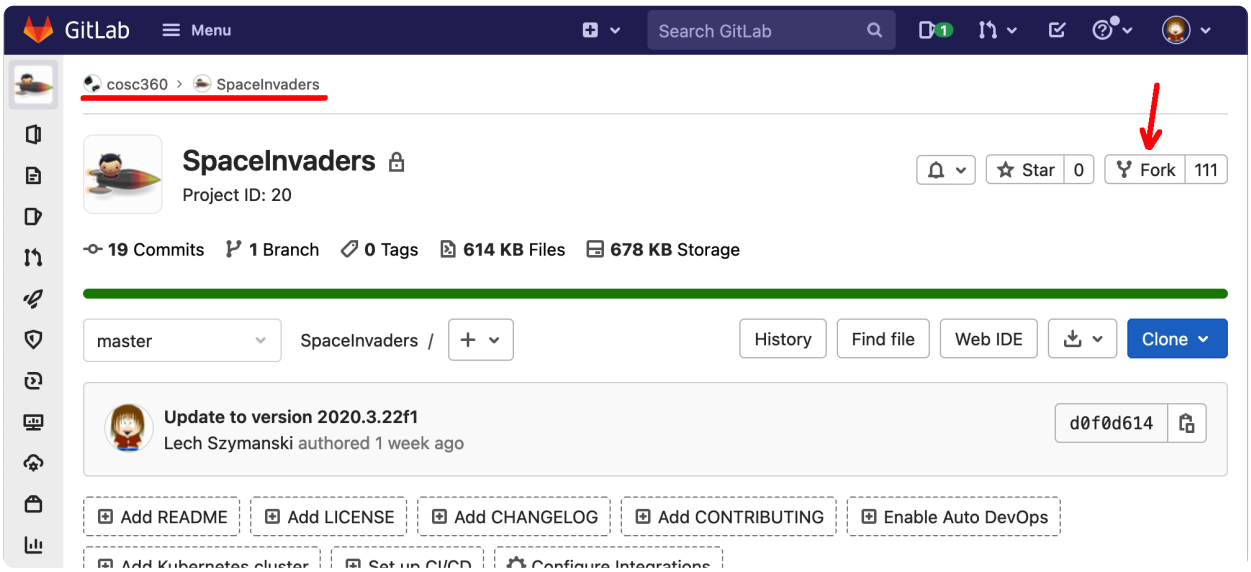
Fetch resources from GitLab

In this paper you'll be eventually working in a group using GIT (https://git-scm.com/) version control system in order to co-ordinate development of the same Unity project. The resources you need for this lab consist of a ready made project that has been posted on CS's internal GitLab (https://altitude.otago.ac.nz) server. Details on how GIT works will follow in Version Control Lab (../lab03/index). In this lab, just follow the instructions, and don't worry too much about the details of the version controls system.

1. Open the cosc360/SpaceInvaders (https://altitude.otago.ac.nz/cosc360/SpaceInvaders) project on Gitlab. This will open in your browser. If you're asked to login, login with your CS account. If, after first ever login to

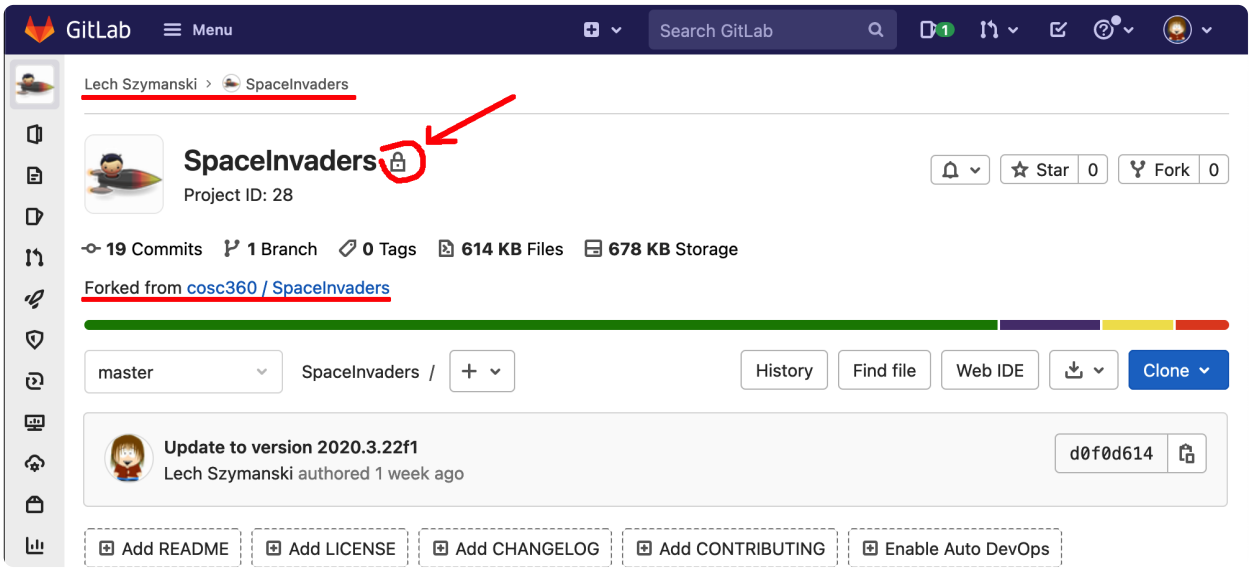
Gitlab you get a 404 page error, it's because it takes a few minutes for the permissions to access the SpaceInvaders project on GitLab to update - wait a few minutes and try again.

. You should find yourself on the "cosc360/SpaceInvaders" project page. Everyone from the class should have read access to this project. You are going to fork the project - that is to make your own private copy of that project on GitLab. Click on the "Fork" button



. A new window will let you make a choice where to Fork. Your username should be one of the choices - select it.

. GitLab will create a copy of the project repository at "https://altitude.otago.ac.nz/<your user name>/SpaceInvaders".

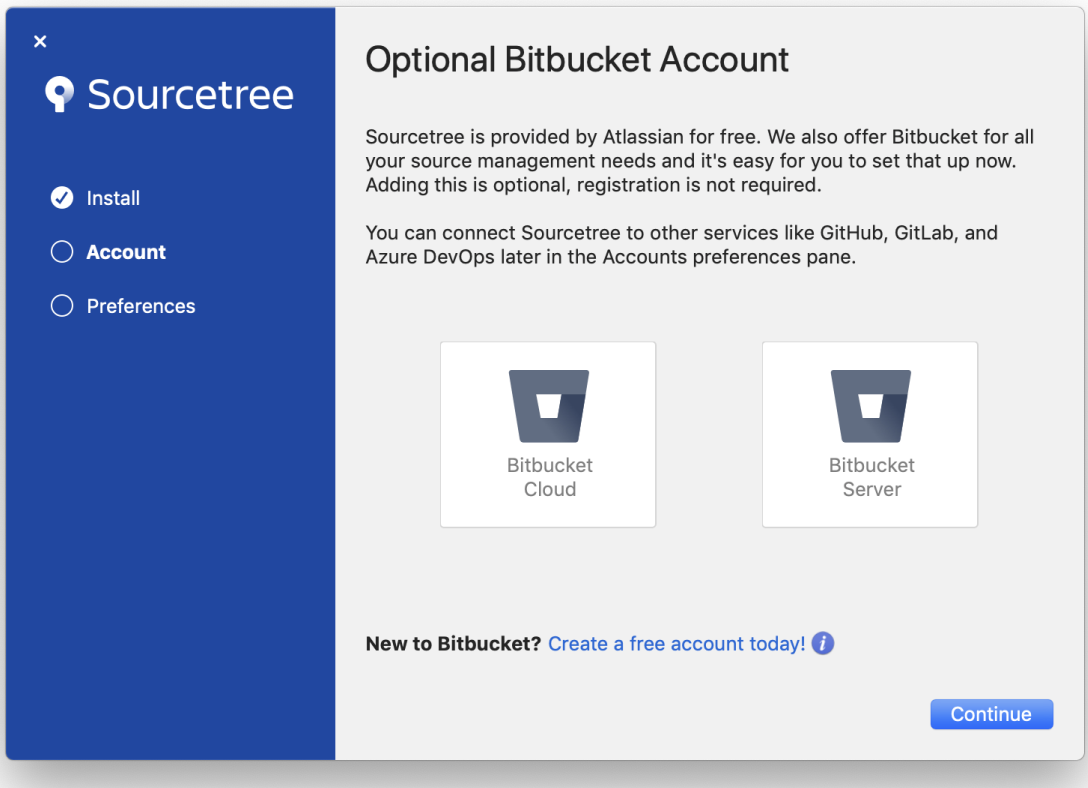


. By default, your project should be private - there should be a lock icon (which stands for "private") on the right hand side of the project name, as highlighted in the above screenshot.

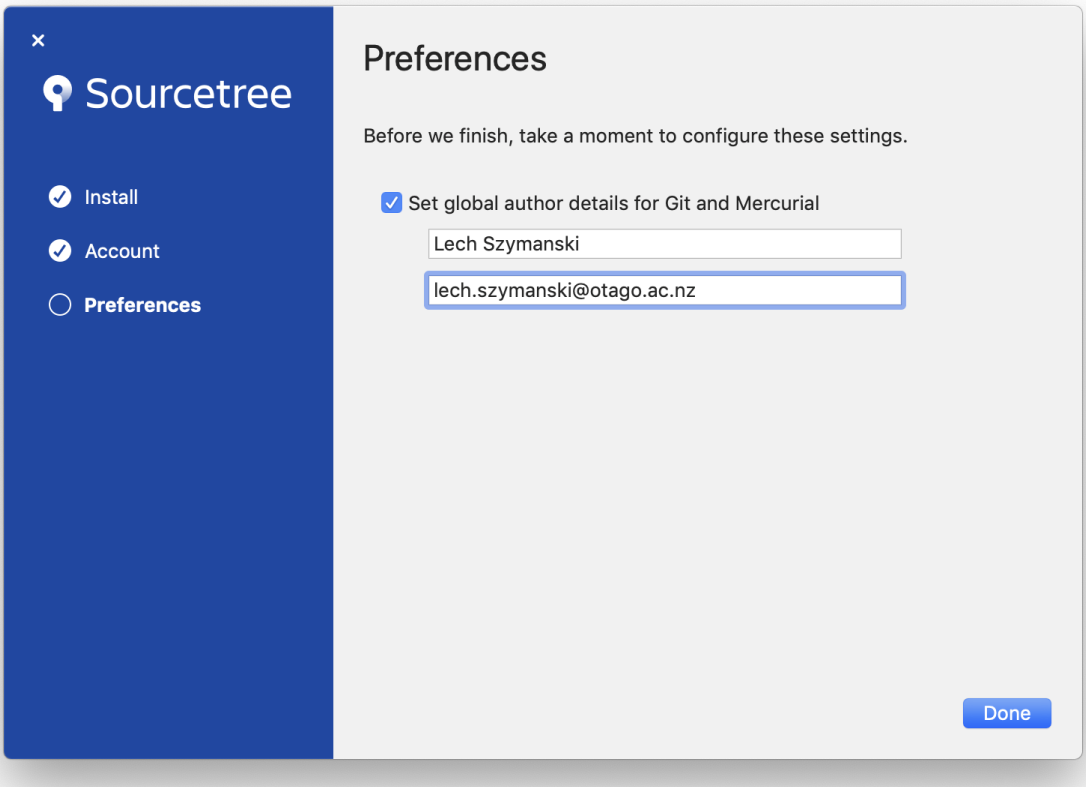
. Now that you have a GitLab copy of the project repository, you need to copy it to your hard-drive. But first, you need to set up secure communication between the GitLab server and your machine. This is done through the SSH protocol. You need to generate a pair of RSA keys on your machine and upload the public key to GitLab. This way, GitLab will be able to encrypt data that only your machine (using the matching

secret key) will be able to decrypt. If you know how to set this up, go ahead and do it. If not, follow the instructions below.

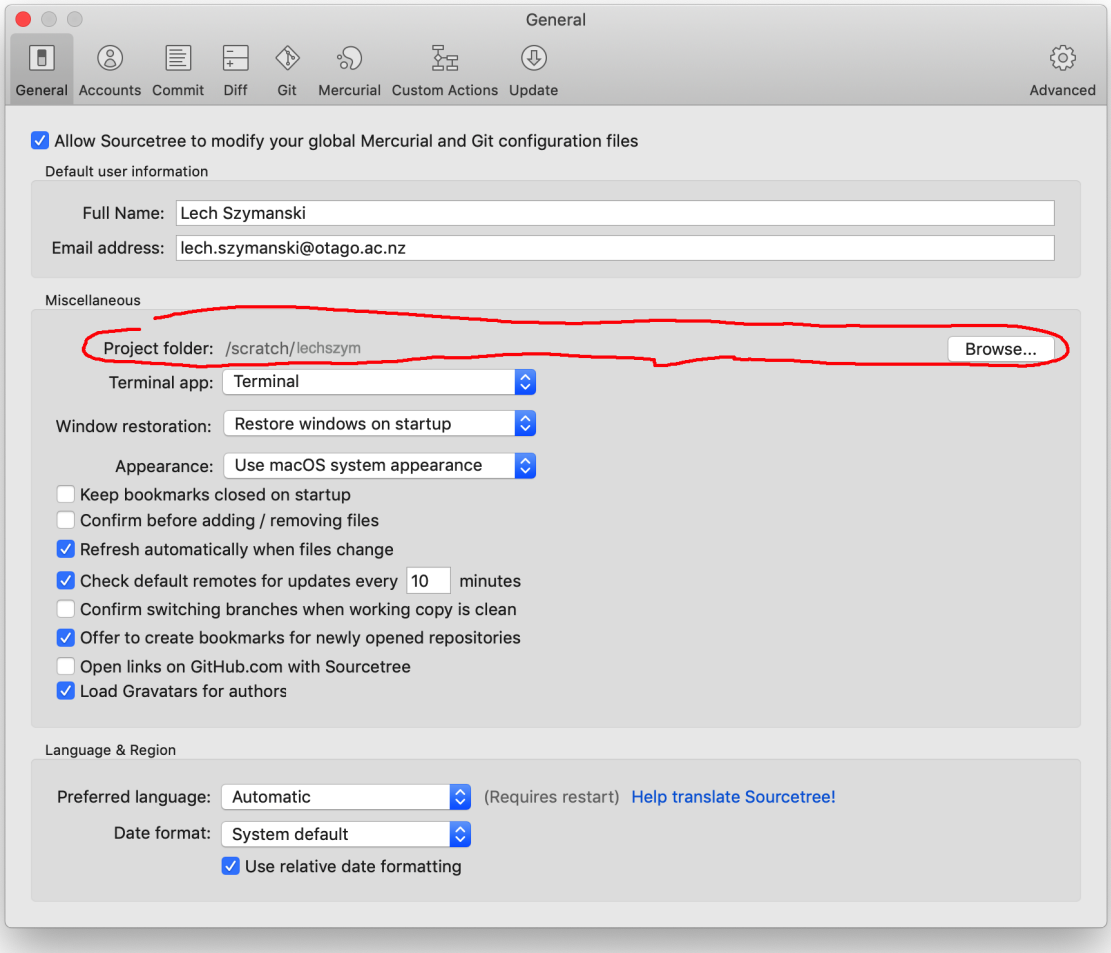
. Open the SourceTree application (the icon in the Dock or Launchpad). SourceTree is an application for managing GIT repositories.



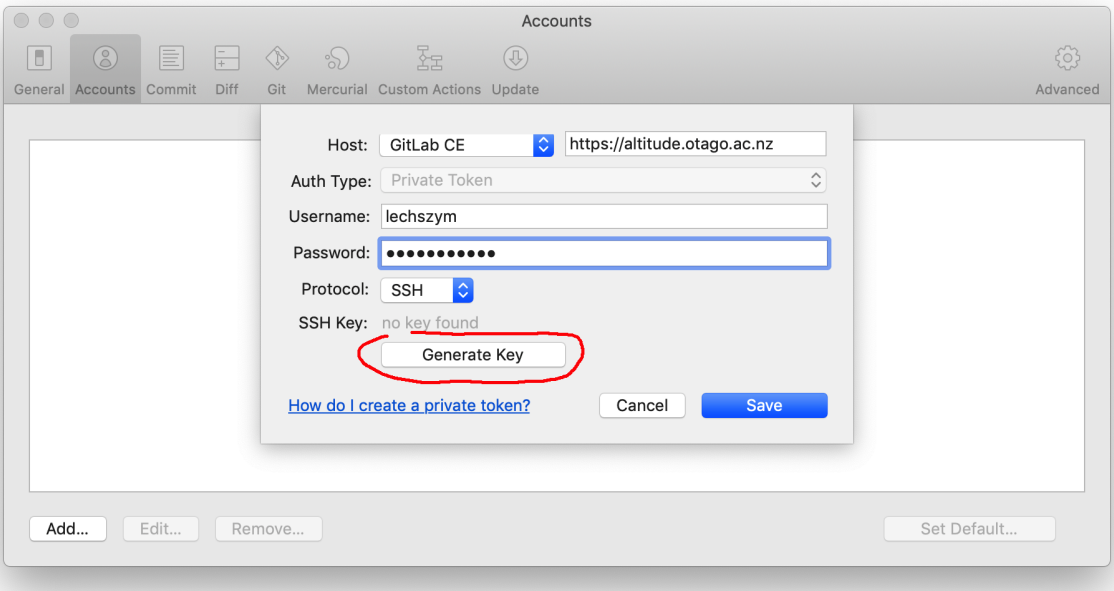
. If it's the first time you are opening SourceTree, "Continue" pass the "Account" screen to "Preferences" and fill out your credentials. Press the "OK" button



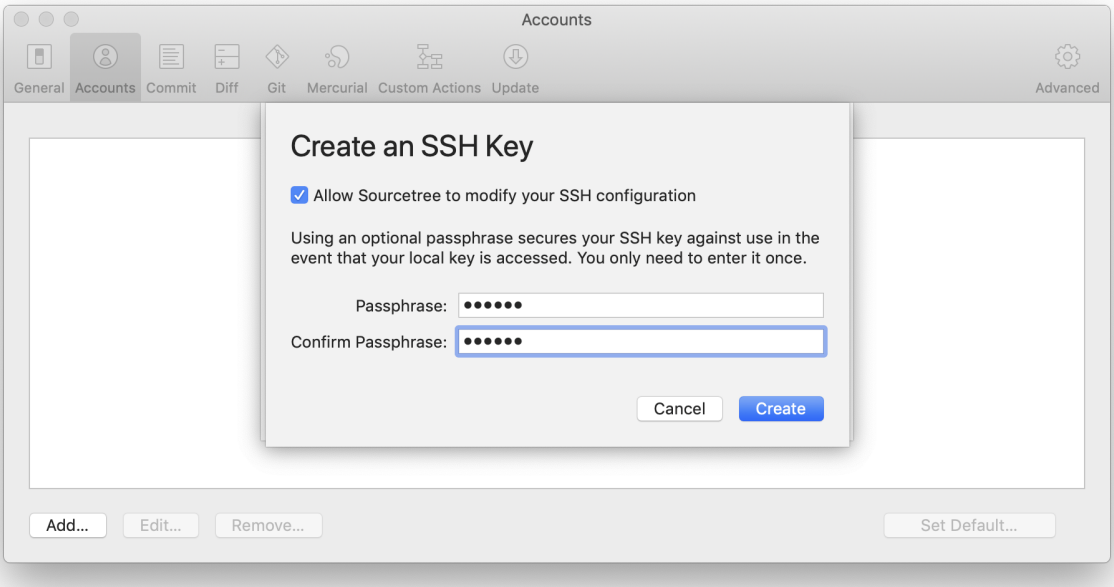
. From the main menu select "Source Tree->Preferences". In the "General" tab, under "Miscalleneous", set the "Project folder" to "/scratch/<your usernmae>" - this will make checking out of new code much easier.



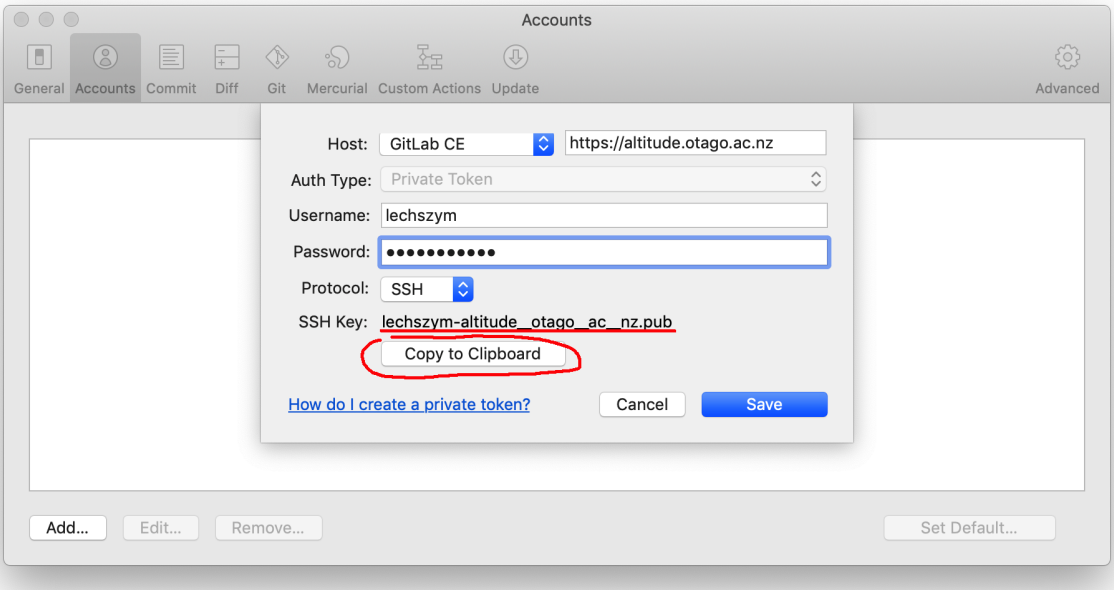
). In the "Accounts" tab, click "Add" and setup the credentials for the GitLab account. For the host select "Gitlab CE" and set the respective URL to "https://altitude.otago.ac.nz". For user credentials use your CS username and password. Select "SSH" for the protocol click the "Generate Key" button.



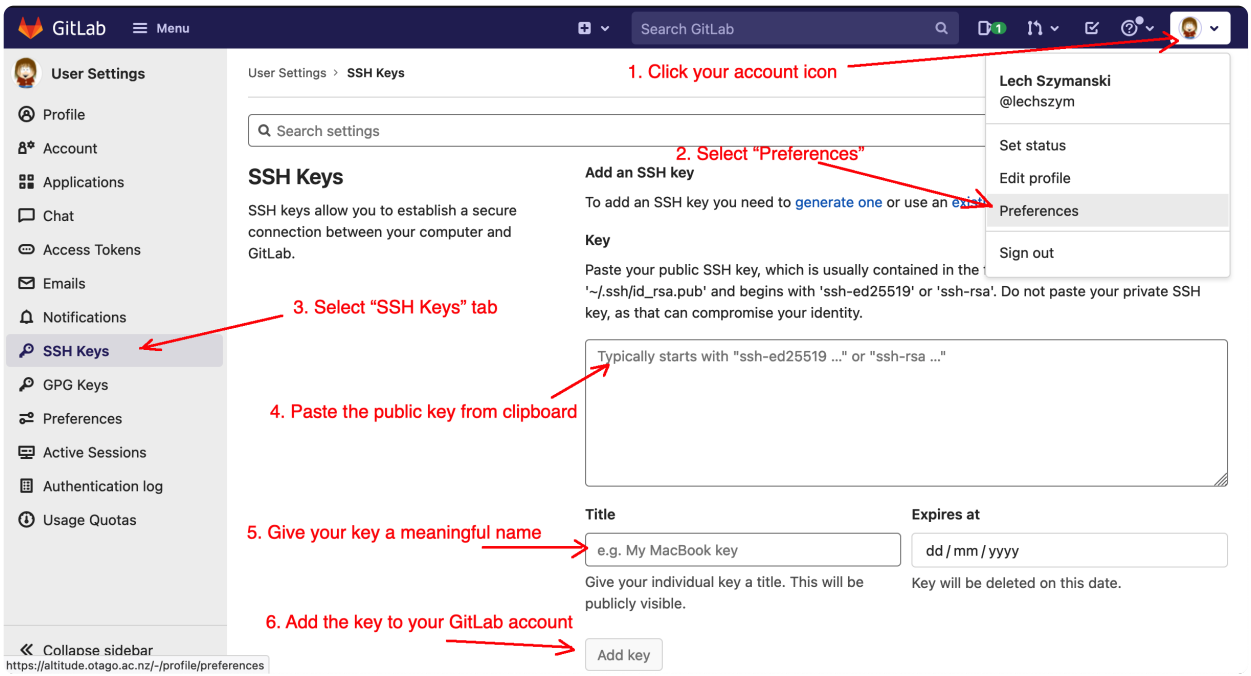
L. Once "Crate an SSH Key" dialog pops up, type in a passphrase of your choosing and click "Create".



2. You should see an entry under the "SSH Key" now. Click on "Copy to Clipboard" - this will copy the public key to clipboard. Click on the "Save" button. If Source Tree refuses to save the account (this might happen if you have two factor authentication enabled on GitLab), don't worry about it - just click "Cancel". The important thing is that you have the public key saved to clipboard.



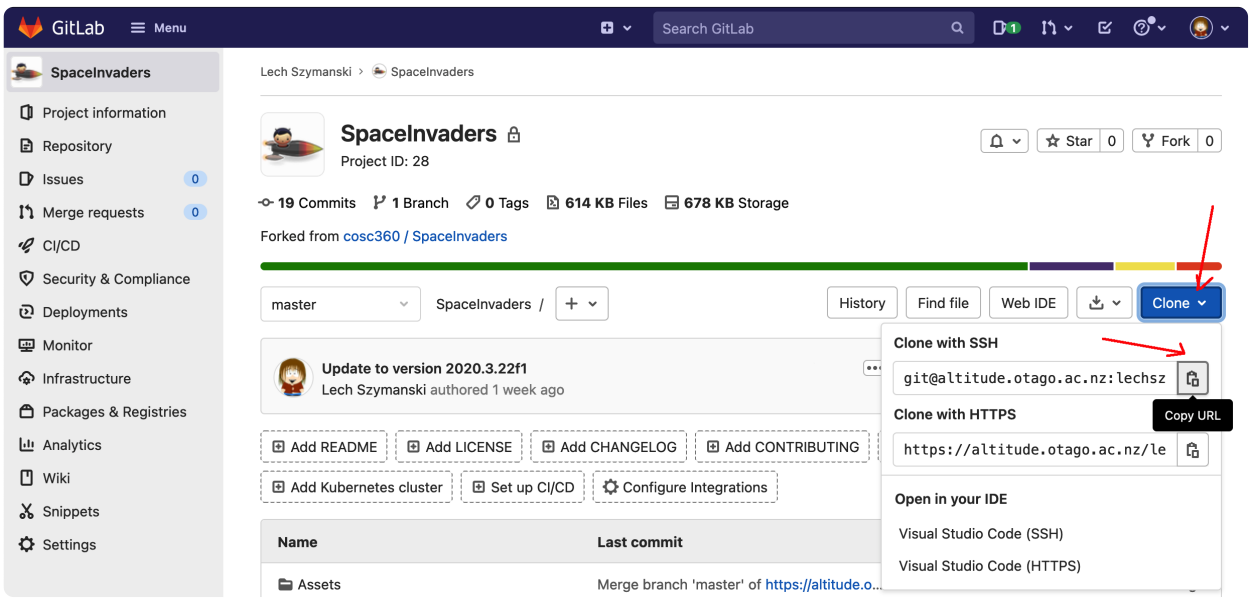
3. Switch to your browser to the GitLab page on https://altitude.otago.ac.nz. Add the ssh key from your keyboard to your GitLab account following the steps given in the screenshot below.



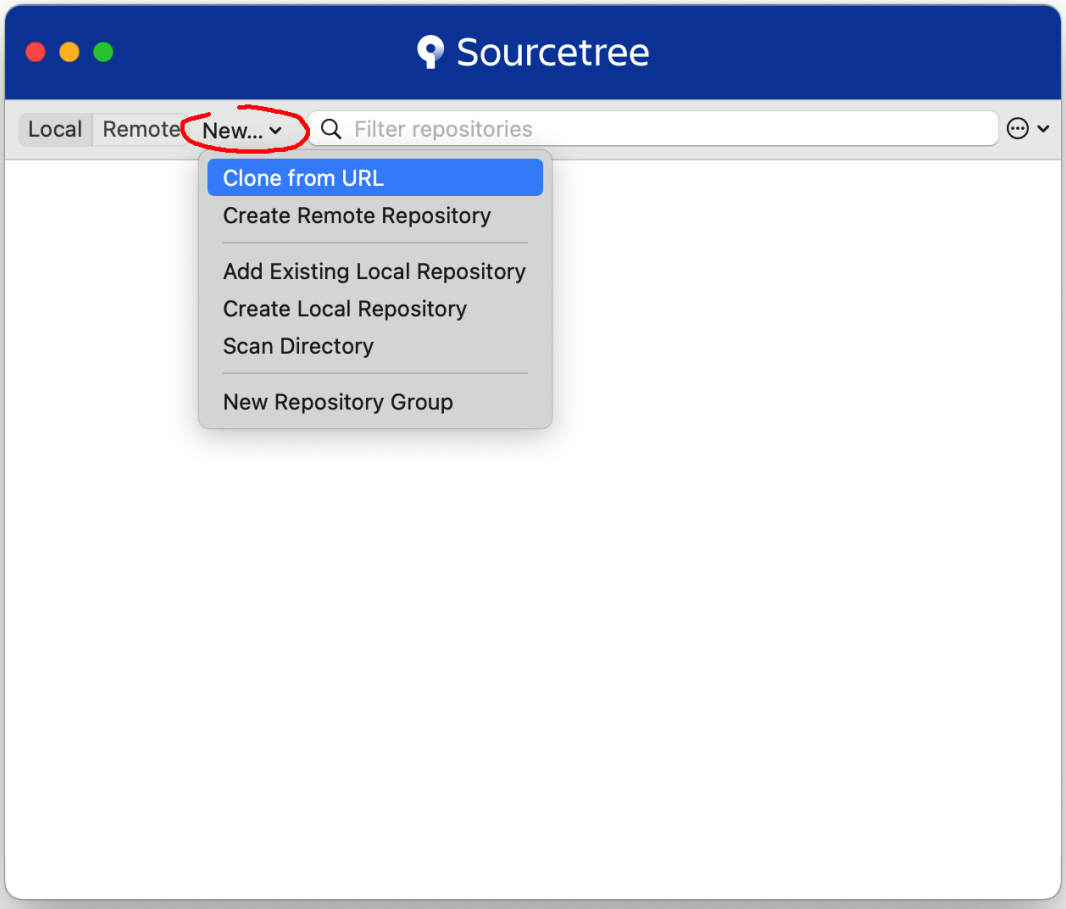
SSH key for GitLab at home

If you want to work at home, you will need to do an SSH key setup from GitLab from your home machine.

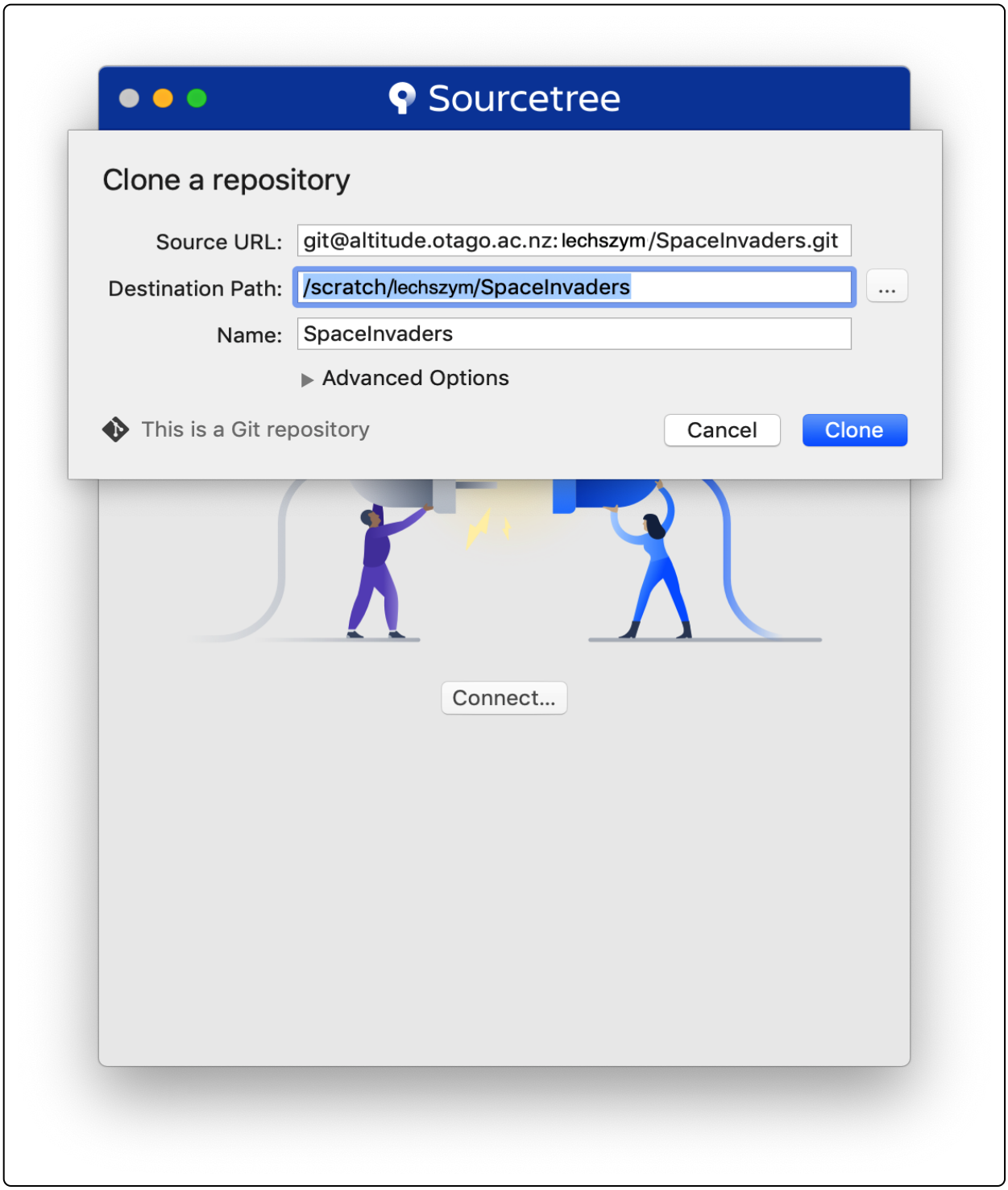
4. Next, get back to the page of your copy of the forked Space Invaders project (at "https://altitude.otago.ac.nz/<your user name>/SpaceInvaders"). Click on the "Clone" button and copy to clipboard the projet's ssh url (under the "Clone with SSH").



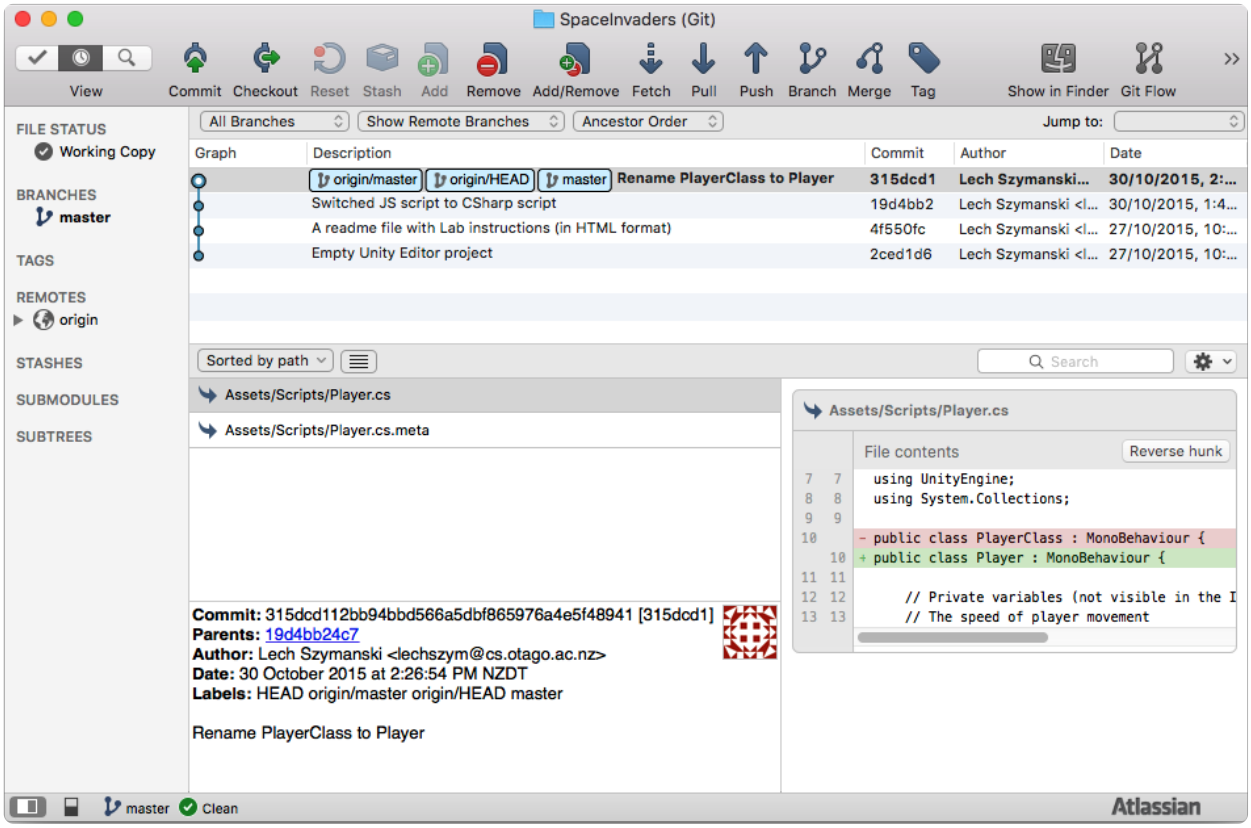
5. Return to SourceTree. Next, you will instruct it to copy your SpaceInvaders project to your hard-drive. Click on the "New Repository" button. Select the "Clone from URL" option.



3. In the window that opens, paste the SSH url you grabbed from GitLab into the "Source URL" box. That URL should be "git@altitude.otago.ac.nz:<your username>/SpaceInvaders.git" and NOT "git@altitude.otago.ac.nz:cosc360/SpaceInvaders.git". For destination select "/scratch/<your username>/SpaceInvaders.



7. Press the "Clone" button. SourceTree will fetch the project from GitLab and download it to your hard-drive. A window will open showing the history of the project - a series of snapshots as changes were made to it.

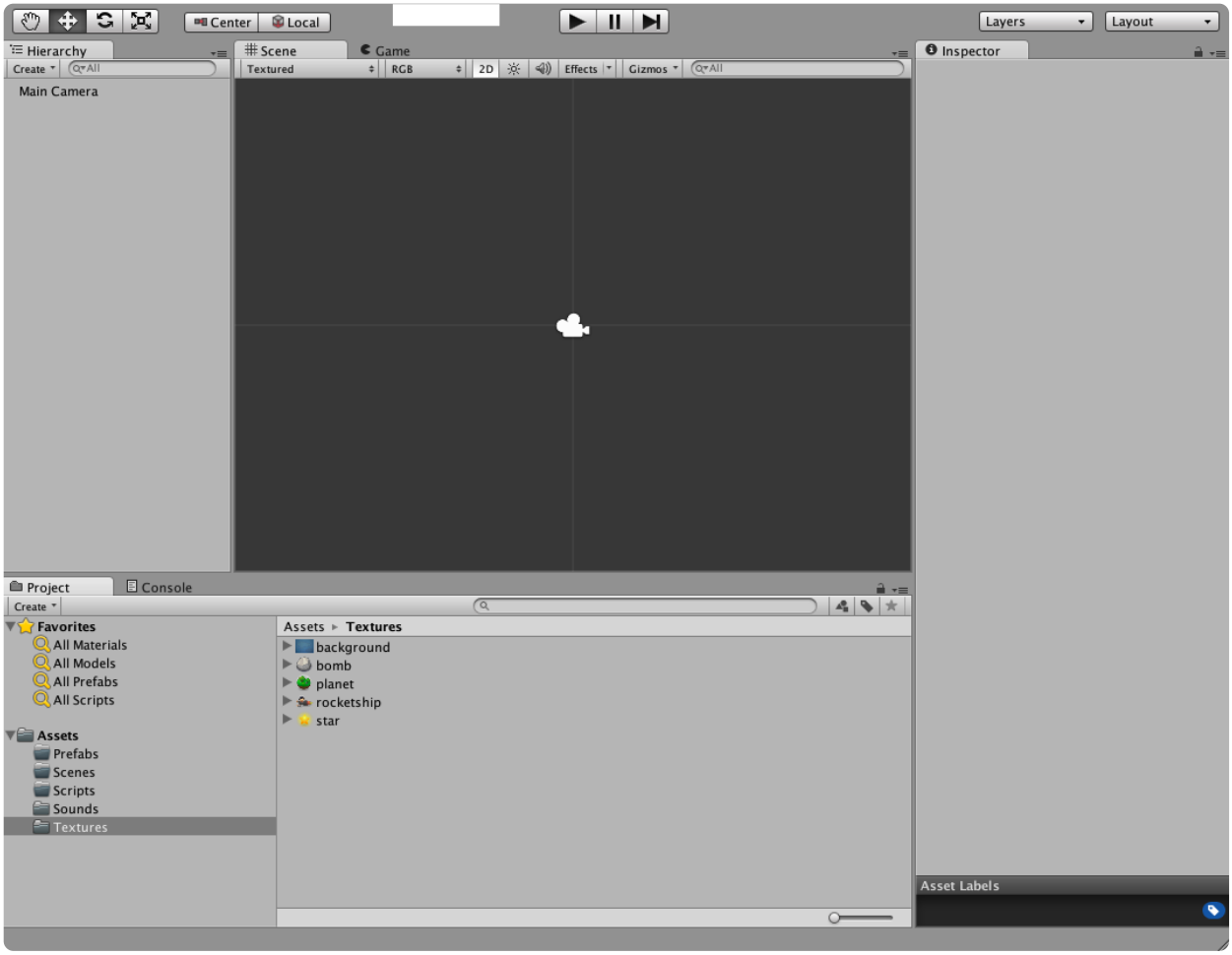


3. Leave SourceTree open (you'll come back to it in a bit), but now it's time to fire up the Unity Editor.

Set the target platform

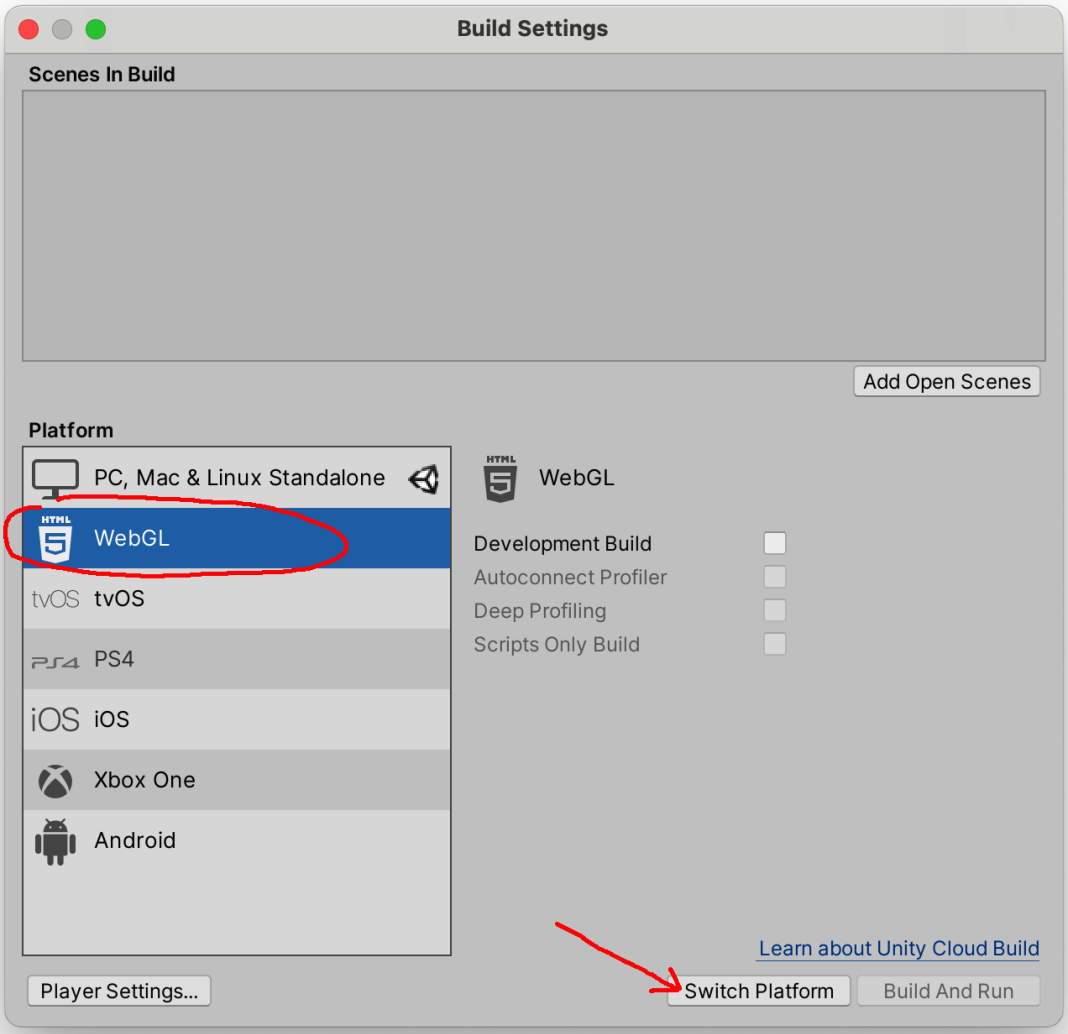
3. Start the Unity Editor.

3. Click on the *Open* icon and navigate to the "SpaceInvaders" folder cloned from GitLab. It's a ready made Unity2D project. Point the dialog box to the folder and click "Open". Unity Editor should load the project.



1. While Unity caters to differnt target platforms, such as Windows, Mac, Android, WebGL, etc., there are subtle differences between them. For instance, some aspect of graphics might produce different artifacts on different platforms. Hence, it is good practice to make sure that the Editor is configured for the correct target platform. For this lab, as for Assignment 1 and your main game in this paper, the taget platform is WebGL.

2. From the main menu select *File->Build Settings...*. In the windows that pops up, select WebGL fo the platform and click on the "Switch Platform" button. The project should recongifugre itself for your platform.



3. **Remember to do switch platform to WebGL right away for your main project, later on, when you create a new Unity project for the game you'll be developing for this course!**

Create the main scene

1. Notice that the *Assets* have been organised into a number of subfolders: *Prefabs*, *Scenes*, *Scripts*, *Sounds* and *Textures*. It's a good idea to keep your assets organised - it makes it easier to find things as the game content grows. For this project, the sprites can be found in the *Textures* directory (all that we need for this game has already been imported). Anything to do with audio resides in *Sounds* (imported already as well). Scripts will go into *Scripts* (just one script there at the moment). Scenes will be saved into *Scenes*, and prefabs (which we'll encounter soon enough) will get saved into *Prefabs*. If later on you want to organise things in a different way, its easy to create these subfolders either in the *Assets* window of the **Project panel** or in the *Assets* directory of your project. For now leave things as they are.

Assets location

Once your assets are used in a scene, it's not a good idea to move them into different subfolders; the editor might not update the references from game objects to new locations.

The *Scenes* folder

Gitlab usually removes empty folders, and so you might see a warning from the Unity Editor about *Assets/Scenes* being removed because the asset does not exist. Just right-click inside the *Assets* and create a new folder. Name it *Scenes*. Next, you'll be saving the new scene into that folder.

5. Create a new scene for the main level of the game. From the main menu select *File | Save Scene*. Name the scene "Level1" and save into the *Assets/Scenes* folder.

6. The scene already contains the *Main Camera* object. Time to add more objects.

7. In the **Hierarchy panel** click on *Create | Create Empty* . A new object, called *GameObject*, should appear. The object is empty, aside from its *Transform* component (recall, every game object has a *Transform* component, which specifies its location in the scene). Name the new object *Background* and set its *Transform | Position* to (0,0,0).

8. Create another empty game object. Name this one *Foreground* and set its *Transform | Position* to (0,0,-1).

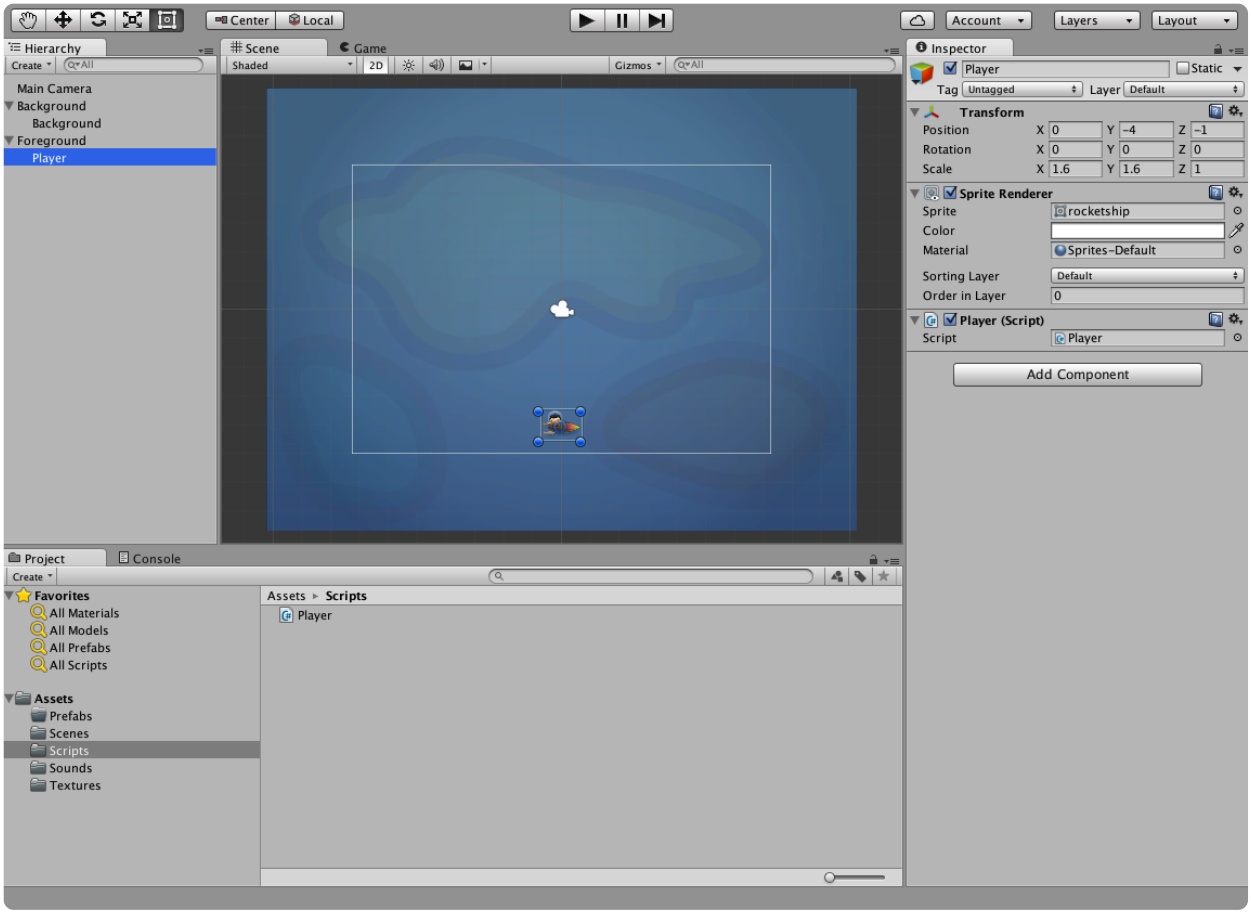
9. The two objects that you have just created are going to function as parent objects. Unity allows you to organise game objects in a tree structure hierarchy, where game objects can be parents or children of other objects. The position of the child object is always relative to its parent. Recall from the first lab, that the *Main Camera* (by default) is at position (0,0,-10) looking towards the positive direction of the Z-axis. Therefore, from the camera's point of view, the "Foreground" object (at z=-1) comes before the *Background* object (at z=0). Hence, the children of the *Foreground* object, whose local z-position is less or equal to 0, will be rendered in front of the children of the "Background".

10. Now, create a new *2D Object/Sprite* object. Name it *Background*. Set its *Sprite Renderer | Sprite* property to the "background" image. Set the *Transform* properties as follows: *Position* to (0,0,0) and *Scale* to (2,2,1). In the **Hierarchy panel** drag the newly created *Background* sprite object into the existing *Background* game object. Yes, game objects can have duplicate names - whether you think it's a good idea to take advantage of that feature or not, is up to you.

11. Create another sprite game object. Name it *Player*. Set its *Sprite* to "rocketship", *Position* to (0,-4,-2) and *Scale* to (1.6,1.6,1). In the **Hierarchy panel** drag the *Player* game object into the *Foreground* object. Note that the z-position of the Player might have changed. This is because, when you change the parent structure of an object in the hierarchy panel the editor preserves the global position of the object by changing its relative position. Just make sure that after making *Player* a child of the *Foreground*, the z-position of the *Player* is -1 (since *Foreground*'s z-position is -1, then *Player* relative z-position of -1 makes it located at z-position -2 in global coordinates).

12. Add a *Script | Player* component to the *Player* game object.

13. Save the scene. At this point your project should look like the screenshot below:



When you press the play button, you should be able to move the player left and right with the arrow keys.

1. Let's examine the script that makes the player object move. In the **Assets** select *Scripts/Player* and double-click to open it in VSC editor.

Player.cs

```
01: using UnityEngine;
02:
03: public class Player : MonoBehaviour {
04:
05:    // Private variables (not visible in the Inspector panel)
06:    // The speed of player movement
07:    float speed = 10;
08:
09:    // Update is called once per frame
10:    void Update () {
11:       // Player movement from input (it's a variable between -1 and 1) for
12:       // degree of left or right movement
13:       float movementInput = Input.GetAxis("Horizontal");
14:       // Move the player object
15:       transform.Translate( new Vector3(Time.deltaTime * speed * movementInput,0,0),
16:    }
17: }
```

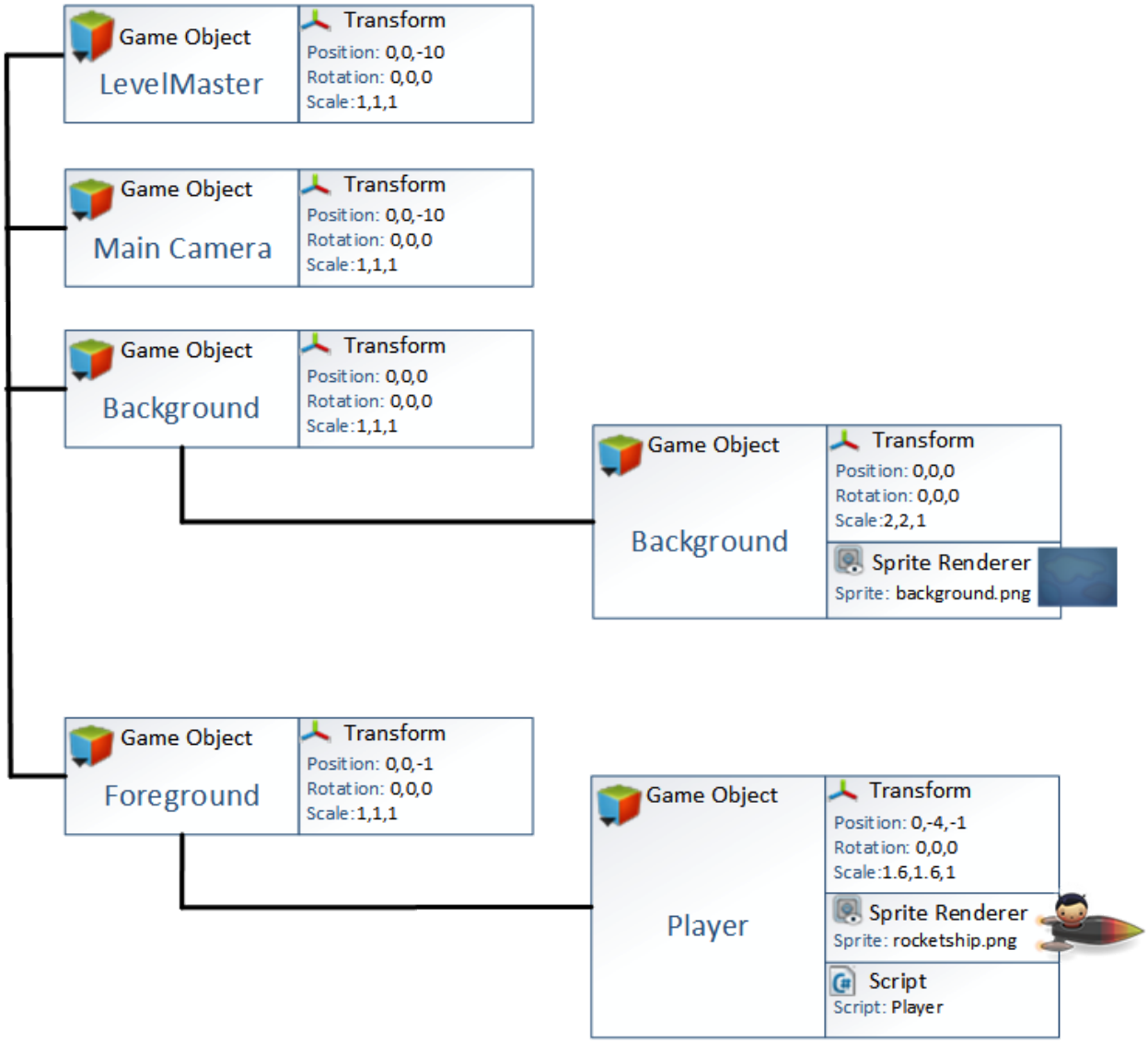
User input is detected in the call GetAxis() method of the *Input* (http://docs.unity3d.com/ScriptReference/Input.GetAxis.html) class. It's a convenience method provided by Unity, which detects user input based on a set of pre-defined keyboard/controller keys. For instance, at the moment, the player can be controller by left and right arrow or by the 'a' and 'd' keys.

Keyboard/controller settings

To view/change the keyboard/controller settings associated with *Input.GetAxis()* method, from the main menu select *Edit | Project Settings | Input*. The properties of the *Input Manager* should appear in the **Inspector Panel** where you can view (or change) the current settings for *Axes*.

The GetAxis() function returns a value between -1 and 1. The sign signifies direction of the movement (negative is movement to the left, positive to the right). The value specifies the amount of movement (-1 meaning maximum movement to the left, 1 meaning maximum movement to the right). This value gets stored in the *movementInput* variable. To compute the translation of the *Player* game object a shift vector is created with its X coordinate equal to: the time since the last Update() times the *speed* constant times the *movementInput* (for direction). The Y and Z coordinates are zero. Passing this vector to the *Translate()* method of the game object, referenced by the *transform* variable, updates the position values of the object for the next time it is rendered.

5. The **Hierarchy panel** gives you a good overview of the object hierarchy in the scene. Here's a diagram of that hierarchy with few more details:

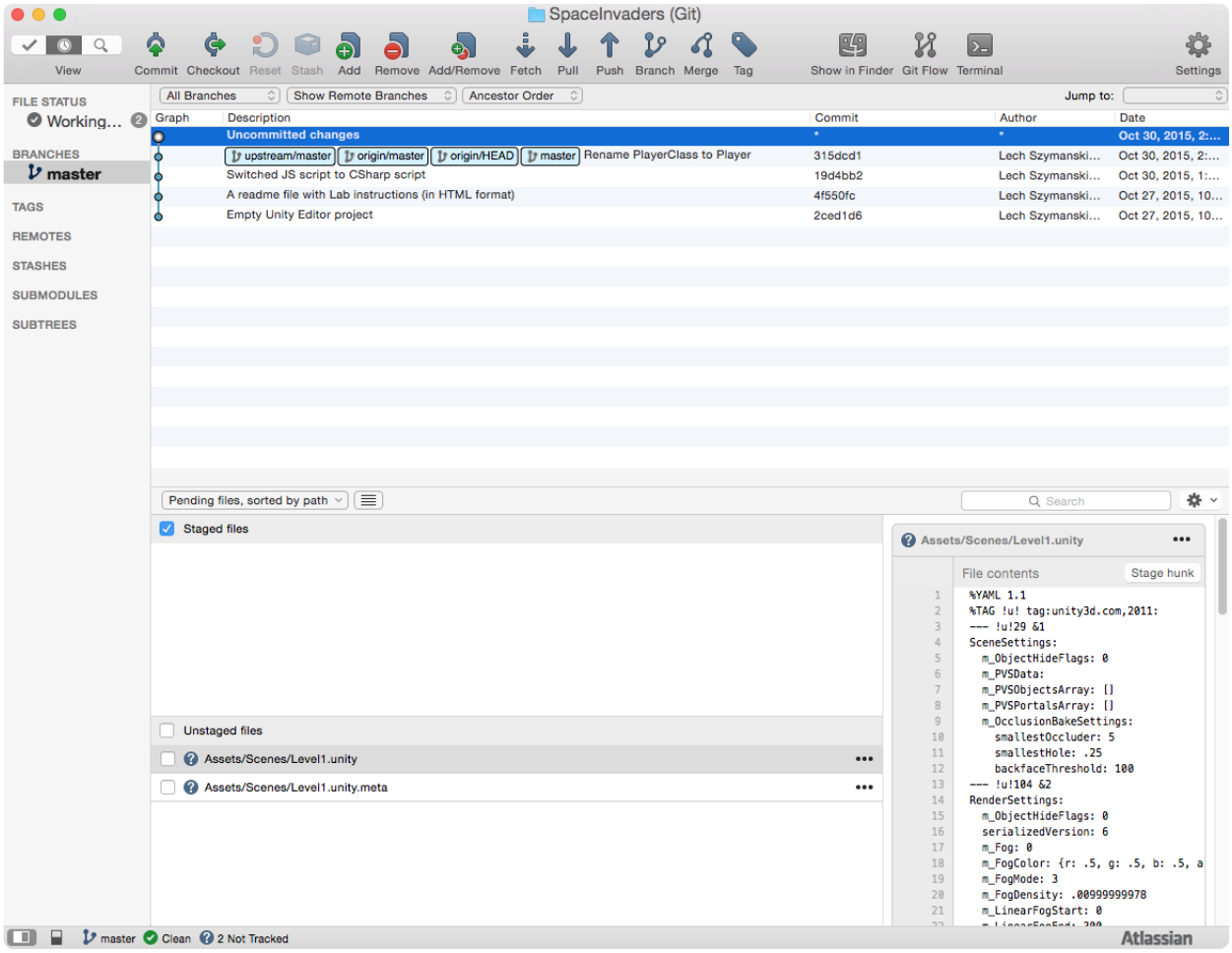


Commit progress to git

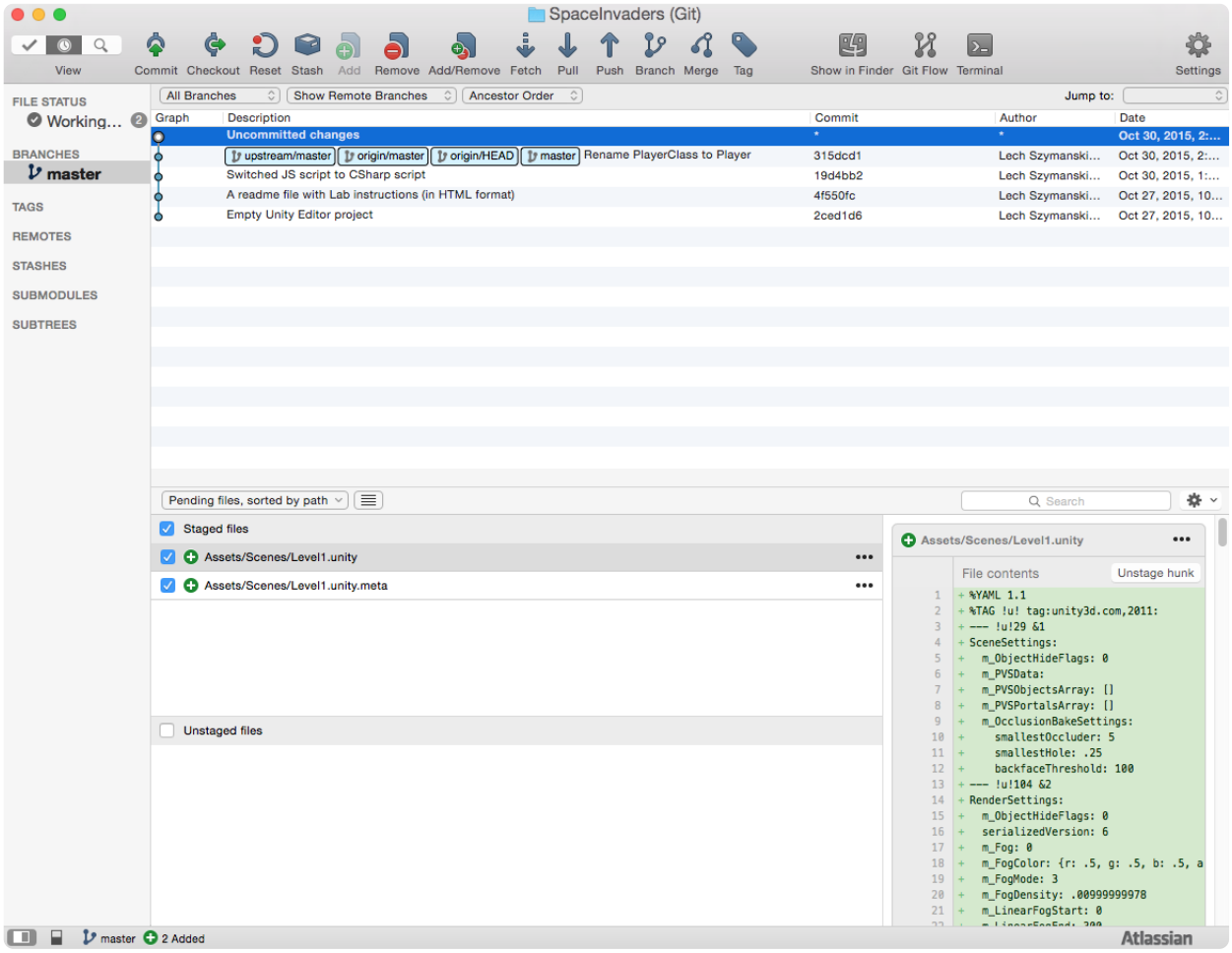
You've done some work on the project and the changes (the new scene) has been saved to your hard-drive. It's a good time now to commit these changes to GIT. Remember, the entire project is a GIT repository, which has the capability to save snapshots. There are many good reasons to keep saving snapshots of your projects as you keep developing it - for one, you can always come back to a previous snapshot in case something goes wrong.

5. Switch to SourceTree, and take a look at the "SpaceInvaders" project. At the top of the history timeline there should be a new entry: "Uncommitted changes". GIT recognises that you've done something to the project. Select the "Uncommitted changes" entry in the timeline.

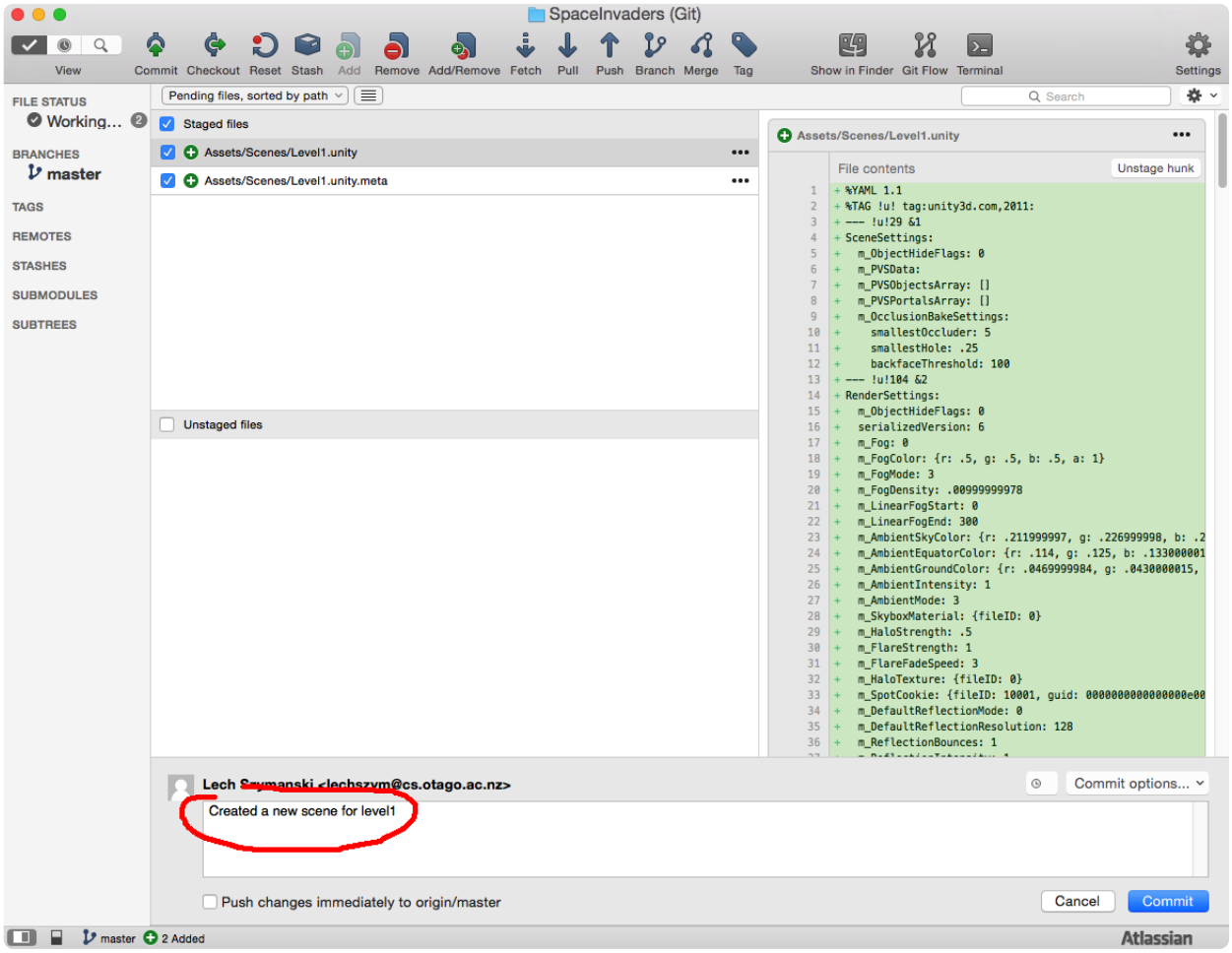
7. If you look below, you'll see a pane titled "Unstaged files". It shows the changes you've made to the project since the last snapshot. If you followed the lab instructions closely, you actually haven't changed anything except for creating a new file, "Level1.unity", which contains the information about the newly created scene. The other file that you see there is an associated "meta" file - Unity creates those, and they need to be saved in the repository as well.



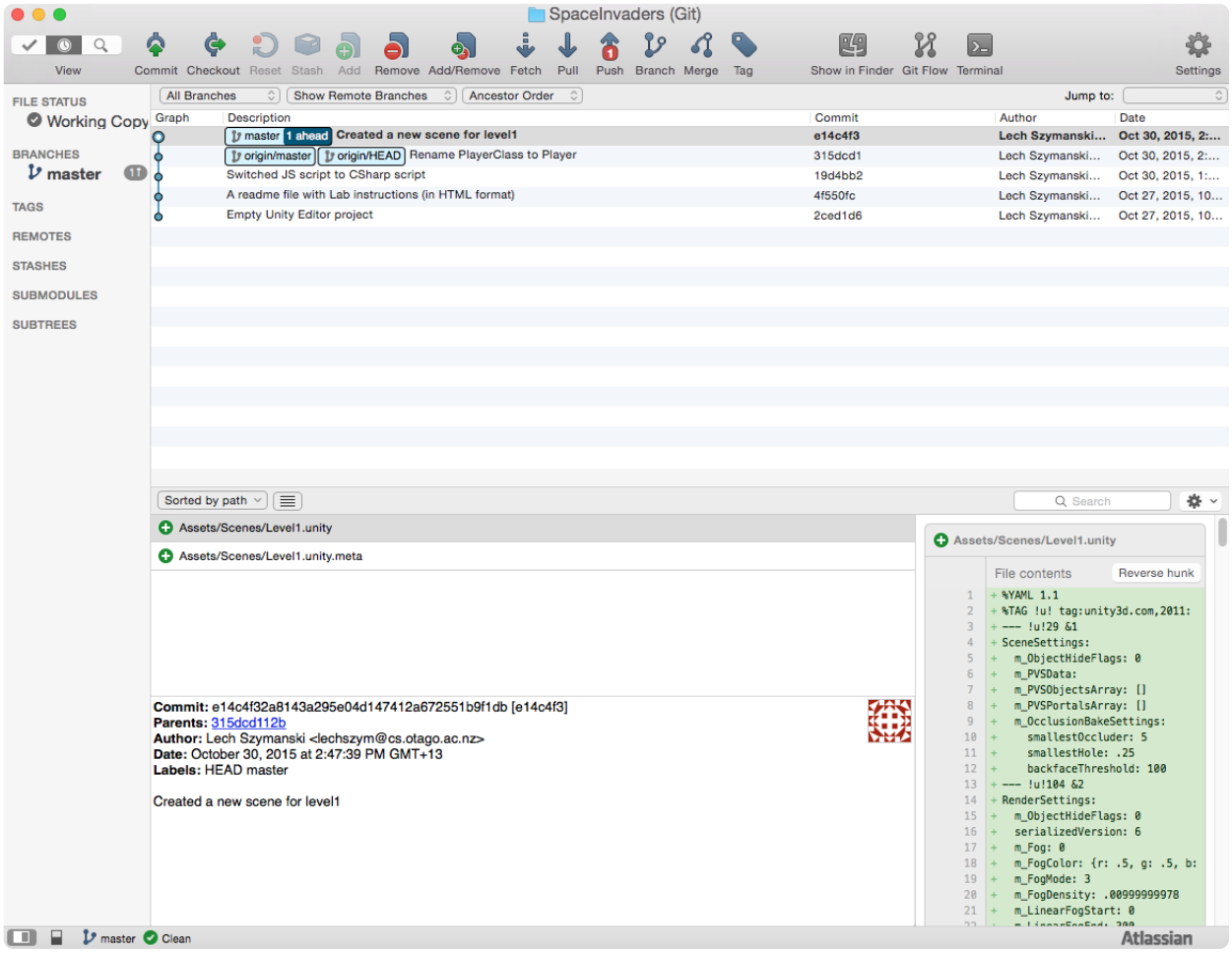
3. From the "Unstaged files" you can select the files that you want to included in the new snapshot. In this case select both. If you want to select everything, just click the box next to "Unstaged files". They selected files will appear in the "Staged files" window.



3. Now, from the toolbar, press "Commit". You will be presented with a new view, with a window where you can put some comments. Git forces you to put comments, so you can't leave that field blank. Try to make a meaningful, but short comment about this snapshot. Take a look at my example in the screenshot below.



). Press the "Commit" button. GIT will make a snapshot including the new staged files. You will see the commit with your comments in the repository timeline. From now on , whatever changes you make, including deletion of the previously committed files, can be undone. So if you make a mistake, or accidentally delete a file, you can always restore it from a given snapshot.

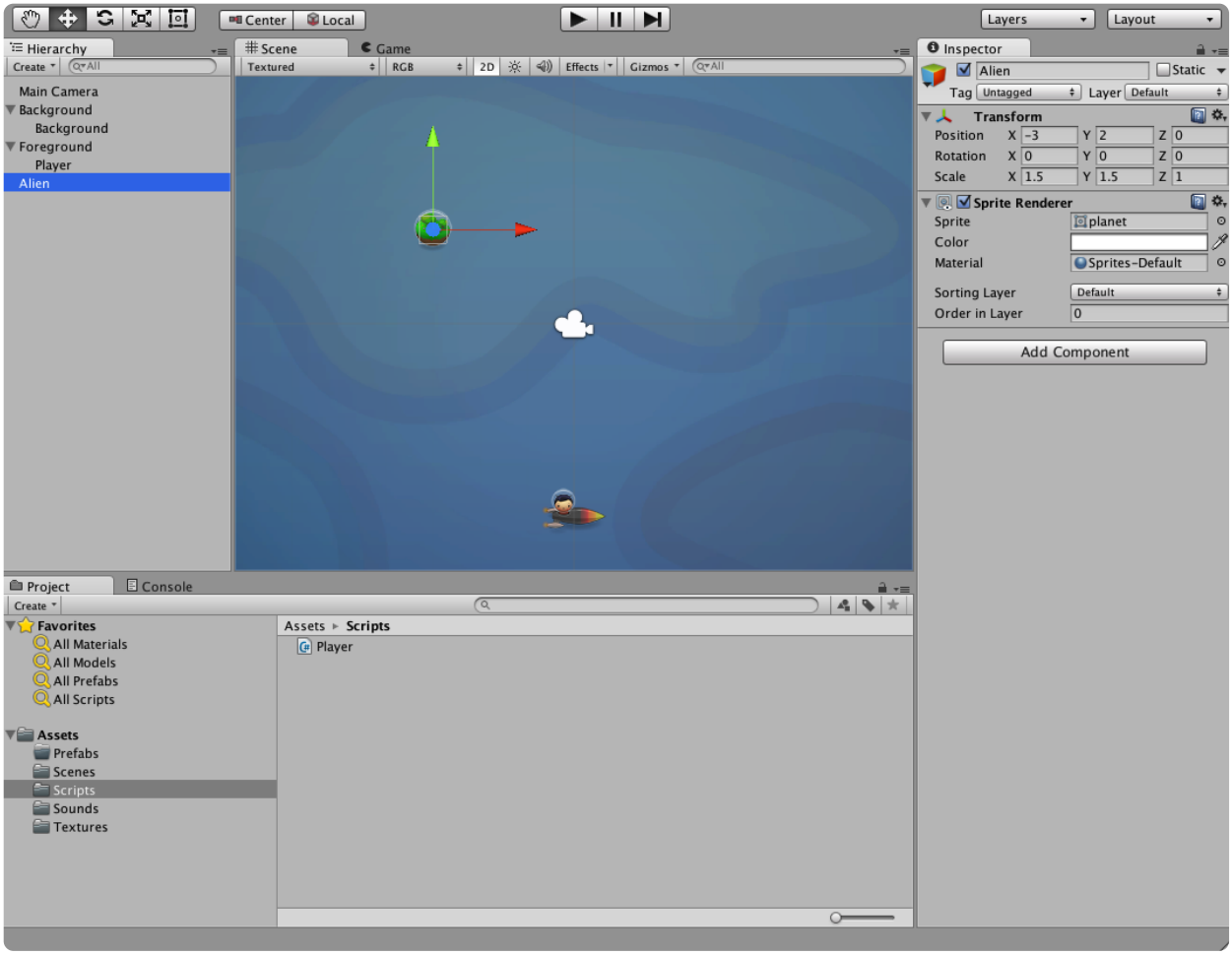


1. Changes have been committed to the repository. Get back to the Unity Editor.

Create an alien wave

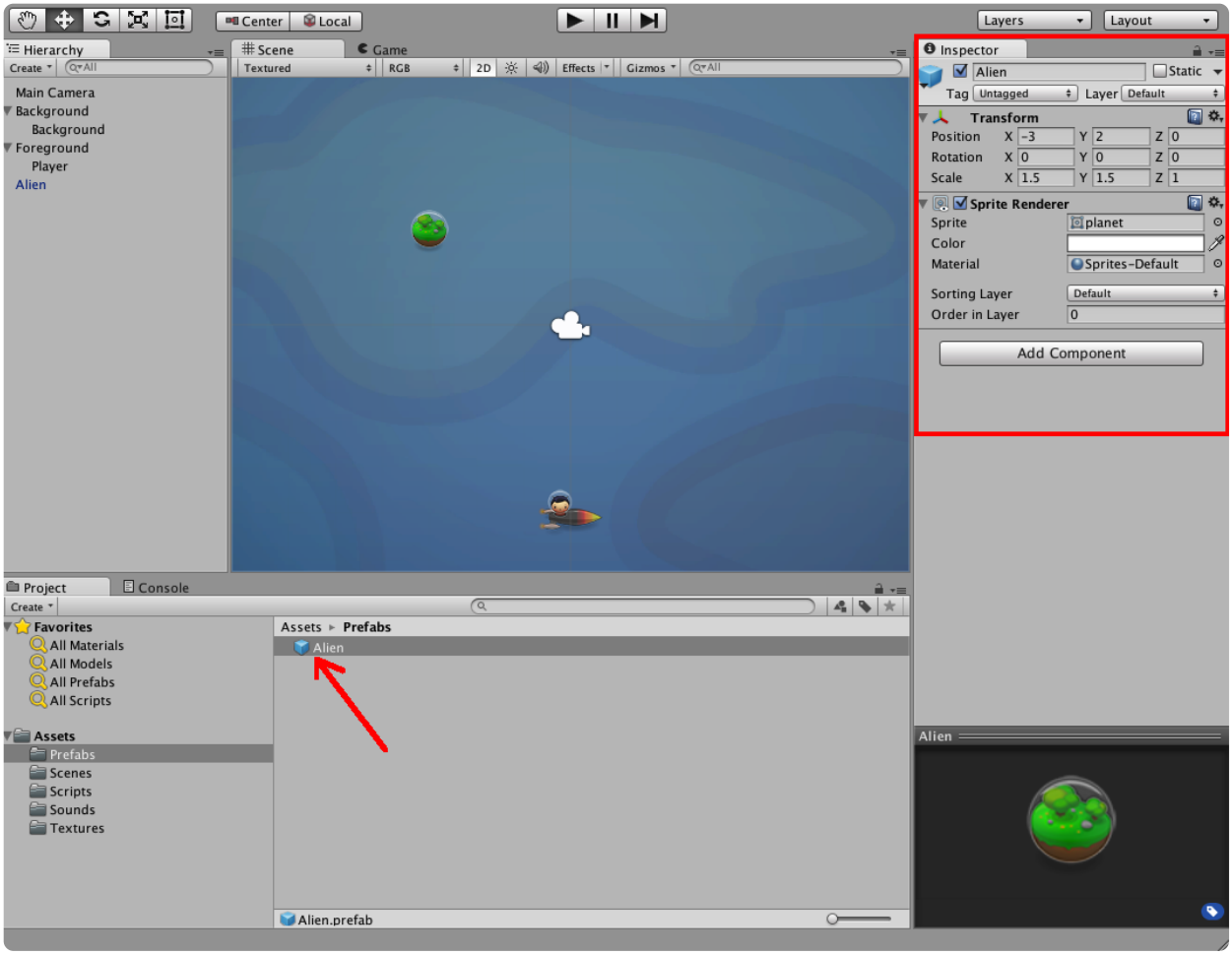
You have a scene with the payer. Time to create some aliens.

2. Create a new *2D Object | Sprite* in the **Hierarchy panel**. Name it *Alien*. Set its *Sprite* property to "planet", *Position* to (-3,2,0), and *Scale* to (1.5,1.5,1).



3. One way to get the *Alien* to move is to write a script for it. Then duplicates could be made and positioned in different parts of the scene. But that's a lot of manual labour. Besides, these aliens will need to move as a wave, so it will be easier to have one script control all of the *Alien*s at once. That script will also spawn a bunch of *Alien* game objects and put them in a formation.

4. In order to spawn a game object from a scrip, a prefab of that object needs to be made. A prefab is a kind of a template for a game object. All it takes to create one is to drag the desired game object from the **Hierarchy panel** to the **Assets panel** - Unity will create an asset which is a prefab for that game object. Go ahead, drag the *Alien* game object from the **Hierarchy panel** to the *Prefabs* subdirectory in **Assets**. You should have your prefab. When you click on the prefab in the **Assets**, you should see its properties (same as for the game object) in the **Inspector panel**.



It is the act of dragging a game object to the **Assets panel** that creates a prefab - it does not matter whether you are dragging it into a *Prefabs* folder or not.

5. Once the prefab is made, you can delete the *Alien* game object from the **Hierarchy panel** (right-click and *Delete*). The prefab in *Assets* should remain unaffected.

6. In *Assets* / *Scripts* create a new script (right-click and select *Create* / *C# Script*). Name the new script "EnemyWave". Double-click it to open in VSC editor and paste in the following code:

EnemyWave.cs

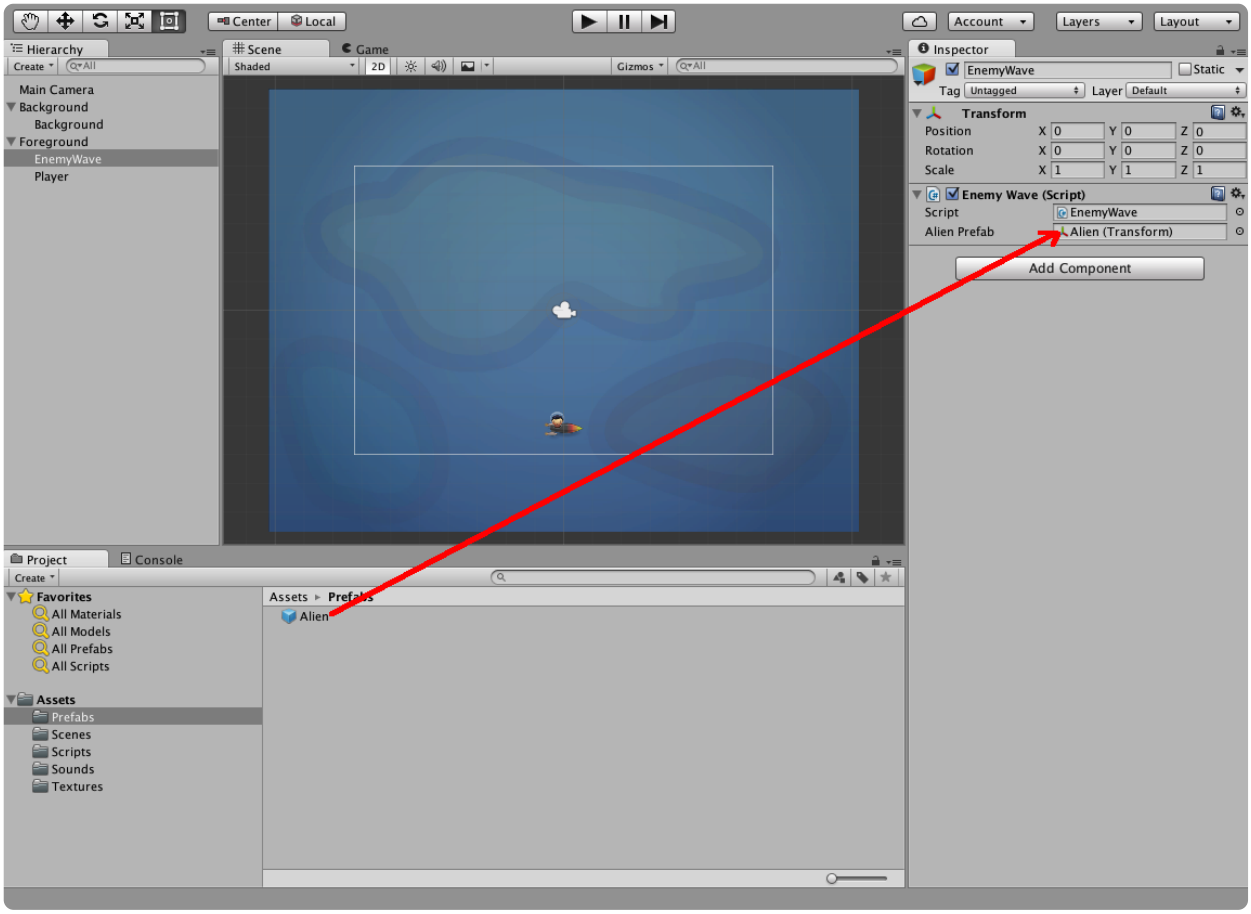
```
01:
02: using UnityEngine;
03:
04: public class EnemyWave : MonoBehaviour {
05:
06:    // Variable poitning to object prefab
07:    public Transform alienPrefab;
08:
09:    // Use this for initialization
10:    void Start () {
11:       float gapBetweenAliens = 1.5f;
12:
13:       for(int y = 0; y < 4; y++) {
14:          // Horizontal offset for every other row
15:          float offsetX = ((y % 2 == 0) ? 0.0f : 0.5f) * gapBetweenAliens;
16:          for(int x = -3; x < 3; ++x) {
17:             // Create new game object (from the prefab)
18:             Transform alien = Instantiate(alienPrefab);
19:             alien.parent = transform;
20:             // Position the newly created object in the wave
21:             alien.localPosition = new Vector3((x*gapBetweenAliens)+ offsetX, 0 + (y
22:          }
23:       }
24:    }
25: }
```

The Start() methods is invoked by the game engine when the game object is instantiated in the scene. Since the wave of aliens is going to be created once, when the scene loads, it makes sense to put the code for the spawning of the aliens inside the Start() method. The two loops inside the function are there to compute the X and Y position of the next alien in the wave - the odd and even rows are offset a bit, assuring there's a gap between the aliens.

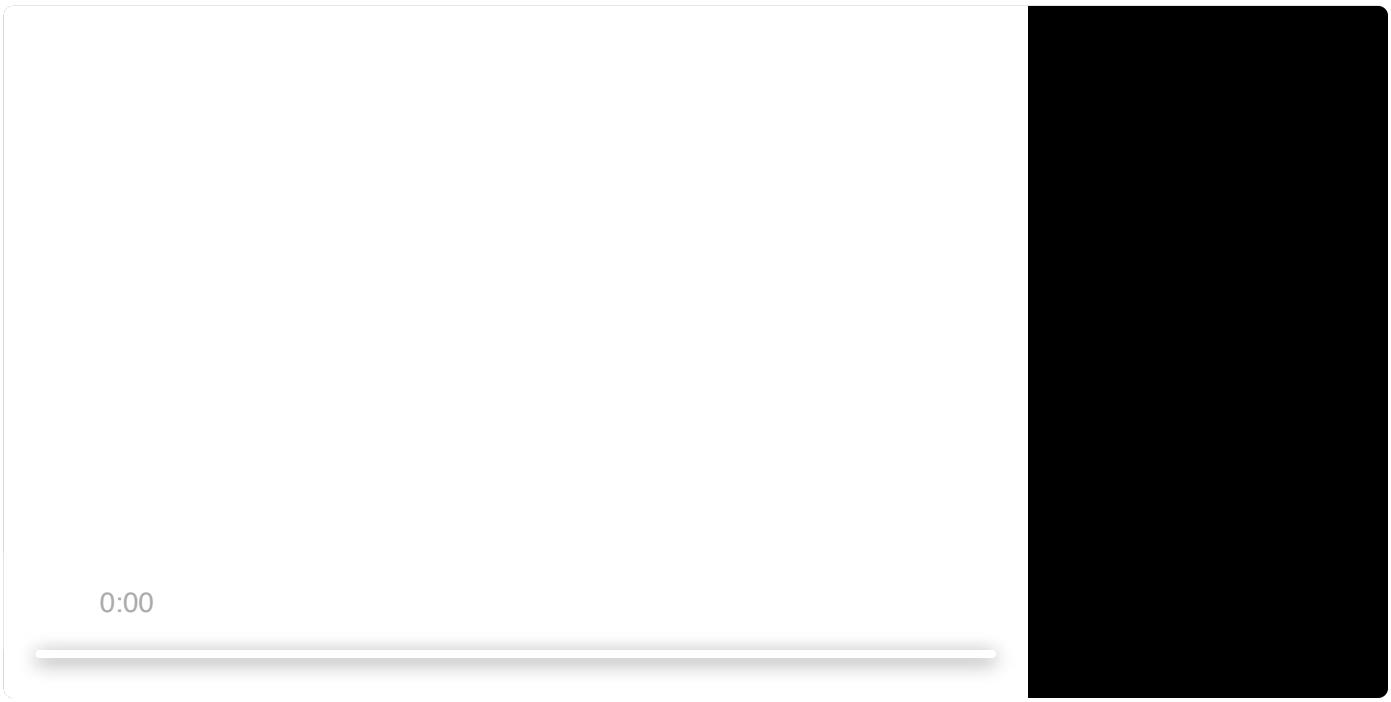
The call to Instantiate() (http://docs.unity3d.com/ScriptReference/Object.Instantiate.html) clones an object whose *Transform* is passed in as the argument. Passing the Transform of a prefab will create an object based on that prefab. The *alienPrefab* variable, which gets passed in to the *Instantiate()* method, is a public variable, and so you'll be able to link it to the *Alien* prefab later on (in the **Inspector panel**). The Instantiate() function returns a reference to the newly created object, which is stored in the *alien* variable. The next line places the newly created object in a hierarchy so that the object with the *EnemyWave* component becomes its parent. Next, the new object is positioned in the scene based on X and Y coordinates computed with the offsets from the two for loops.

7. Back to the Unity editor. Create another empty game object. Name it *EnemyWave* and drag it into the *Foreground* game object.

8. Add "EnemyWave" script as a component of *EnemyWave* game object. Note the *Alien Prefab* attribute that appears along with the new script component. Drag the *Alien* prefab from the *Assets* over the *Alien Prefab* attribute of "EnemyWave" script component. The variable now points to the prefab.

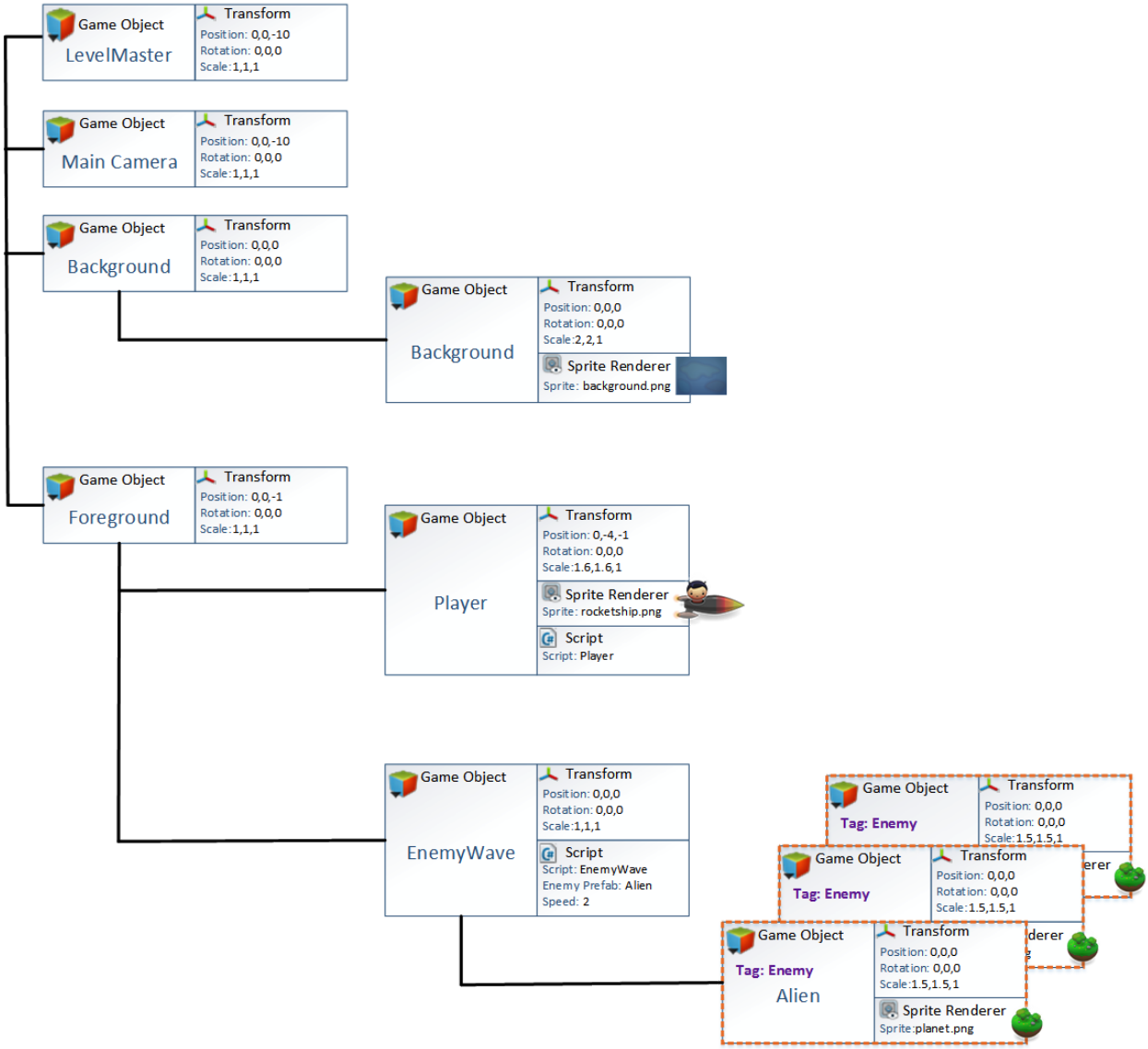


3. Run the game. You should see something like this:

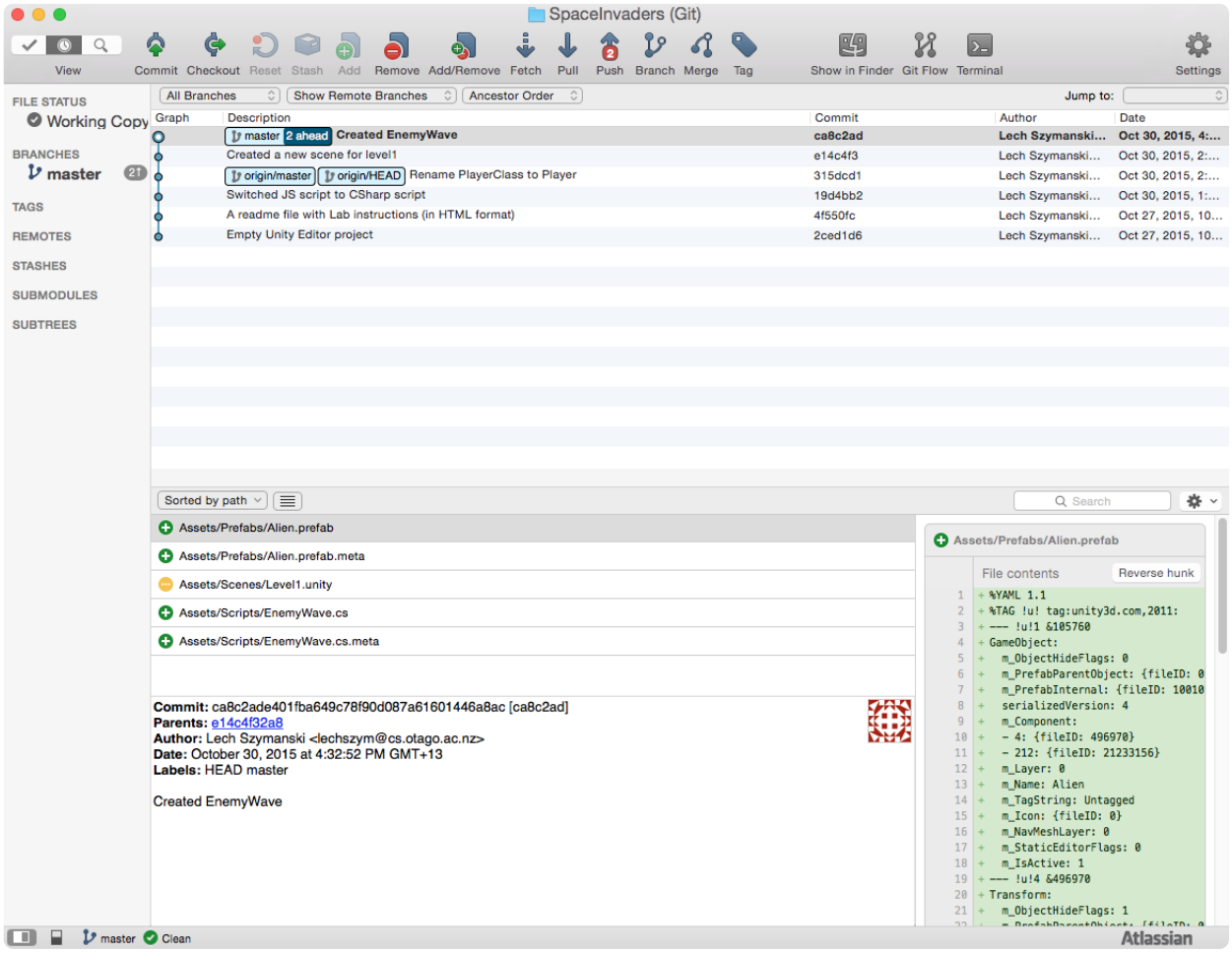


If you start the game with the *EnemyWave* game object selected in the **Hierarchy panel** you'll get to see all the newly created children objects underneath (as shown in the video above).

4. Let's review the game object tree. You have now the *EnemyWave* object under *Foreground*, which is a parent of many *Alien* game objects - the dashed border around *Alien* objects in the diagram below indicates that they are loaded at run time.



1. This is a good time to commit progress to GIT repository. Switch to SourceTree. In the "Unstaged files" you should see two new files for Alien prefab and "Wave.cs" script (plus their corresponding "*.meta" files). If you didn't forget to save the scene, you will also see that "Level1.unity" has been modified. Stage everything and commit with a meaningful comment.



Moving the alien wave

To move the wave you are going to move the *EnemyWave* object. All its children (*Aliens*) will move along.

2. Update the "EnemyWave" script as shown below (the entire script is shown, but the old code is made semitransparent, so the new code, and its location with respect to the old code, should be easy to see):

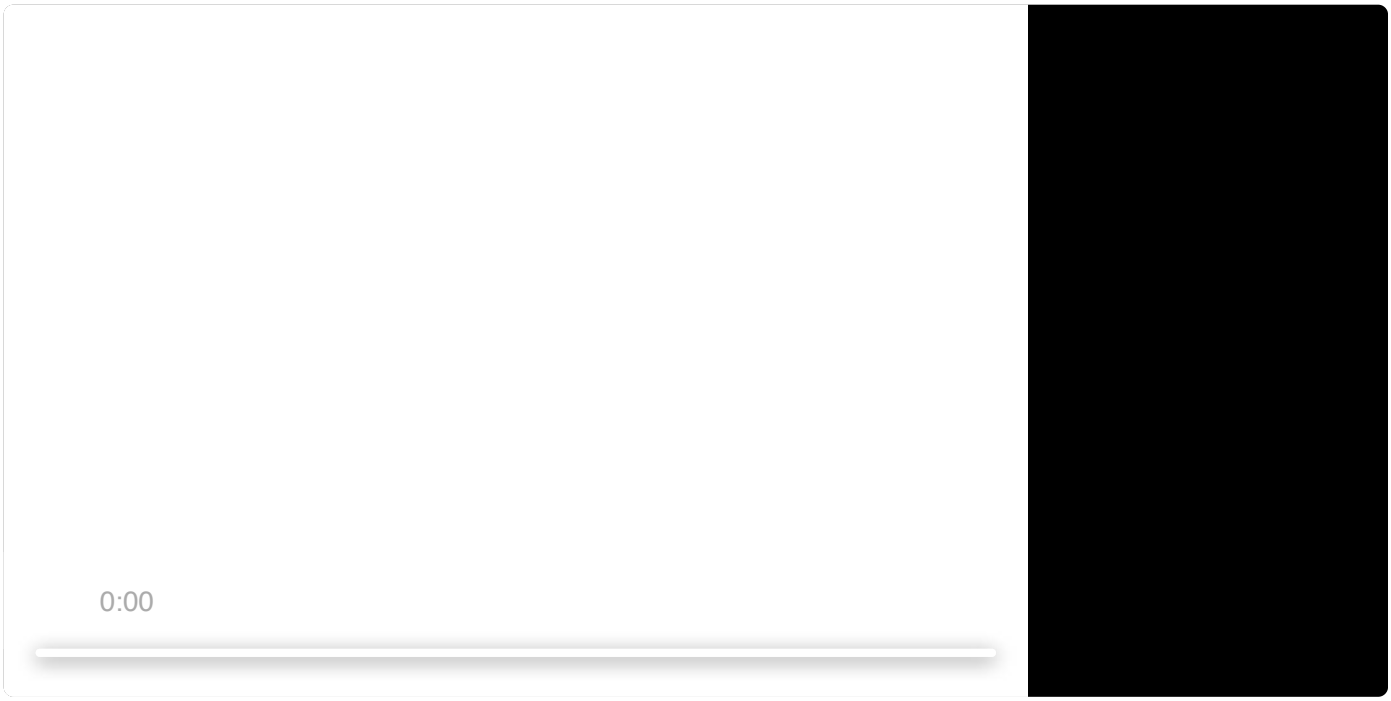
EnemyWave.cs

```
using UnityEngine;

public class EnemyWave : MonoBehaviour {

   // Variable poitning to object prefab
   public Transform alienPrefab;

   // Speed of the wave movement
   public float speed;

   // Direction of the wave movement (-ve means left, +ve is right)
   int direction =-1;

   // Use this for initialization
   void Start () {
      float gapBetweenAliens = 1.5f;

      for(int y = 0; y < 4; y++) {
         // Horizontal offset for every other row
         float offsetX = ((y % 2 == 0) ? 0.0f : 0.5f) * gapBetweenAliens;
         for(int x = -3; x < 3; ++x) {
            // Create new game object (from the prefab)
            Transform alien = Instantiate(alienPrefab);
            alien.parent = transform;
            // Position the newly created object in the wave
            alien.localPosition = new Vector3((x*gapBetweenAliens)+ offsetX, 0 + (y
         }

      }
   }

   // Update is called once per frame
   void Update () {
      // Move the wave on the horisonatal axis
      transform.Translate( new Vector3(Time.deltaTime * direction * speed,0,0));
   }
}
```

Two variables have been added: *speed*, which controls the speed of the wave movement, and a private variable indicating the direction of the movement. At scene start the wave will move to the left. In the *Update()* method the game object (*EnemyWave*) gets translated along the x axis. The computation for the translation is identical to that for movement of the *Player* game object, except the direction is not controlled by user input.

3. Select *EnemyWave* and set its *EnemyWave | Speed* attribute to 2.

4. When you play the game, the wave should move to the left.



5. Save the scene in Unity editor, commit changes to GIT via SourceTree. Probably just two files have been changed - the scene file and the "EnemyWave.cs" script.

Your first collision

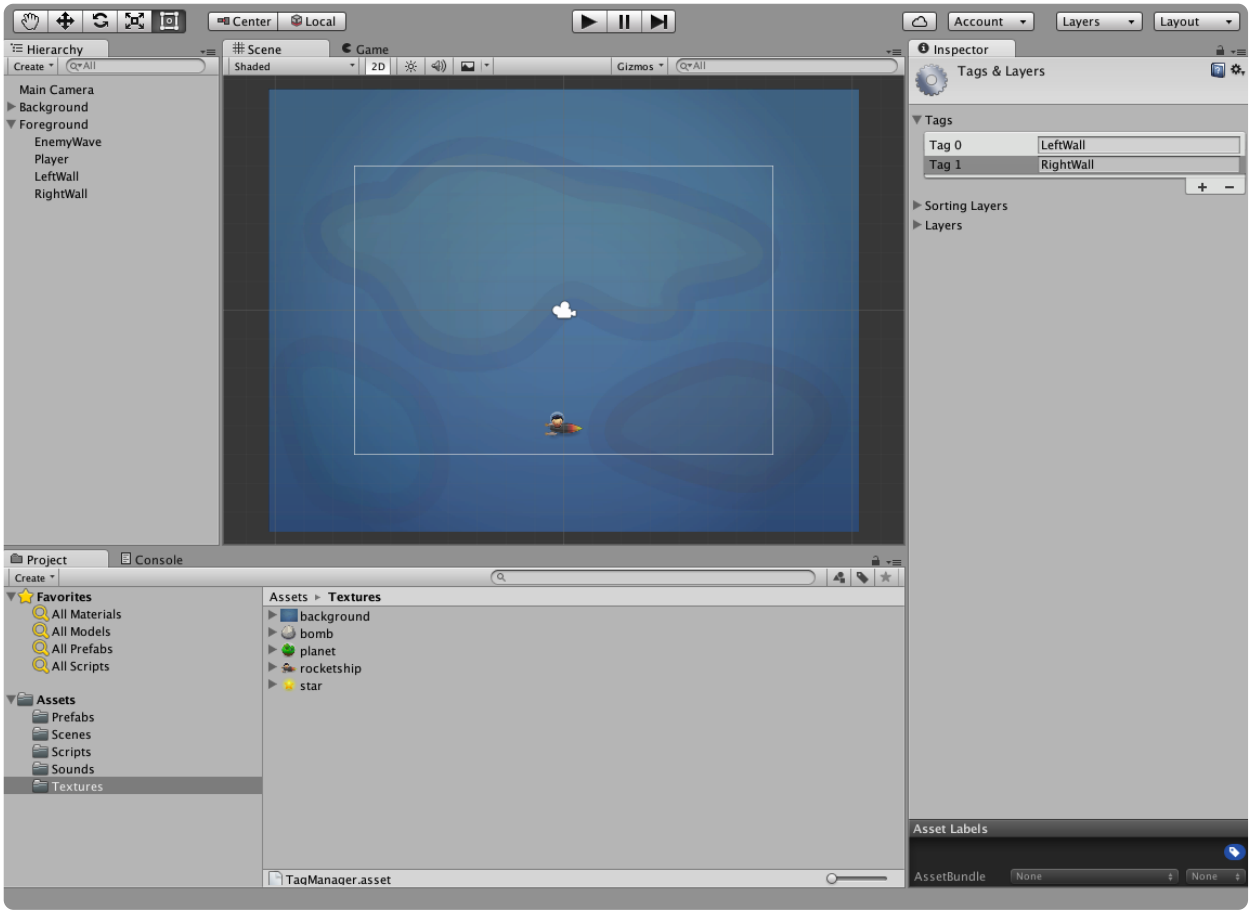
Right now the wave will continue moving in one direction and disappear off the screen, as will the *Player* object (if so directed by the user). To stop those objects from going off screen, you'll need to set up barriers on either side and use the collision system to keep the objects in bounds.

6. Create an empty game object. Name it *LeftWall* and drag it into the *Foreground* game object. Make sure its *Position* is (0,0,0). Add a *Physics 2D | Edge Collider 2D* component.

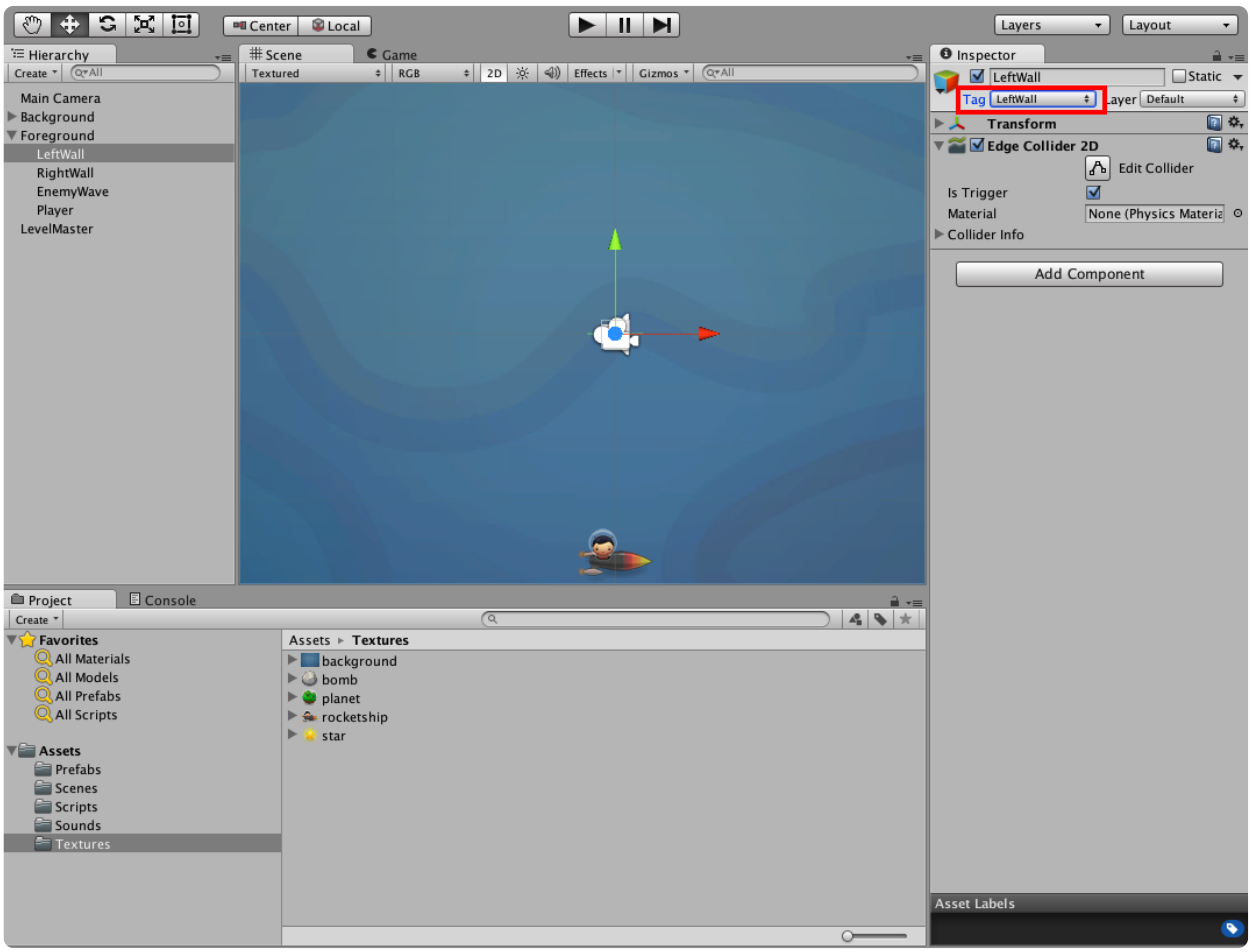
7. In the **Hierarchy panel** right-click the *LeftWall* object and *Duplicate*. Name the new object *RightWall*.

8. Next, you're going to create two tags, which will be used to label the two objects you just created. A tag is a string that can be attached to a game object. Tags are useful for identification (you'll see an example of this shortly). Select the *LeftWall* game object and click on the *Tag* drop-down in the **Inspector panel**. From the set of drop-down options, select the *Add Tag...* option. The **Inspector panel** should now show settings for "Tags & Layers".

9. In the **Inspector panel** expand *Tags* and add two new entries "LeftWall" and "RightWall".



). Select the *LeftWall* game object and click on the *Tag* drop-down in the **Inspector panel** - the newly created tags should be listed as one of the options (the other options are a set of built-in tags). Select "LeftWall".



1. Do the same thing for the *RightWall* game object, except tag it with the "RightWall" label instead. The two walls have been tagged with different string labels. These will be useful when you need to figure out which wall an *Alien* or the *Player* bumped into.

2. The wall objects need to be positioned on the left and right-hand side of the screen. It's possible to do this by hand (manipulating those objects in the Unity editor), but doing it programmatically makes it more precise and adaptable to changes in screen size. Remember, Unity can compile games into different platforms and so taking into account the possibility of slight variations, such as screen size/aspect ratio, is a good idea.

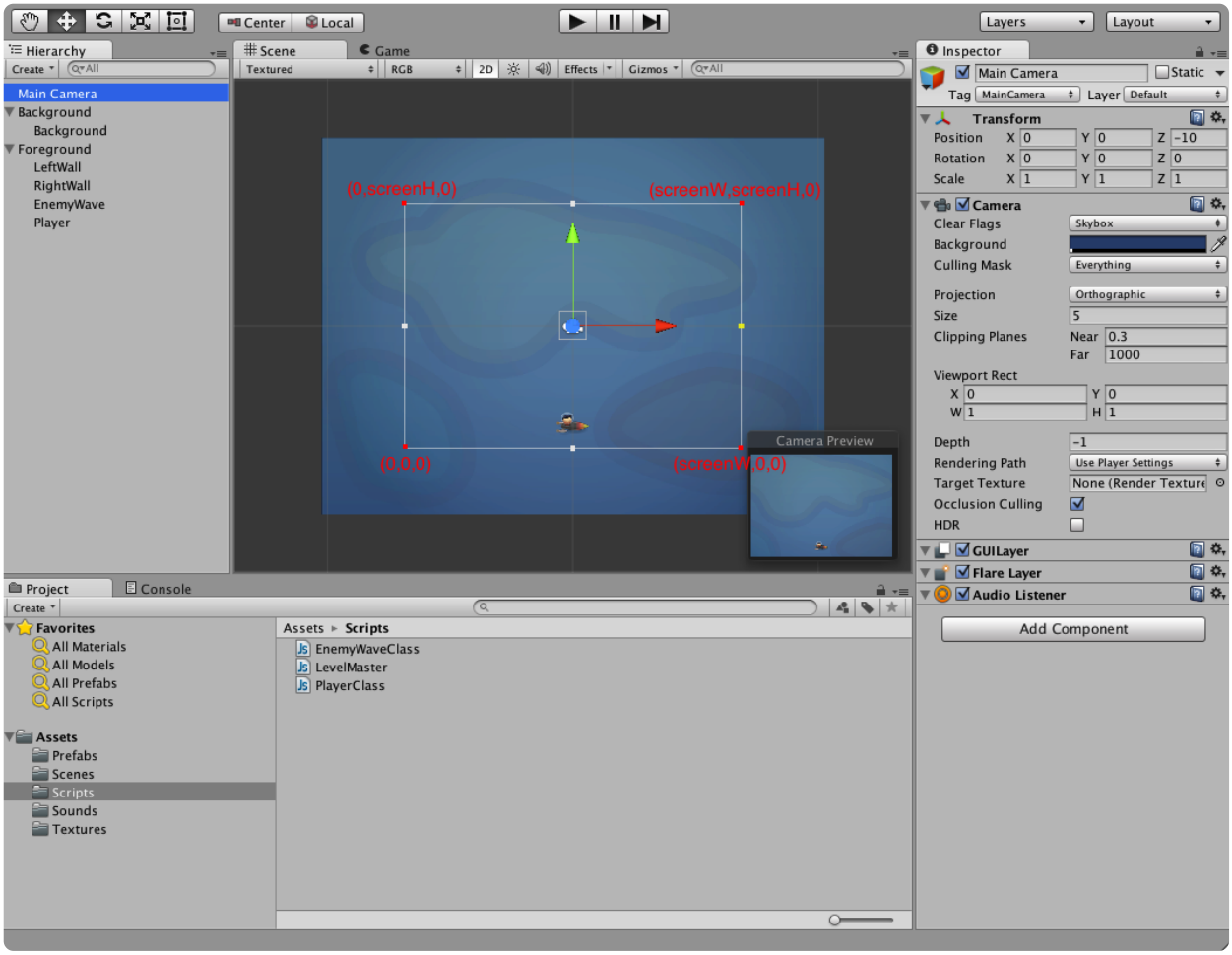
3. Create a new C# script in *Assets | Scripts*. Name it "LevelMaster". Double-click it to open in VSC editor and paste in the following code:

LevelMaster.cs

```
01: using UnityEngine;
02:
03: public class LevelMaster : MonoBehaviour {
04:
05:    // Variables referencing two edge colliders
06:    public EdgeCollider2D leftWall;
07:    public EdgeCollider2D rightWall;
08:
09:    // Use this for initialization
10:    void Start () {
11:       // Get the width and height of the camera (in pixels)
12:       float screenW = Camera.main.pixelWidth;
13:       float screenH = Camera.main.pixelHeight;
14:
15:       // Create an array consisting of two Vector2 object
16:       Vector2[] edgePoints = new Vector2[2];
17:
18:       // Convert screen coordinates point (0,0) to world coordinates
19:       Vector3 leftBottom  = Camera.main.ScreenToWorldPoint(new Vector3(0f, 0f, 0f))
20:       // Convert screen coordinates point (0,H) to world coordinates
21:       Vector3 leftTop = Camera.main.ScreenToWorldPoint(new Vector3(0f, screenH, 0f)
22:
23:       // Set the two points in the array to screen left bottom
24:       // and screen left top points
25:       edgePoints[0] = leftBottom;
26:       edgePoints[1] = leftTop;
27:
28:       // Position the left wall edge collider
29:       // at the left edge of the camera
30:       leftWall.points = edgePoints;
31:
32:       // Convert screen coordinates point (W,0) to world coordinates
33:       Vector3 rightBottom = Camera.main.ScreenToWorldPoint(new Vector3(screenW, 0f,
34:       // Convert screen coordinates point (W,H) to world coordinates
35:       Vector3 rightTop = Camera.main.ScreenToWorldPoint(new Vector3(screenW, screenH
36:
37:       // Set the two points in the array to screen right bottom
38:       // and screen right top points
39:       edgePoints[0] = rightBottom;
40:       edgePoints[1] = rightTop;
41:
42:       // Position the left wall edge collider
43:       // at the left edge of the camera
44:       rightWall.points = edgePoints;
45:    }
46: }
```

This script positions the edge colliders, not the game objects that the colliders are attached to - for our purposes that's sufficient. The *leftWall* and *rightWall* variables refer to those edge colliders (you'll connect the right objects to those references in the Unity Editor in a bit). The position of an edge collider is configured by specifying the points that it passes through. Since for this game, these colliders are straight

lines, the location of each is defined by only two points. The problem is that information about the screen size comes in screen coordinates (in pixels), while those points for the edge collider must be defined in world coordinates (in world "units").



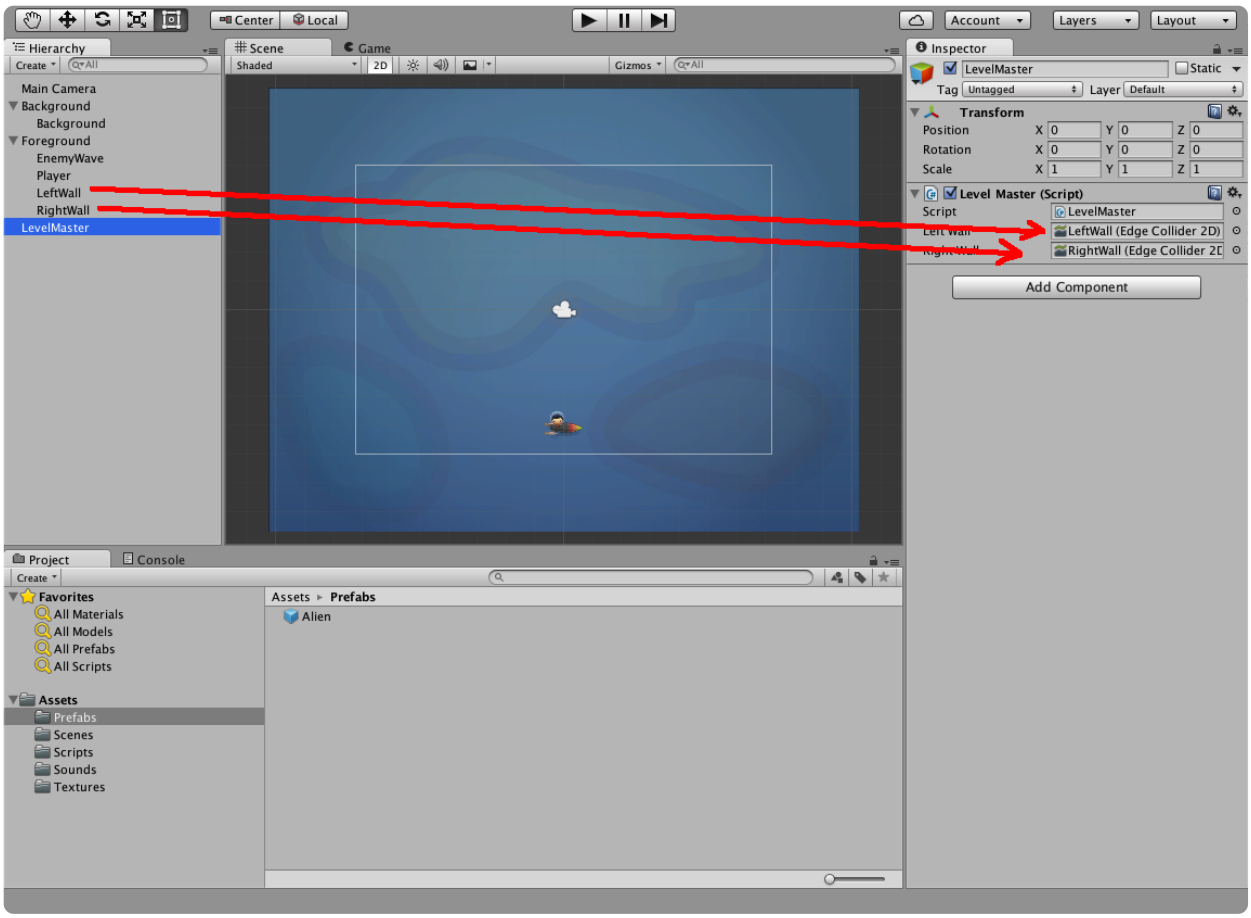
The above screenshot shows the four corners of the screen view in screen coordinates. The value for "screenW" and "screenH" can be found from the *Camera.main* object (which reffers to the main camera that is showing the game view), the Z coordinate is going to be set to zero.

The conversion from screen coordinates to world coordinates is done with the call to the *ScreenToWorldPoint()* (http://docs.unity3d.com/ScriptReference/Camera.ScreenToWorldPoint.html) method of the Camera.main (http://docs.unity3d.com/ScriptReference/Camera.html) object. The camera keeps track of its own location in the world, so it is able to provide a function that can translate screen coordinates to world coordinates. In the script above, the world coordinates for the four corners of the screen are stored in the *leftBottom*, *leftTop*, *rightBottom* and *rightTop* variables.

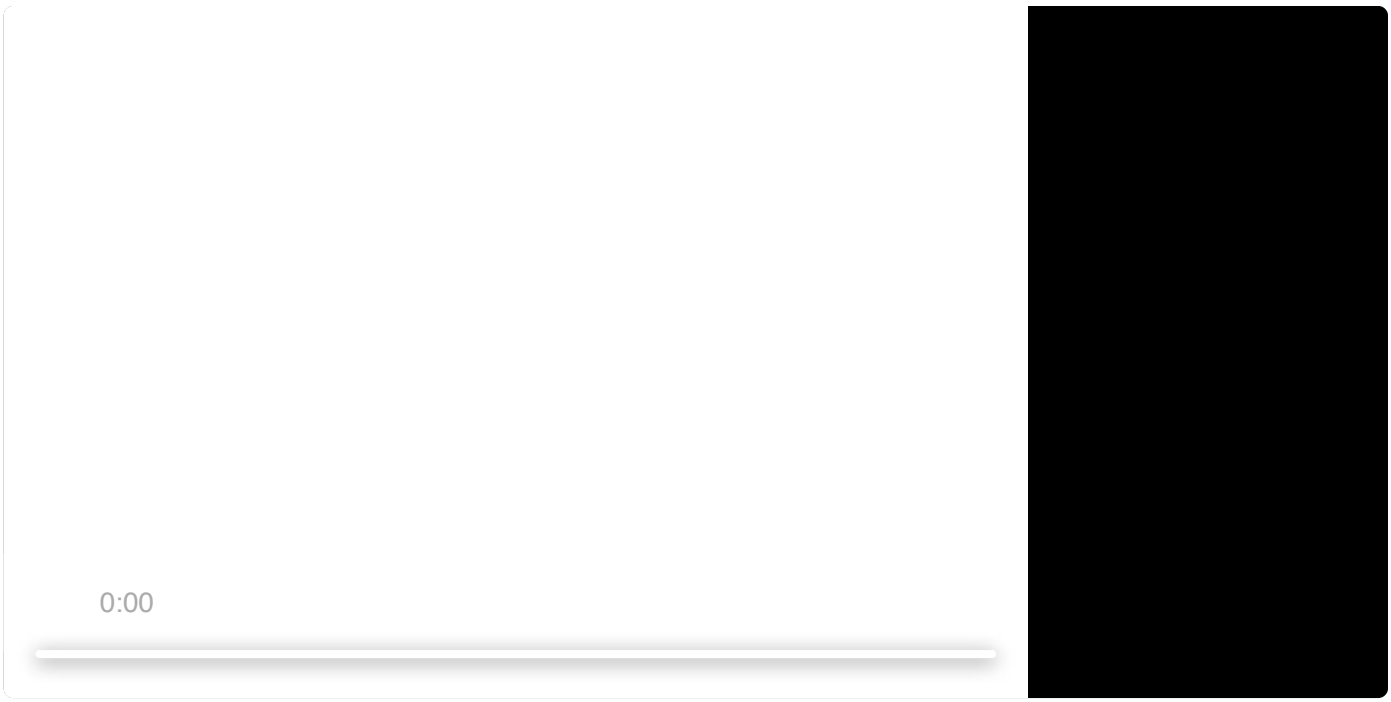
The *edgePoints* variable is an array of 2D vectors - it holds the points that define the position of the edge collider. Position of a 2D edge collider is specified without a Z-axis co-ordinate. For the *leftWall* the array points are set to *leftBottom* and *leftTop* vectors. Did you notice the assignment of a 3D vector to a 2D vector? It would seem that Unity has no problem with that (just drops the Z coordinate). Next, the *leftWall* collider is positioned by setting its *position* property equal to *edgePoints* array. The same thing is done for the *rightWall* collider, except the two points in the *edgePoints* array are set to *rightBottom* and *rightTop* points respectively.

1. Back to the Unity Editor. Create an empty game object. Name it *LevelMaster*, and add the "LevelMaster" script as its script component. Drag the *LeftWall* and *RightWall* game objects over the *Left Wall* and *Right*

Wall attributes of the new component. The two script variables now point to edge 2D colliders.



5. Video below shows the location of the two colliders when the game is played. To see it for yourself, enter play mode, and switch to *Scene view*. Command-click the *LeftWall* and *RightWall* objects in the **Hierarchy panel** to see the outline of their colliders in the view. You might notice that the colliders are not positioned at the edges of the background sprite - that's because the camera view does not encompass the entire background. The walls should outline exactly the left and right edges of the camera.



5. The edge colliders are in position. Time to modify the *Alien* prefab, so that the spawned *Aliens* collide with those edges.

7. Select the *Alien* prefab from *Assets | Prefabs*. Add a *Physics 2D |Rigidbody 2D* component. The Rigidbody (http://docs.unity3d.com/ScriptReference/Rigidbody.html) component makes the game object subject to the rules of the physics engine. Try to play the game now. What happens to the aliens? They drop under gravity. That's because by default rigid bodies come with a mass and are subject to gravity. To stop the aliens from falling down set the *Gravity Scale* attribute of the *Rigidbody 2D* component to 0.

8. Add a *Physic 2D | Circle Collider 2D* component to the *Alien* prefab. The collider is actually a circle that is superimposed over the game object. You can't see it now, because the prefab is not shown in the scene. The easiest thing to do at this point is to drag the *Alien* prefab to the *Scene view*, thus creating an *Alien* game object in the scene. Do you see a green circle around the sprite? That's the collider. It's a bit too big, so you need to shrink it to match closely the outline of the sprite. In the **Inspector panel** under the *Circle Collider 2D* component find *Radius* property and set it to 0.28.

9. Changing the properties of a game object in the scene does not automatically change the prefab from which the object was created. To update the prefab click on the *Apply* button near the top of the **Inspector panel** when the *Alien* game object is selected. You can delete the *Alien* game object from the scene - the prefab will remember all the updates.

10. Hit the play button. What happens? Do individual *Alien*s collide with the wall and then with each other? That means the colliders work, but not in the way they are supposed to in this game. It would be better if an event was triggered on a collision, so that a change in the direction of the entire wave could be made.

11. Select the *LeftWall* game object and enable its *Edge Collider 2D | Is Trigger* attribute. Do the same for the *RightWall* game object.

12. Play the game. Now, the *Alien*s pass through the wall. Trigger colliders do not prevent collisions, but they will send notifications when collisions occur. All you need to do is implement appropriate method(s) in the script component of the colliding objets to catch those triggers.

13. Before continuing with collisions, add two methods to the "EnemyWave" script, which will allow setting of the direction of the wave

EnemyWave.cs

```
using UnityEngine;

public class EnemyWave : MonoBehaviour {

   // Variable poitning to object prefab
   public Transform alienPrefab;

   // Speed of the wave movement
   public float speed;

   // Direction of the wave movement (-ve means left, +ve is right)
   int direction =-1;

   // Use this for initialization
   void Start () {
      float gapBetweenAliens = 1.5f;

      for(int y = 0; y < 4; y++) {
         // Horizontal offset for every other row
         float offsetX = ((y % 2 == 0) ? 0.0f : 0.5f) * gapBetweenAliens;
         for(int x = -3; x < 3; ++x) {
            // Create new game object (from the prefab)
            Transform alien = Instantiate(alienPrefab);
            alien.parent = transform;
            // Position the newly created object in the wave
            alien.position = new Vector3((x*gapBetweenAliens)+ offsetX, 0 + (y * ga
         }

      }
   }

   // Update is called once per frame
   void Update () {
      // Move the wave on the horisonatal axis
      transform.Translate( new Vector3(Time.deltaTime * direction * speed,0,0));
   }

   // Method for changing wave direction (to be invoked
   // from a collider)
   public void SetDirectionLeft() {
      // Check if the current direction is to the right
      if(direction == 1) {
         // Changing the direction
         // push the wave down a bit as well
         direction = -1;
         transform.Translate(new Vector3(0,-0.5f,0));
      }
   }

   // Method for changing wave direction (to be invoked
   // from a collider)
```

```
52:     public void SetDirectionRight() {
53:        // Check if the current direction is to the left
54:        if(direction == -1) {
55:           // Changing the direction
56:           // push the wave down a bit as well
57:           direction = 1;
58:           transform.Translate(new Vector3(0,-0.5f,0));
59:        }
60:     }
61: }
```

In each of the new methods, a check is made whether the new direction setting is different to the one that the wave is moving in at the moment. If the wave is already moving to the left, and the function is called attempting to change direction to the left, nothing is going to happen. This accounts for the possibility of a number of aliens bumping into the wall all at the same time, in which case, the collider will trigger multiple events. While setting direction multiple times would not be the end of the world, there is an aspect of the direction change that should occur only once - the downward movement of the wave. When direction is changed, the wave translates down a bit - that should occur only once when wave hits the a wall.

4. Back to collisions. When two colliders touch, and one of them is a trigger, the engine invokes a game object's *OnTriggerEnter2D()* (http://docs.unity3d.com/ScriptReference/MonoBehaviour.OnTriggerEnter2D.html) method. You can override this method in a game object's script component, providing custom behaviour.

5. Create a new C# Script in *Assets | Scripts*. Name it "Alien". Double-click to open it in VSC editor, and paste in the following code:

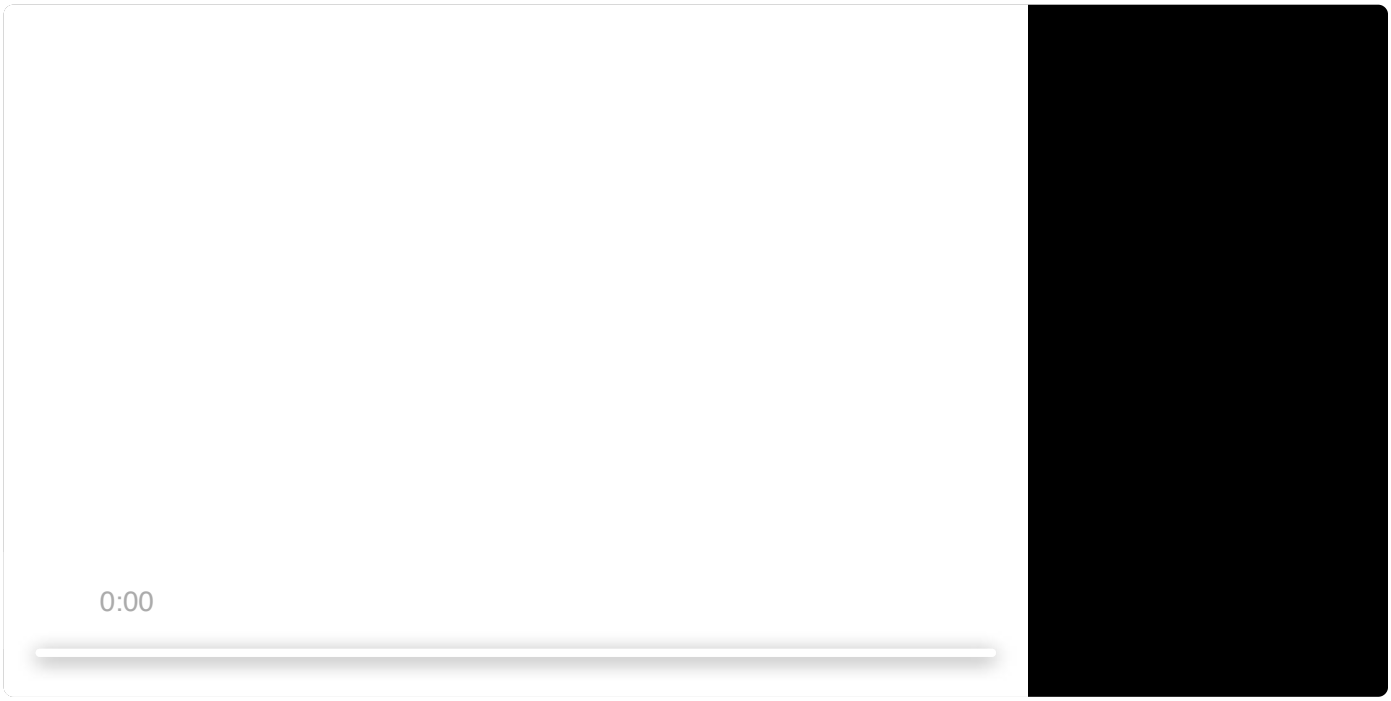
Alien.cs

```
01: using UnityEngine;
02:
03: public class Alien : MonoBehaviour {
04:
05:    // When enemy collides with an object with a
06:    // collider that is a trigger...
07:    void OnTriggerEnter2D(Collider2D other) {
08:       EnemyWave wave;
09:
10:       // Check if colliding with the left or right wall
11:       // (by checking the tags of the collider that the enemy
12:       //  collided with)
13:       if(other.tag == "LeftWall") {
14:          // If collided with the left wall, get a reference
15:          // to the EnemyWave object, which should be a component
16:          // of enemies parent
17:          wave = transform.parent.GetComponent<EnemyWave>();
18:          // Set direction of the wave
19:          wave.SetDirectionRight();
20:       } else if(other.tag == "RightWall") {
21:          // If collided with the right wall, get a reference
22:          // to the EnemyWave object, which should be a component
23:          // of enemies parent
24:          wave = transform.parent.GetComponent<EnemyWave>();
25:          // Set direction of the wave
26:          wave.SetDirectionLeft();
27:       }
28:
29:    }
}
```

The *other* parameter, that is passed to the *OnTriggerEnter2D()* method is a reference to the collider of the other object (involved in the collision). The script relies on tags to determine which wall the *other* reference corresponds to. Remember how you tagged the two walls? If the *other* object is the *LeftWall*'s collider, the wave direction will be set to move to the right. If the *other* object is the *RightWall*'s collider, the wave direction will be set to move to the left. In order to get a reference to the *EnemyWave*'s object, which implements the *SetDirectionRight()* and *SetDirectionLeft()* methods, the script follows the object hierarchy - from *transform* of the current (*Alien*) object to its parent's (*EnemyWave* game object), component of type *EnemyWave*

5. Back in the Unity Editor add the "Alien" script component to the *Alien* prefab.

7. Play the game, the wave should now go back and forth, moving down after each collision with a wall:



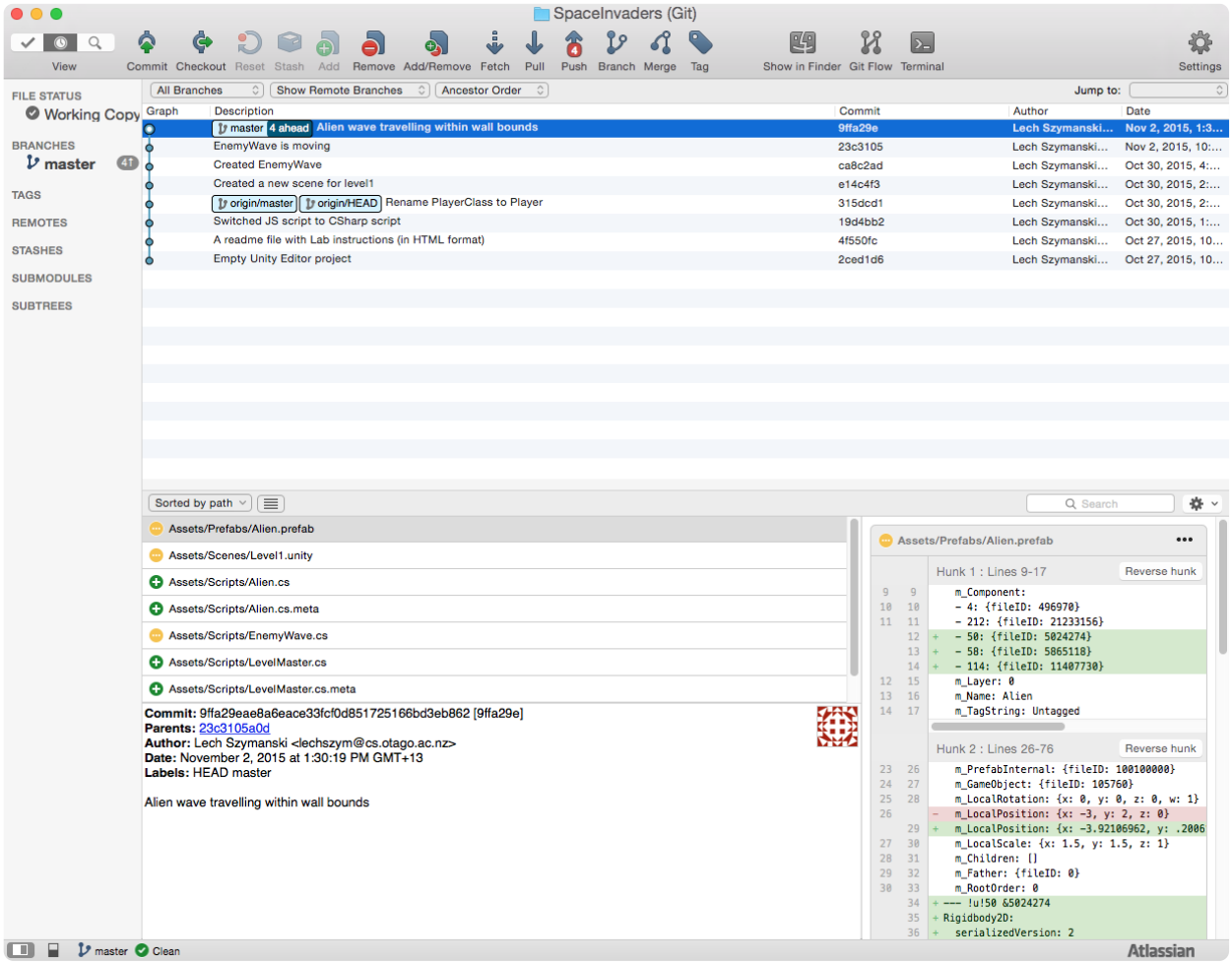
About Colliders

In Unity, in order to get an interaction (a collision) between two game objects, they both need to have a collider component, and at least one of them needs to be a rigid body. A collider is a geometric shape superimposed over the game object. The game engine moves these shapes with the associated game objects and regularly performs computations to check if any of the shapes overlap. Depending on the type of the game object, or the look of the sprite, different shapes will work best; hence Unity provides a choice of circle (http://docs.unity3d.com/Manual/class-CircleCollider2D.html), box (http://docs.unity3d.com/Manual/class-BoxCollider2D.html) (which is a rectangle), or edge (http://docs.unity3d.com/Manual/class-EdgeCollider2D.html) colliders.

2D colliders (http://docs.unity3d.com/ScriptReference/Collider2D.html) operate in the XY plane - the Z-axis of the game object is not taken into account when detecting collisions. That means that in 2D physics things may collide even if they're at different depth from camera. It is still possible to assign objects to different layers and disable collisions between given layers.

Depending on the methods implemented in the scripts of the colliding objects, the engine can send messages when the objects Enter, Exit or Stay in collision state. When one of the colliders is a trigger, messages are sent to the *OnTrigger* methods, otherwise messages are sent to the *OnCollision* methods.

3. Save the scene in the Unity Editor, commit changes to GIT via SourceTree. There should be a number of new files and changes. Stage everything and commit.



Keep the player within the bounds

Let's set up collisions between the walls and the *Player* game object, so that the player can't venture off the screen.

3. Add the *Physics 2D | Rigidbody 2D* component to the *Player* game object. Set its *Gravity Scale* attribute to 0. The *Player* game object is now part of the physics system that checks for collisions.

3. Add the *Physics 2D | Box Collider 2D* component to the *Player* game object. Adjust its *Size* attribute to fit the sprite (settings of X=0.85 and Y=0.6 work well enough). The collision region for the *Player* object is now defined.

4. Now, to detect *Player* collisions, and prevent the object passing through the walls, add the following code to the "Player" script:

Player.cs

```
using UnityEngine;

public class Player : MonoBehaviour {

    // Private variables (not visible in the Inspector panel)
    // The speed of player movement
    float speed = 10;

    // Flag indicating whether the player is at the
    // left edge of the screen
    bool atLeftWall = false;

    // Flag indicating whether the player is at the
    // right edge of the screen
    bool atRightWall = false;

    // On collision with a trigger collider...
    void OnTriggerEnter2D(Collider2D other) {
        // Check the tag of the object the player
        // has collided with
        if(other.tag == "LeftWall") {
            // If collided with the left wall, set
            // the left wall flag to true
            atLeftWall = true;
        } else if(other.tag == "RightWall") {
            // If collided with the right wall, set
            // the right wall flag to true
            atRightWall = true;
        }
    }

    // When no longer colliding with an object...
    void OnTriggerExit2D(Collider2D other) {
        // Check the tag of the object the player
        // has ceased to collide with
        if(other.tag == "LeftWall") {
            // If collided with the left wall, set
            // the left wall flag to true
            atLeftWall = false;
        } else if(other.tag == "RightWall") {
            // If collided with the right wall, set
            // the right wall flag to true
            atRightWall = false;
        }
    }

    // Update is called once per frame
    void Update () {
        // Player movement from input (it's a variable between -1 and 1) for
        // degree of left or right movement
        float movementInput = Input.GetAxis("Horizontal");
```

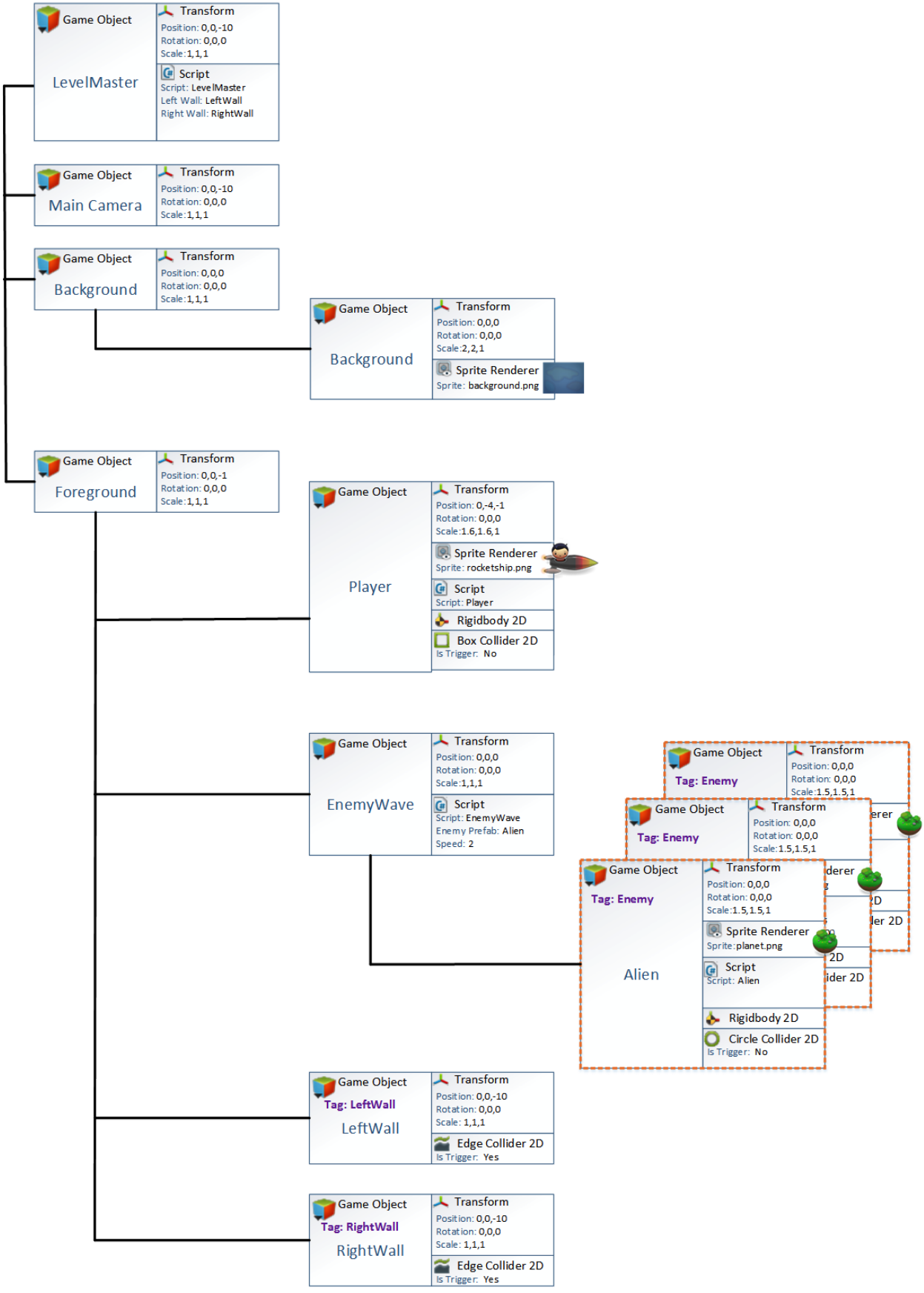
```
52:
53:         // If close to wall and moving towards it,
54:         // stop the movment
55:         if(atLeftWall && (movementInput < 0) )  {
56:            movementInput = 0;
57:         }
58:         if(atRightWall && (movementInput > 0) )  {
59:            movementInput = 0;
60:         }
61:
62:         // Move the player object
63:         transform.Translate( new Vector3(Time.deltaTime * speed * movementInput,0,0))
64:      }
65:   }
```

The *atLeftWall* and *atRightWall* flags indicate whether *Player* is touching one of the walls. They're set in the *OnTriggerEnter2D()* and cleared in the *OnTriggerExit2D()* method. That means the flag is true while the player stays in collision with the wall.

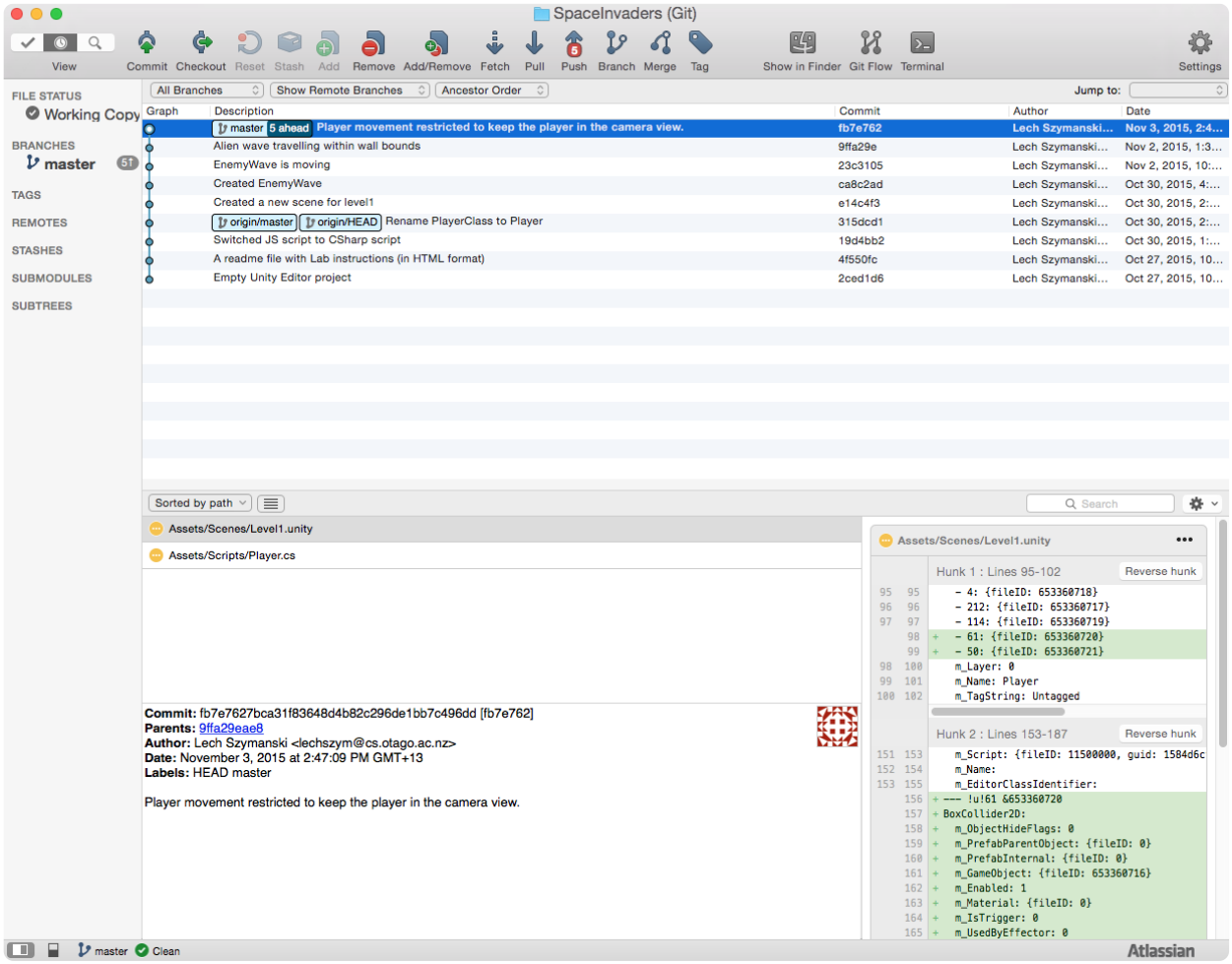
In the *Update()* conditions have been added so that when *atLeftWall* is true, movement to the left is not allowed, and when *atRigthWall* is true, movement to the right is not allowed.

2. Play the game. The *Player* sprite should be confined to the region between the walls.

3. Here's a diagram of the game object hierarchy of the current scene:



1. In SourceTree, stage all the changes and commit.



Adding projectiles

5. Create a new *2D Object | Sprite*. Name it *AlienShot*. Set its *Sprite* property to the "bomb" image and *Scale* to (1.2,1.2,1). Position doesn't matter since you're going to make this game object into a prefab for the alien projectile.

6. Create a new *2D Object | Sprite*. Name it *PlayerShot*. Set its *Sprite* property to the "star" image and *Scale* to (1.5,1.5,1). This game object will be made into the player's projectile.

7. The same script will control both the alien and the player projectiles. In *Assets | Scripts* create a new C# Script and name it "Projectile". Double-click to open it in VSC editor, and paste in the following code

Projectile.cs

```
01: using UnityEngine;
02:
03: public class Projectile : MonoBehaviour {
04:
05:    // The speed fo the projectile
06:    public float speed;
07:
08:    // Flag identifyng whether the projectile
09:    // is sent by enemy or the player
10:    public bool enemyProjectile;
11:
12:    // Update is called once per frame
13:    void Update () {
14:       // The projectile travels up (in the direction of positive y axis), but
15:       // the movement is multiplies by speed (so negative speed will get
16:       // move the projectile down)
17:       transform.Translate(Vector3.up * speed * Time.deltaTime);
18:    }
19: }
```

The script moves the game object during *Update()*. This time, instead of creating a new 3D vector to pass to the *Translate()* method, *Vector3.up* (http://docs.unity3d.com/ScriptReference/Vector3-up.html) is used. It's a built-in shorthand for *new Vector3(0,1,0)*. Multiplying the vector times *speed* (and time since the last *Update()*) produces a vertical translation. Sign of the speed variable controls whether the translation is going up or down.

The *enemyProjectile* boolean denotes whether a particular object instance of this script is a component of the alien or player projectile. It will be useful when dealing with collisions.

3. Add the *Projectile* script component to the *AlienShot* game object. Set its *Speed* attribute to -5 and *Enemy Projectile* to true.

9. Add the *Projectile* script component to the *PlayerShot* game object. Set its *Speed* variable to 5 and leave *Enemy Projectile* set to false.

0. Press play, the projectiles should move in opposite directions

1. Both projectile types need to be instantiated from a script: either at random times (for *AlienShot*s) or after pressing *Fire* (for *PlayerShots*s). Drag the *AlienShot* and then *PlayerShot* game objects from the **Hierarchy panel** to *Assets | Prefabs* to create corresponding prefabs. Once their prefabs have been made, you can delete the two game objects from the scene.

2. Create a new C# Script in *Assets | Scripts*. Name it "Attack". Double-click to open it in VSC editor and paste in the following code:

Attack.cs

```
using UnityEngine;

public class Attack : MonoBehaviour {

   // Variable storing projectile object
   // prefab
   public Transform shotPrefab;

   // Probability of auto-shoot (0 if
   // no autoshoot)
   public float autoShootProbability;

   // Cooldown time for firing
   public float fireCooldownTime;

   // How much time is left until able to fire again
   float fireCooldownTimeLeft = 0;

   // Per every frame...
   void Update () {
      // If still some time left until can fire again
      // reduce the time by the time since last
      // frame
      if(fireCooldownTimeLeft > 0) {
         fireCooldownTimeLeft -= Time.deltaTime;
      }

      // If auto-shoot probability is more than zero...
      if(autoShootProbability > 0) {
         // Generate number a random number between 0 and 1
         float randomSample = Random.Range(0f, 1f);
         // If that random number is less than the
         // probability of shooting, then try to shoot
         if(randomSample < autoShootProbability) {
            Shoot();
         }
      }
   }

   // Method for firing a projectile
   public void Shoot() {
      // Shoot only if the fire cooldown period
      // has expired
      if(fireCooldownTimeLeft <= 0) {

         // Create a projectile object from
         // the shot prefab
         Transform shot = Instantiate(shotPrefab);
         // Set the position of the projectile object
         // to the position of the firing game object
         shot.position = transform.position;
         // Set time left until next shot
         // to the cooldown time
```

```
52:             // to the cooldown time
53:             fireCooldownTimeLeft = fireCooldownTime;
54:         }
55:     }
56: }
```

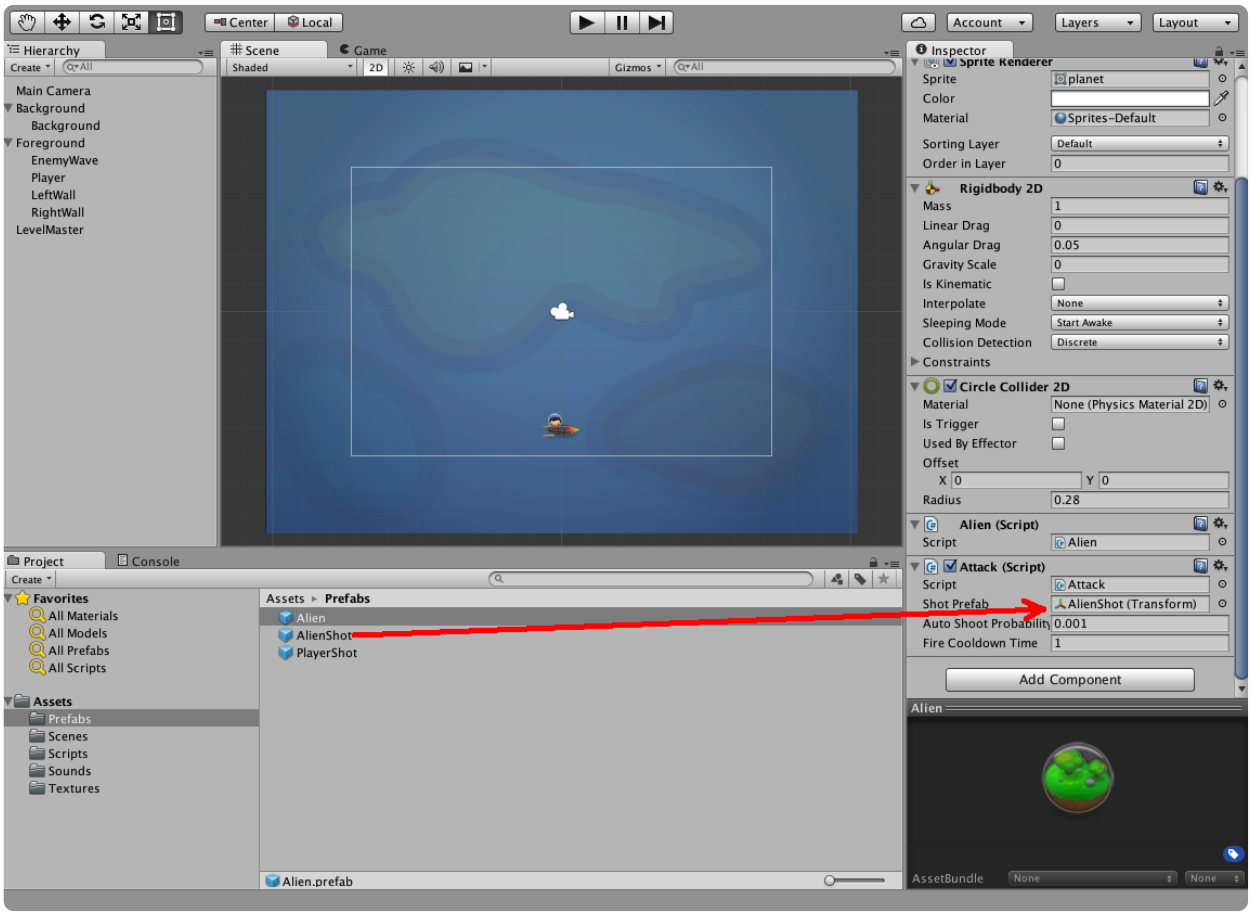
In the script there are three public variables: *shotPrefab* is a reference to the prefab corresponding to the shot game object instantiated by the script, *autoShootProbability* is the probability of firing a shot at random, and *fireCooldownTime* is the time (in seconds) between consecutive shots.

There is also a private variable *fireCooldownTimeLeft* which tracks the amount of cool down time left before the next shot. This counter is set to *fireCooldownTime* after each shot, and decremented in the the *Update()* method with the time since the previous *Update()*.

If *autoShotoProbability* is not 0, a random number sample is drawn from uniform distribution between 0 and 1. If that sample is less than the shoot probability, then the *Shoot()* method is called.

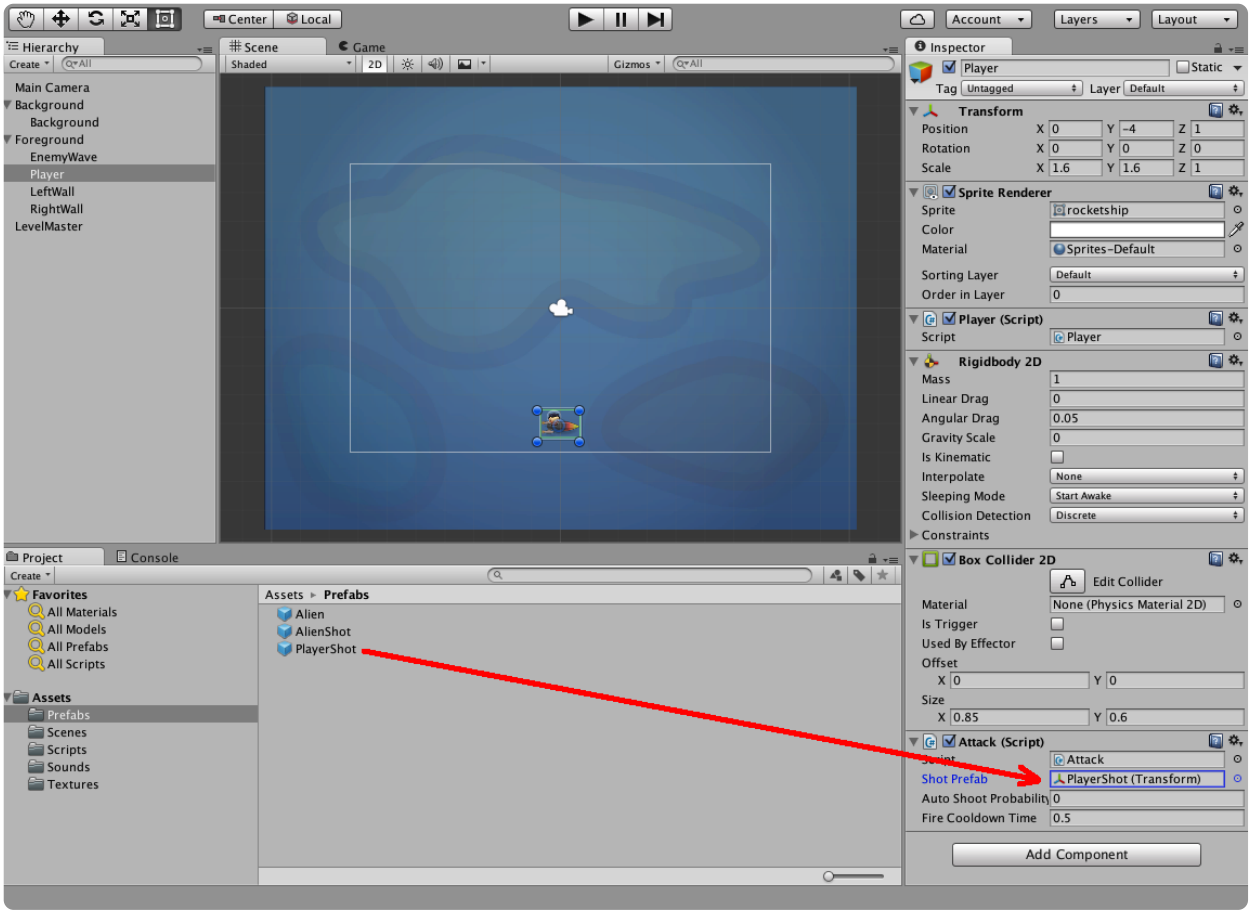
Inside the *Shoot()* method, if the *fireCooldownTimeLeft* has expired, a projectile is spawned from the *shotPrefab* variable. The new game object is positioned where the object doing the shooting happens to be at this time, and the *fireCooldownTimeLeft* is reset to the *fireCooldownTime*.

3. Find the *Alien* prefab in the *Assets | Prefabs*. Add the "Attack" script component. Initialise all the public variables: dragging the *AlienShot* prefab to *Shot Prefab*, set *Auto Shoot Probability* to 0.001 and *Fire Cooldown Time* to 1.



4. Press play. Aliens should be dropping their bombs now.

5. Stop the game. Select the *Player* game object and add the "Attack" script component. Initialise all the variables again, but this time: drag the *PlayerShot* prefab to *Shot Prefab*, set *Auto Shoot Probability* to 0 and *Fire Cooldown Time* to 0.5.



6. The probability of shooting automatically for the *Player* is set to 0, because player shots are to be triggered by user input. To enable shooting when the *Space*-key is pressed, add the following code to the *Player* script:

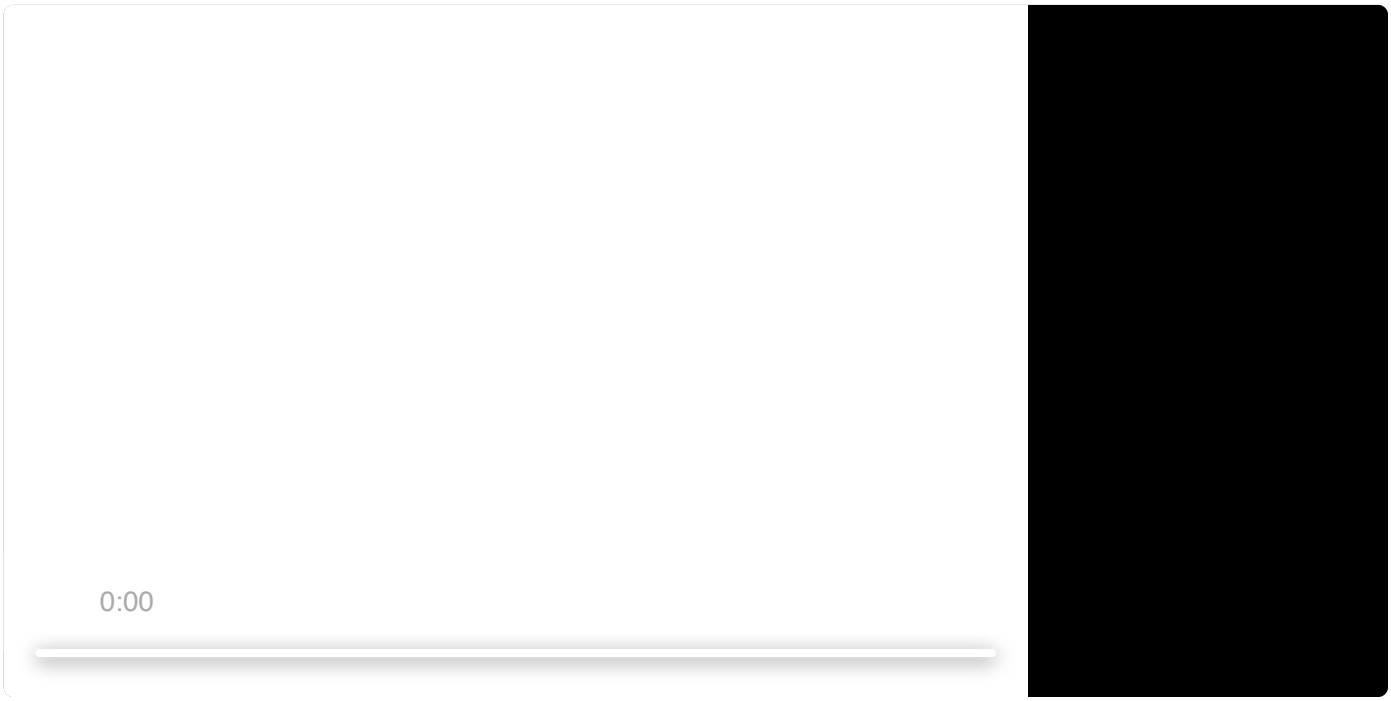
Player.cs

```
using UnityEngine;

public class Player : MonoBehaviour {

   // Private variables (not visible in the Inspector panel)
   // The speed of player movement
   float speed = 10;

   // Flag indicating whether the player is at the
   // left edge of the screen
   bool atLeftWall = false;

   // Flag indicating whether the player is at the
   // right edge of the screen
   bool atRightWall = false;

   // On collision with a trigger collider...
   void OnTriggerEnter2D(Collider2D other) {
      // Check the tag of the object the player
      // has collided with
      if(other.tag == "LeftWall") {
         // If collided with the left wall, set
         // the left wall flag to true
         atLeftWall = true;
      } else if(other.tag == "RightWall") {
         // If collided with the right wall, set
         // the right wall flag to true
         atRightWall = true;
      }
   }

   // When no longer colliding with an object...
   void OnTriggerExit2D(Collider2D other) {
      // Check the tag of the object the player
      // has ceased to collide with
      if(other.tag == "LeftWall") {
         // If collided with the left wall, set
         // the left wall flag to true
         atLeftWall = false;
      } else if(other.tag == "RightWall") {
         // If collided with the right wall, set
         // the right wall flag to true
         atRightWall = false;
      }
   }

   // Update is called once per frame
   void Update () {
      // Player movement from input (it's a variable between -1 and 1) for
      // degree of left or right movement
      float movementInput = Input.GetAxis("Horizontal");
```

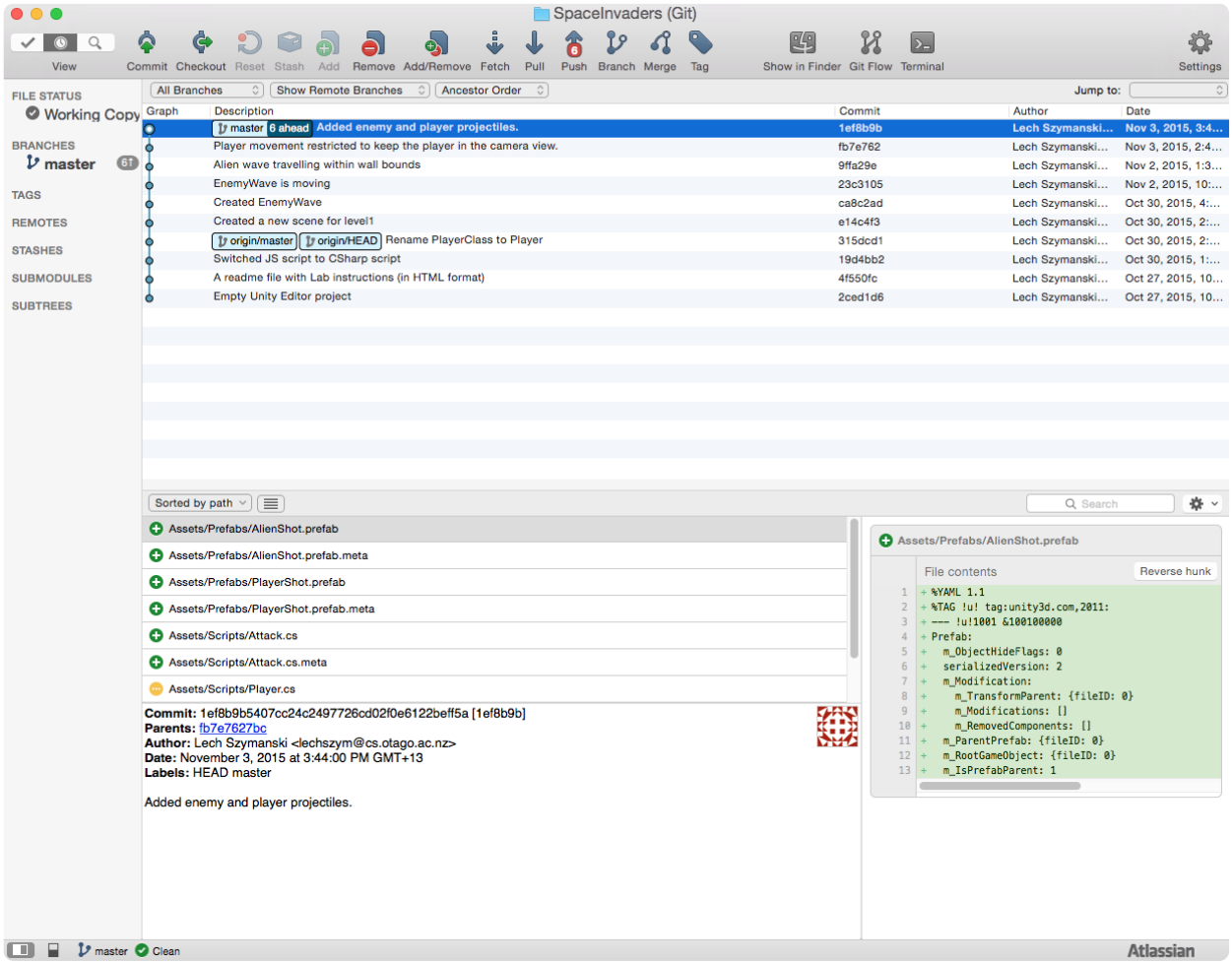
```
52:
53:         // If close to wall and moving towards it,
54:         // stop the movment
55:         if(atLeftWall && (movementInput < 0) )  {
56:            movementInput = 0;
57:         }
58:         if(atRightWall && (movementInput > 0) )  {
59:            movementInput = 0;
60:         }
61:
62:         // Move the player object
63:         transform.Translate( new Vector3(Time.deltaTime * speed * movementInput,0,0))
64:
65:         if(Input.GetButton("Jump")) {
66:            // Get player's attack component
67:            // and execute its Shoot() method
68:            Attack attack = GetComponent<Attack>();
69:            attack.Shoot();
70:         }
71:      }
72:   }
```

The GetButton() (http://docs.unity3d.com/ScriptReference/Input.GetButton.html) method, like the *GetAxis()*, is a pre-built method for indicating user input on pre-defined set of keys. In the default setting "Jump" corresponds to the *Space* key.

7. Hit play - should be able to shoot now.



8. In SourceTree, stage all the changes and commit.



Projectile collisions

In order for a projectile to hit an *Alien* or the *Player*, more collisions need to be detected.

9. Find the *AlienShot* prefab in the *Assets | Prefabs* and add the *Physics 2D | Circle Collider 2D* component. Enable the *Is Trigger* flag and set the *Radius* of the collider to 0.2.

10. Do the same for the *PlayerShot* prefab. Enable the *Is Trigger* flag as well and and set the *Radius* of the collider to 0.12.

11. These colliders are set as triggers, because the physics engine is not supposed to decide what to do on their collisions. Instead, you'll be handling the collision events in scripts.

12. Add the following code to the *OnTriggerEnter2D()* function in the "Player" script:

Player.cs

```
using UnityEngine;

public class Player : MonoBehaviour {

   // Private variables (not visible in the Inspector panel)
   // The speed of player movement
   float speed = 10;

   // Flag indicating whether the player is at the
   // left edge of the screen
   bool atLeftWall = false;

   // Flag indicating whether the player is at the
   // right edge of the screen
   bool atRightWall = false;

   // On collision with a trigger collider...
   void OnTriggerEnter2D(Collider2D other) {
      // Check the tag of the object the player
      // has collided with
      if(other.tag == "LeftWall") {
         // If collided with the left wall, set
         // the left wall flag to true
         atLeftWall = true;
      } else if(other.tag == "RightWall") {
         // If collided with the right wall, set
         // the right wall flag to true
         atRightWall = true;
      } else {
         // Collision with something that is not a wall
         // Check if collided with a projectile
         // A projectile has a Projectile script component,
         // so try to get a reference to that component
         Projectile projectile = other.GetComponent<Projectile>();

         //If that refernce is not null, then check if it's an enemyProjectile
         if(projectile != null && projectile.enemyProjectile) {
            // Collided with an enemy projectile

            // Destroy the projectile game object
            Destroy(other.gameObject);

            // Destroy self
            Destroy(gameObject);
         }
      }
   }

   // When no longer colliding with an object...
   void OnTriggerExit2D(Collider2D other) {
      // Check the tag of the object the player
```

```
52:         // has ceased to collide with
53:         if(other.tag == "LeftWall") {
54:            // If collided with the left wall, set
55:            // the left wall flag to true
56:            atLeftWall = false;
57:         } else if(other.tag == "RightWall") {
58:            // If collided with the right wall, set
59:            // the right wall flag to true
60:            atRightWall = false;
61:         }
62:      }
63:
64:      // Update is called once per frame
65:      void Update () {
66:         // Player movement from input (it's a variable between -1 and 1) for
67:         // degree of left or right movement
68:         float movementInput = Input.GetAxis("Horizontal");
69:
70:         // If close to wall and moving towards it,
71:         // stop the movment
72:         if(atLeftWall && (movementInput < 0) )  {
73:            movementInput = 0;
74:         }
75:         if(atRightWall && (movementInput > 0) )  {
76:            movementInput = 0;
77:         }
78:
79:         // Move the player object
80:         transform.Translate( new Vector3(Time.deltaTime * speed * movementInput,0,0))
81:
82:         if(Input.GetButton("Jump")) {
83:            // Get player's attack component
84:            // and execute its Shoot() method
85:            Attack attack = GetComponent<Attack>();
86:            attack.Shoot();
87:         }
88:      }
89:   }
```

The added block of code is executed when the *Player* object collides with a trigger that is not a `LeftWall` or a `RightWall`. In order to check whether the *other* collider comes from a projectile, a check is made to see if it's got a *Projectile* component. The *GetComponent()* method returns a non-null reference if that's the case.

At that point, another check needs to be made, whether the projectile is an enemy projectile or not. If the *enemyProjectile* flag of the fetched *Projectile* component is true, then the *Player* has been hit by an alien projectile. When that occurs, the Destroy() (http://docs.unity3d.com/ScriptReference/Object.Destroy.html) function is used to remove the *other* game object (the enemy projectile) and the current game object (the *Player*) from the scene.

3. Play the game. Once hit with a bomb, the *Player* game object and the projectile should disappear from the scene.

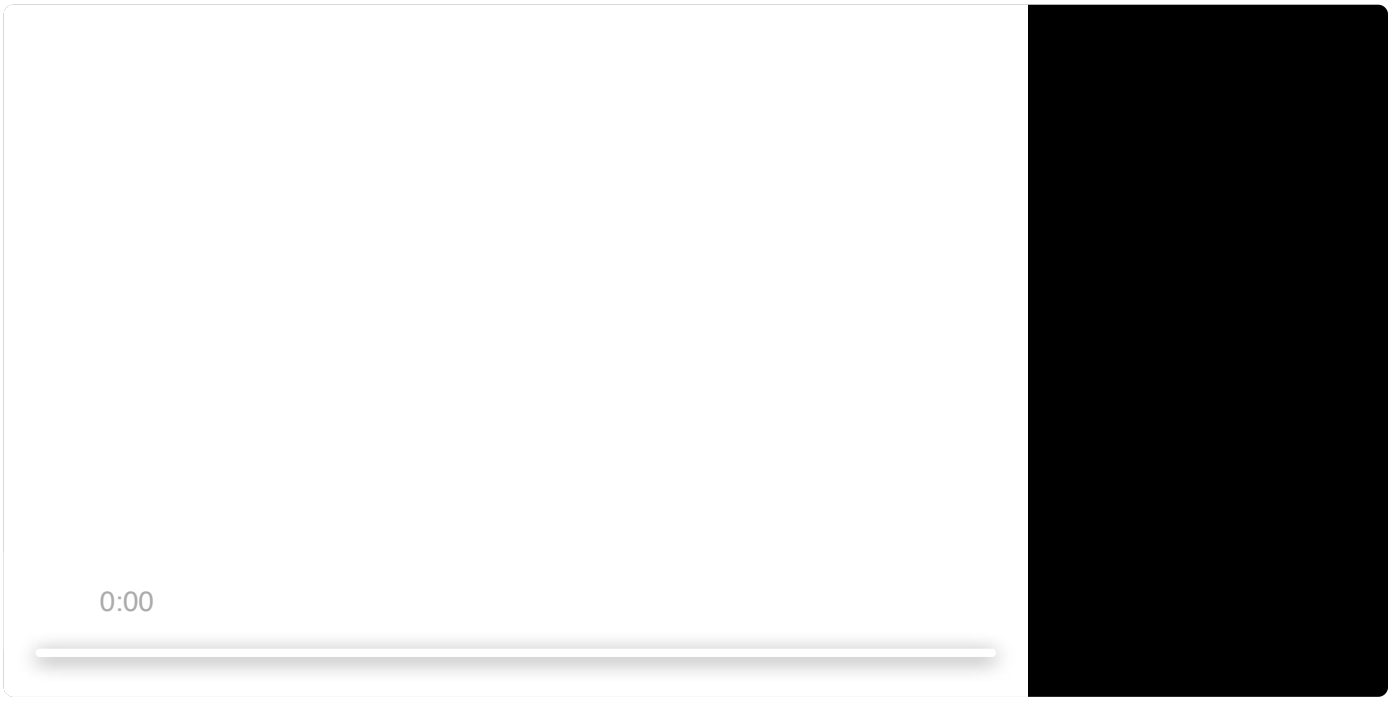
4. Add the following code to the *OnTriggerEnter2D()* method of the "Alien" script:

Alien.cs

```
using UnityEngine;

public class Alien : MonoBehaviour {

   // When enemy collides with an object with a
   // collider that is a trigger...
   void OnTriggerEnter2D(Collider2D other) {
      EnemyWave wave;

      // Check if colliding with the left or right wall
      // (by checking the tags of the collider that the enemy
      //  collided with)
      if(other.tag == "LeftWall") {
         // If collided with the left wall, get a reference
         // to the EnemyWave object, which should be a component
         // of enemies parent
         wave = transform.parent.GetComponent<EnemyWave>();
         // Set direction of the wave
         wave.SetDirectionRight();
      } else if(other.tag == "RightWall") {
         // If collided with the right wall, get a reference
         // to the EnemyWave object, which should be a component
         // of enemies parent
         wave = transform.parent.GetComponent<EnemyWave>();
         // Set direction of the wave
         wave.SetDirectionLeft();
      } else {
         // Collision with something that is not a wall
         // Check if collided with a projectile
         // A projectile has a Projectile script component,
         // so try to get a reference to that component
         Projectile projectile = other.GetComponent<Projectile>();

         //If that refernce is not null, then check if it's an enemyProjectile
         if(projectile != null && !projectile.enemyProjectile) {
            // Collided with non enemy projectile (so a player projectile)

            // Destroy the projectile game object
            Destroy(other.gameObject);

            // Destroy self
            Destroy(gameObject);
         }
      }
   }
}
```

Just like in the "Player" script, this code follows a test for collision with the walls, and checks for a collision with a projectile. The only difference is that the hit occurs if the *enemyProjectile* flag is false, signifying a collision with the player's projectile.

5. Now you should be able to shoot the aliens when you play the game.



6. Did you notice that the game objects for the alien shots that miss and go off screen linger forever in the **Hierarchy panel**? That's because these shots are still in the scene, just off camera. They don't get rendered, but they are still in the scene. Their scripts still get executed, and if too many of those objects are lingering around, eventually they can slow down the whole game. Therefore, you need to make sure that a projectile that goes off screen gets destroyed.

7. Create another C# script. Name it *Utility* and paste in the following code:

Utility.cs

```
01: using UnityEngine;
02:
03: public class Utility : MonoBehaviour {
04:
05:    // Returns true if game object is visible by the specified camera,
06:    // otherwise returns false.
07:    public static bool isVisible(Renderer renderer, Camera camera) {
08:       Plane[] planes = GeometryUtility.CalculateFrustumPlanes(camera);
09:       return GeometryUtility.TestPlanesAABB(planes, renderer.bounds);
10:    }
11: }
```

This is a neat little function which will tell you if a given object is in the view of a specified camera. This method is *static*, which means it can invoked without creating an instance of the *Utility* object. Remember, in Unity, every script is taken as a definition of a class with the same name. When you add script components to game objects, you are instructing the engine to create (and attach) an instance of an object of the corresponding class. These object instances carry their own state in their public and private variables. Static variables are shared across all the object instances of a particular class. You can think of them as "global" variables. Static methods are methods that do not depend on the internal state of an

object instance. They cannot read or write to internal variables unless these variables are static. Since they don't depend on the state of a given object, static methods can be referenced through their class (script) name.

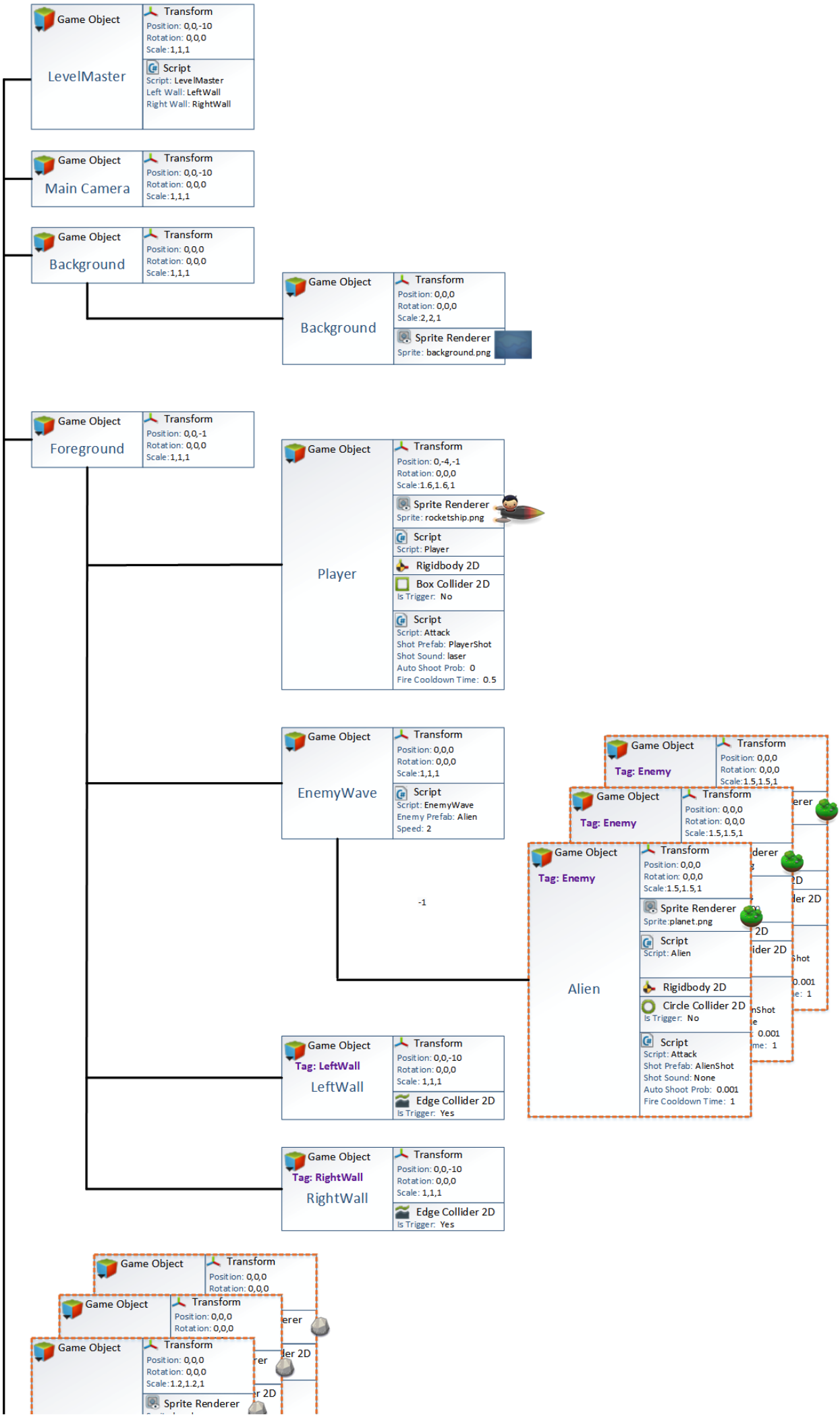
18. Here's how to use the *isVisible()* static method to assure projectile destruction once it goes off screen:

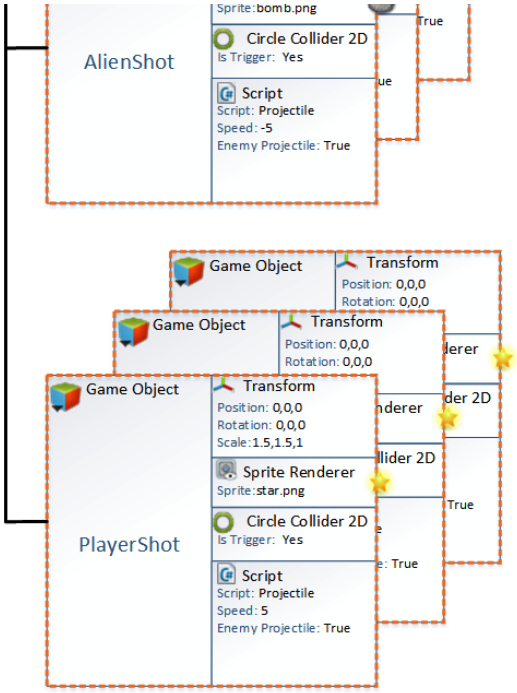
Projectile.cs

```
01: using UnityEngine;
02:
03: public class Projectile : MonoBehaviour {
04:
05:    // The speed fo the projectile
06:    public float speed;
07:
08:    // Flag identifyng whether the projectile
09:    // is sent by enemy or the player
10:    public bool enemyProjectile;
11:
12:    // Update is called once per frame
13:    void Update () {
14:       // The projectile travels up (in the direction of positive y axis), but
15:       // the movement is multiplies by speed (so negative speed will get
16:       // move the projectile down)
17:       transform.Translate(Vector3.up * speed * Time.deltaTime);
18:
19:       // Check if the game object is visible, if not, destroy self
20:       if(!Utility.isVisible(GetComponent<Renderer>(), Camera.main)) {
21:          Destroy(gameObject);
22:       }
23:    }
24: }
```

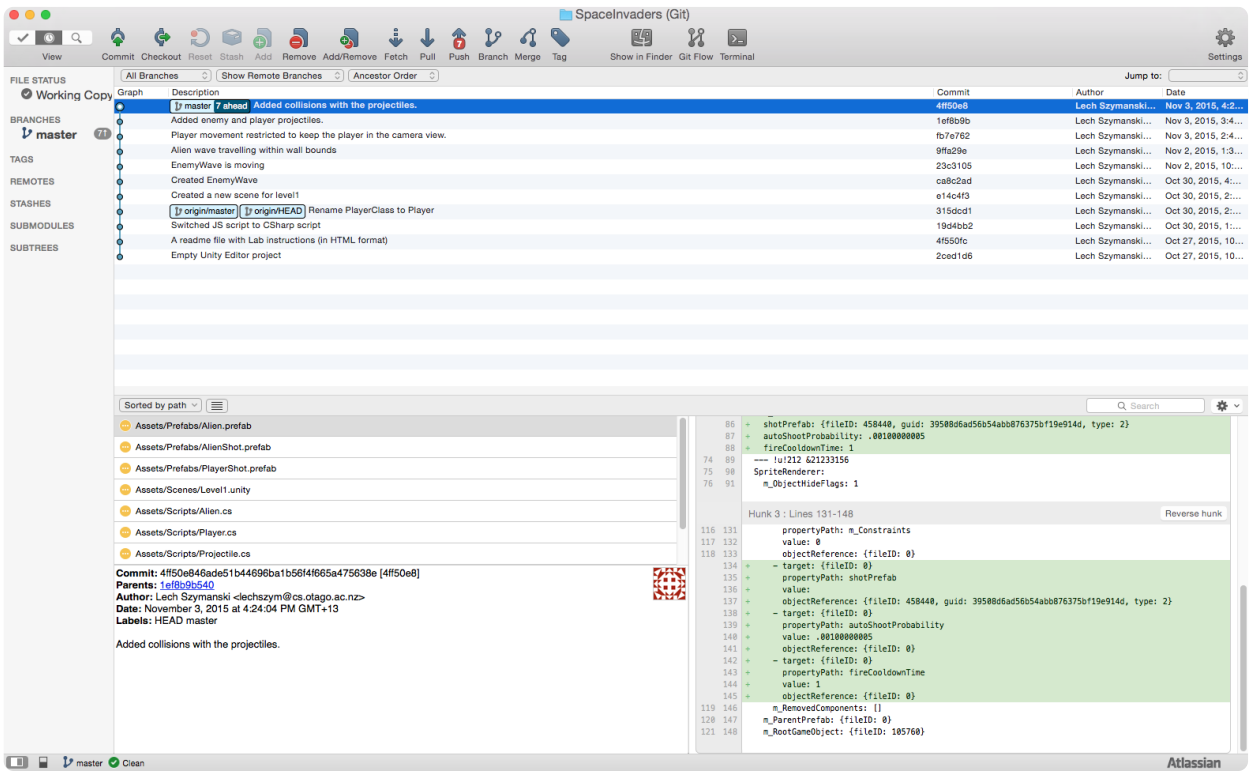
The *GetComponent<Renderer>()* call will fetch a reference for the renderer of the game object - every sprite has a renderer component. The new code checks if the renderer is visible from the *MainCamera*'s point of view - if not, it destroys itself.

19. The diagram below summarises the current game object tree:


Game Object
LevelMaster
Transform
Position: 0,0,-10
Rotation: 0,0,0
Scale:1,1,1
Script
Script: LevelMaster
Left Wall: LeftWall
Right Wall: RightWall
Game Object
Main Camera
Transform
Position: 0,0,-10
Rotation: 0,0,0
Scale:1,1,1
Game Object
Background
Transform
Position: 0,0,0
Rotation: 0,0,0
Scale:1,1,1
Game Object
Background
Transform
Position: 0,0,0
Rotation: 0,0,0
Scale:2,2,1
Sprite Renderer
Sprite: background.png
Game Object
Foreground
Transform
Position: 0,0,-1
Rotation: 0,0,0
Scale:1,1,1
Game Object
Player
Transform
Position: 0,-4,-1
Rotation: 0,0,0
Scale:1.6,1.6,1
Sprite Renderer
Sprite: rocketship.png
Script
Script: Player
Rigidbody 2D
Box Collider 2D
Is Trigger: No
Script
Script: Attack
Shot Prefab: PlayerShot
Shot Sound: laser
Auto Shoot Prob: 0
Fire Cooldown Time: 0.5
Game Object
EnemyWave
Transform
Position: 0,0,0
Rotation: 0,0,0
Scale:1,1,1
Script
Script: EnemyWave
Enemy Prefab: Alien
Speed: 2
-1
Game Object
Tag: Enemy
Alien
Transform
Position: 0,0,0
Rotation: 0,0,0
Scale:1.5,1.5,1
Sprite Renderer
Sprite:planet.png
Script
Script: Alien
Rigidbody 2D
Circle Collider 2D
Is Trigger: No
Script
Script: Attack
Shot Prefab: AlienShot
Shot Sound: None
Auto Shoot Prob: 0.001
Fire Cooldown Time: 1
Game Object
Tag: LeftWall
LeftWall
Transform
Position: 0,0,-10
Rotation: 0,0,0
Scale: 1,1,1
Edge Collider 2D
Is Trigger: Yes
Game Object
Tag: RightWall
RightWall
Transform
Position: 0,0,-10
Rotation: 0,0,0
Scale: 1,1,1
Edge Collider 2D
Is Trigger: Yes
Game Object
Transform
Position: 0,0,0
Rotation: 0,0,0
Scale:1.2,1.2,1
Sprite Renderer



0. In SourceTree, stage all the changes and commit.



Game Master

You're done with the game mechanics. Time to implement behaviour that keeps track of player lives and score.

1. Create a new C# Script. Name it "GameMaster" and paste in the following code:

GameMaster.cs

```
01: using UnityEngine;
02: using UnityEngine.SceneManagement;
03:
04: public class GameMaster : MonoBehaviour {
05:
06:
07:    // Static variables - there's only one instance
08:    // of this variable for the entire game
09:
10:    // Player health - always start with 3 lives
11:    public static int playerHealth = 3;
12:    // Player score
13:    public static int playerScore = 0;
14:
15:    // Method to call when enemy is hit
16:    public static void EnemyHit(Alien alien) {
17:       // Add enemy points to player's score
18:       playerScore += alien.points;
19:    }
20:
21:    // Method to call when player is hit
22:    public static void PlayerHit() {
23:       playerHealth--;
24:       // Reduce player's lives
25:       if(playerHealth > 0) {
26:          // If more lives left, then reload the
27:          // level
28:          SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex);
29:       }
30:    }
31: }
```

All the methods and internal variables of the *GameMaster* class are static - they have a global scope, and their values are shared among all object instances of the *GameMaster* class. In fact, there is no needs to instantiate *GameMaster* in order to use any of its static methods or variables.

The two *static* variables, *playerHealth* and *playerScore* keep track of the number of lives the player has left (starting out with 3) and the player score (starting out with 0)

The *EnemyHit()* method is meant to be invoked after an alien is hit, so that player's score can be updated. This method accepts a reference to an object of *Alien* type, which contains a variable specifying the number of points the alien is worth (you'll be adding this variable in the *Alien* shortly). The reason why a reference to the entire object is passed into the function, instead of just an integer value for the points, will become apparent later on, when the Game Over condition gets implemented.

The *PlayerHit()* method is meant to be invoked after the player is hit. The *playerHealth* variable is decremented for every hit. Once it reaches 0, the current scene is reloaded. The *SceneManager.LoadScene()* method can take a string name of the scene as an argument, or a reference to the scene - *SceneManager.GetActiveScene().buildIndex* is a reference to the currently loaded scene. Note the use of the library: *using UnityEngine.SceneManagement;* - this library is needed for Unity to understand the reference to the *SceneManager*.

2. To report an alien hit and specify number of points an alien is worth, add the following code to the "Alien" script:

Alien.cs

```
01: using UnityEngine;
02:
03: public class Alien : MonoBehaviour {
04:
05:    //Points the alien is worth
06:    public int points = 100;
07:
08:    // When enemy collides with an object with a
09:    // collider that is a trigger...
10:    void OnTriggerEnter2D(Collider2D other) {
11:       EnemyWave wave;
12:
13:       // Check if colliding with the left or right wall
14:       // (by checking the tags of the collider that the enemy
15:       //  collided with)
16:       if(other.tag == "LeftWall") {
17:          // If collided with the left wall, get a reference
18:          // to the EnemyWave object, which should be a component
19:          // of enemies parent
20:          wave = transform.parent.GetComponent<EnemyWave>();
21:          // Set direction of the wave
22:          wave.SetDirectionRight();
23:       } else if(other.tag == "RightWall") {
24:          // If collided with the right wall, get a reference
25:          // to the EnemyWave object, which should be a component
26:          // of enemies parent
27:          wave = transform.parent.GetComponent<EnemyWave>();
28:          // Set direction of the wave
29:          wave.SetDirectionLeft();
30:       } else {
31:          // Collision with something that is not a wall
32:          // Check if collided with a projectile
33:          // A projectile has a Projectile script component,
34:          // so try to get a reference to that component
35:          Projectile projectile = other.GetComponent<Projectile>();
36:
37:          //If that refernce is not null, then check if it's an enemyProjectile
38:          if(projectile != null && !projectile.enemyProjectile) {
39:             // Collided with non enemy projectile (so a player projectile)
40:
41:             // Destroy the projectile game object
42:             Destroy(other.gameObject);
43:
44:             // Report enemy hit to the game master
45:             GameMaster.EnemyHit(this);
46:
47:             // Destroy self
48:             Destroy(gameObject);
49:          }
50:       }
```

```
51:     }
52: }
```

The *Alien* component of the *Alien* game object, which has collided with a non-enemy projectile, passes a reference to itself (*this*) to the *EnemyHit()* method of *GameMaster*.

3. To report a player hit, add the following code to the "Player" script:

Player.cs

```
001: using UnityEngine;
002:
003: public class Player : MonoBehaviour {
004:
005:    // Private variables (not visible in the Inspector panel)
006:    // The speed of player movement
007:    float speed = 10;
008:
009:    // Flag indicating whether the player is at the
010:    // left edge of the screen
011:    bool atLeftWall = false;
012:
013:    // Flag indicating whether the player is at the
014:    // right edge of the screen
015:    bool atRightWall = false;
016:
017:    // On collision with a trigger collider...
018:    void OnTriggerEnter2D(Collider2D other) {
019:       // Check the tag of the object the player
020:       // has collided with
021:       if(other.tag == "LeftWall") {
022:          // If collided with the left wall, set
023:          // the left wall flag to true
024:          atLeftWall = true;
025:       } else if(other.tag == "RightWall") {
026:          // If collided with the right wall, set
027:          // the right wall flag to true
028:          atRightWall = true;
029:       } else {
030:          // Collision with something that is not a wall
031:          // Check if collided with a projectile
032:          // A projectile has a Projectile script component,
033:          // so try to get a reference to that component
034:          Projectile projectile = other.GetComponent<Projectile>();
035:
036:          //If that refernce is not null, then check if it's an enemyProjectile
037:          if(projectile != null && projectile.enemyProjectile) {
038:             // Collided with an enemy projectile
039:
040:             // Destroy the projectile game object
041:             Destroy(other.gameObject);
042:
043:             // Report player hit to the game master
044:             GameMaster.PlayerHit();
045:
046:             // Destroy self
047:             Destroy(gameObject);
048:          }
049:       }
050:    }
051:
```

```
052:     // When no longer colliding with an object...
053:     void OnTriggerExit2D(Collider2D other) {
054:        // Check the tag of the object the player
055:        // has ceased to collide with
056:        if(other.tag == "LeftWall") {
057:           // If collided with the left wall, set
058:           // the left wall flag to true
059:           atLeftWall = false;
060:        } else if(other.tag == "RightWall") {
061:           // If collided with the right wall, set
062:           // the right wall flag to true
063:           atRightWall = false;
064:        }
065:     }
066:
067:     // When player collides with an object that is
068:     // not a trigger...
069:     void OnCollisionEnter2D(Collision2D other) {
070:        // If the other object is tagged as "Player"...
071:        if(other.gameObject.tag == "Enemy") {
072:
073:           // Report player hit to the game master
074:           GameMaster.PlayerHit();
075:
076:           // Destroy the Player game object
077:           Destroy(gameObject);
078:        }
079:     }
080:
081:     // Update is called once per frame
082:     void Update () {
083:        // Player movement from input (it's a variable between -1 and 1) for
084:        // degree of left or right movement
085:        float movementInput = Input.GetAxis("Horizontal");
086:
087:        // If close to wall and moving towards it,
088:        // stop the movment
089:        if(atLeftWall && (movementInput < 0) )  {
090:           movementInput = 0;
091:        }
092:        if(atRightWall && (movementInput > 0) )  {
093:           movementInput = 0;
094:        }
095:
096:        // Move the player object
097:        transform.Translate( new Vector3(Time.deltaTime * speed * movementInput,0,0)
098:
099:        if(Input.GetButton("Jump")) {
100:           // Get player's attack component
101:           // and execute its Shoot() method
102:           Attack attack = GetComponent<Attack>();
103:           attack.Shoot();
```

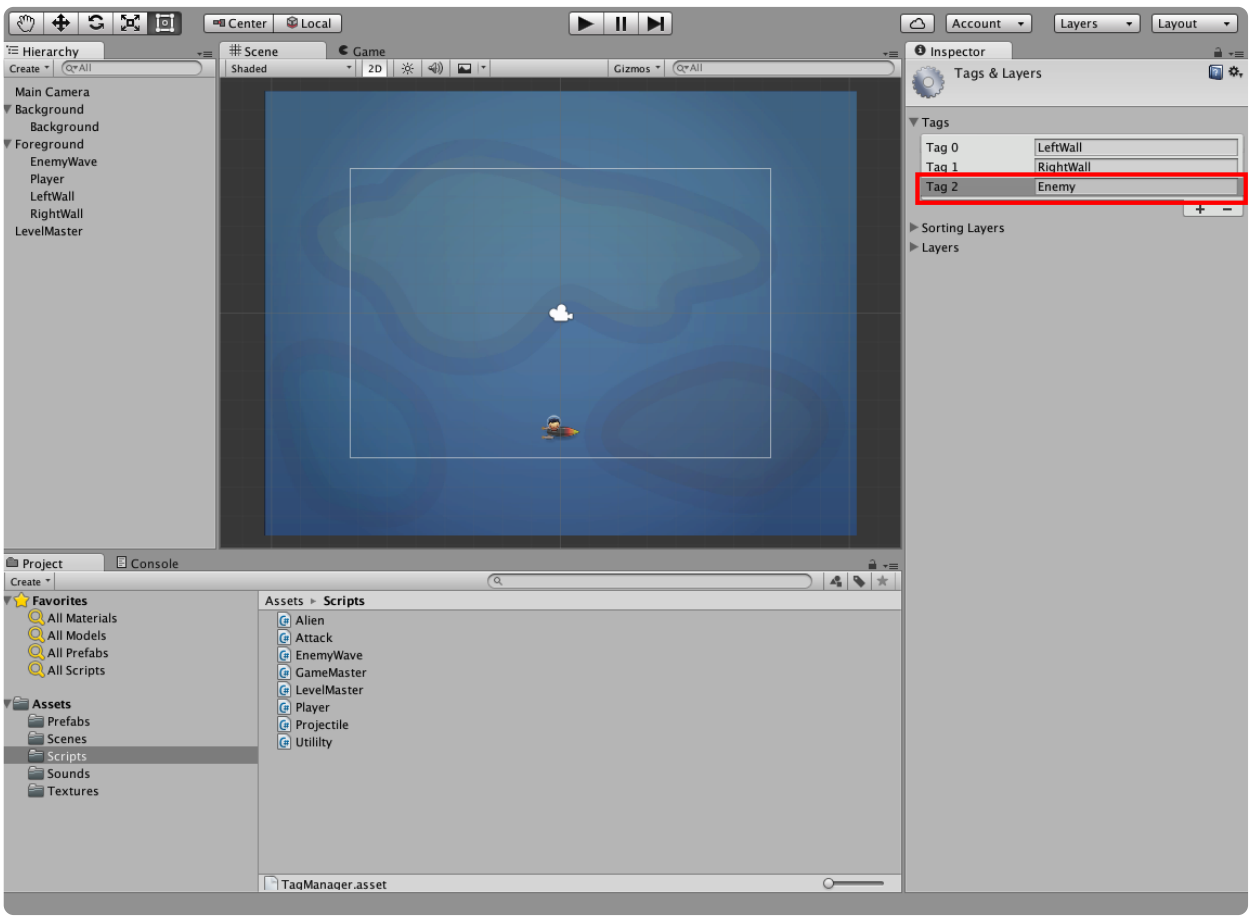
```
104:         }
105:     }
106: }
```

The first condition for declaring a hit is on collision with the enemy projectile. The second condition is for a collision with the alien object. This is to handle the case when the wave gets all the way to the bottom and one of the aliens collides with the player. Since both colliders for an *Alien* and the *Player* are not triggers, the collision between the two object invokes invokes *OnCollisionEnter2D()* method, not the *OnTriggerEnter2D()* method. Though in the game there will be non-trigger colliders, other than *Aliens*, to collide with, it's best to verify what the *Player* has collided with. This is done by checking whether the *other* object carries the "Enemy" tag.

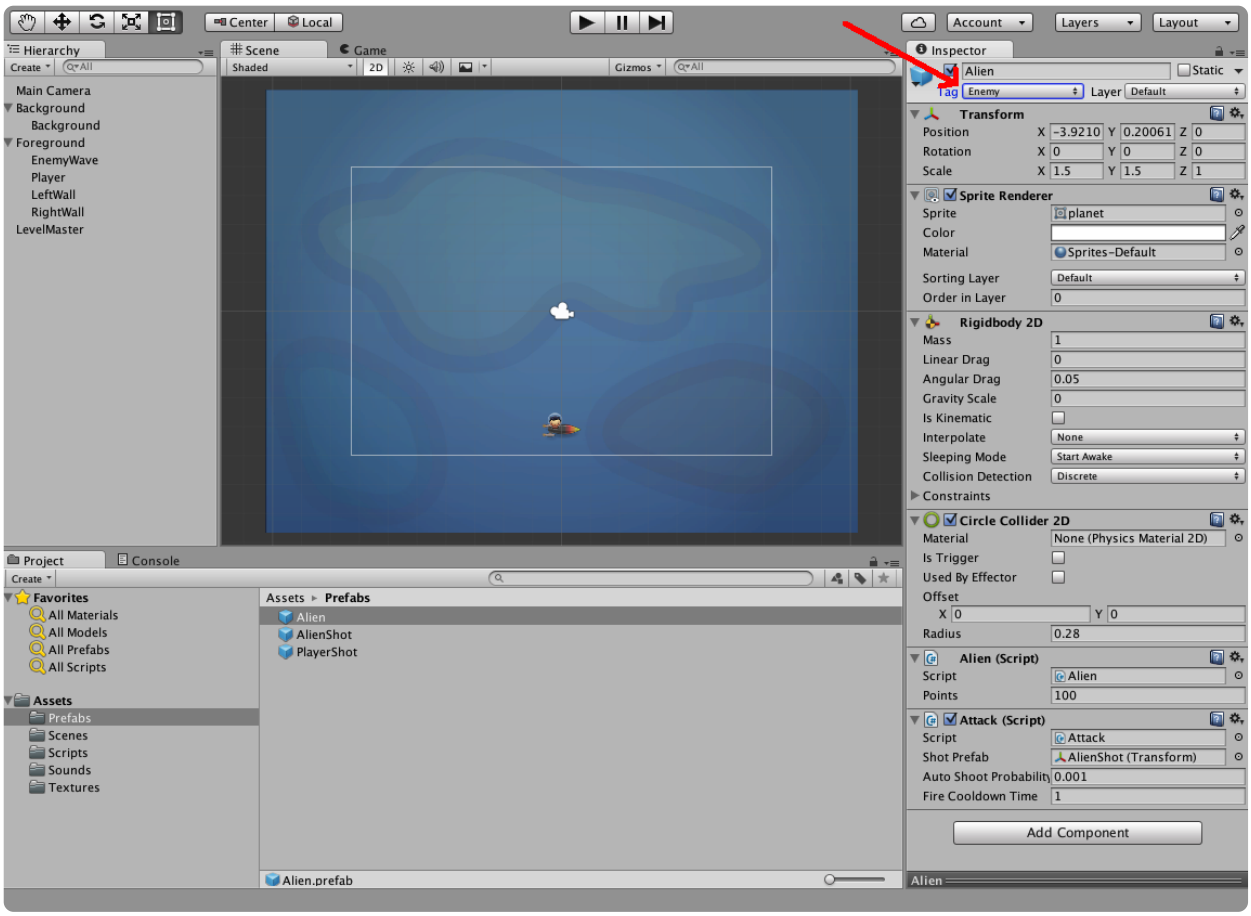
Collider vs. collision

There is a difference between the *other* parameter passed in to the *OnTrigger()* and *OnCollision()* events. In the *OnTrigger()* it refers to *Collider2D* (http://docs.unity3d.com/ScriptReference/Collider2D.html), whereas in *OnCollision()* it refers to *Collision2D* (http://docs.unity3d.com/ScriptReference/Collision2D.html). The difference is subtle. Whereas inside the *OnTrigger()* event the tag of the colliding object can be accessed using the *other.tag* reference, in the *OnCollision()* this is done through reference to *other.gameObject.tag*.

4. Create a new tag. From the main menu select *Edit | Project Settings | Tags and Layers*. Expand the *Tags* and add a new tag string: "Enemy".

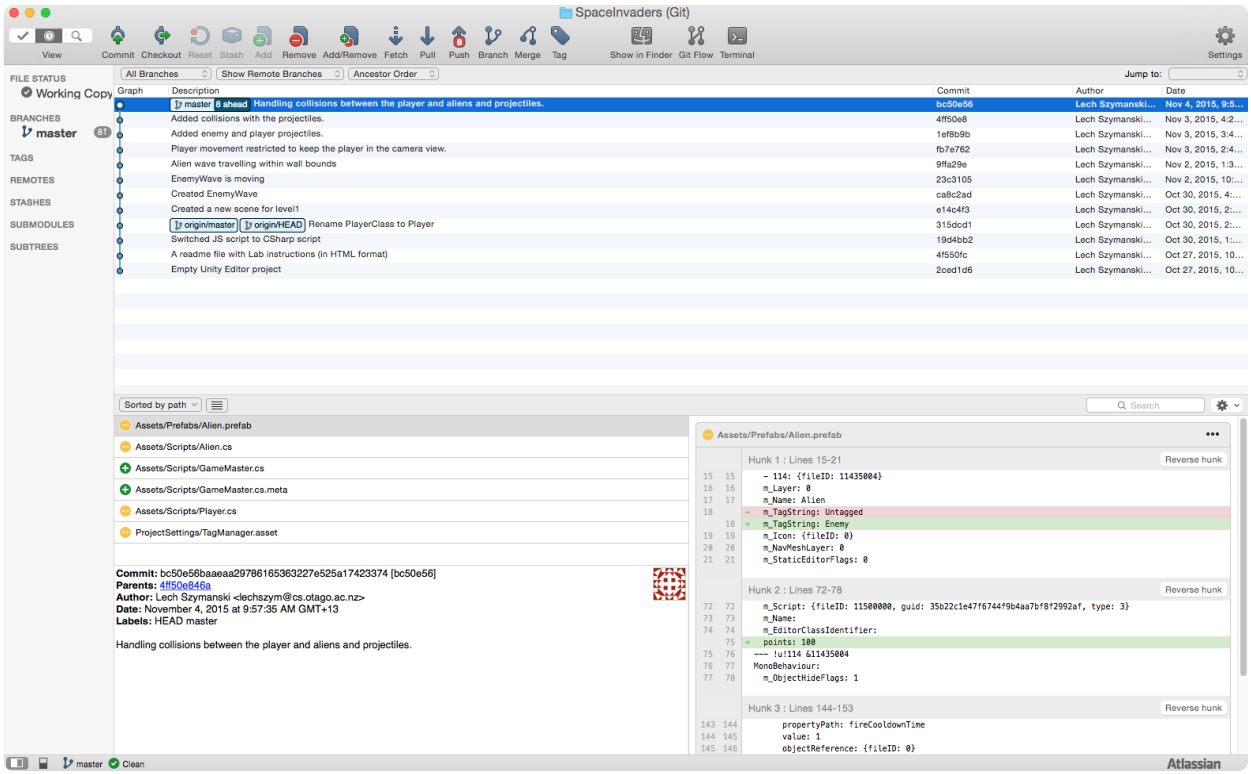


5. Find the *Alien* prefab in the *Assets | Prefabs* and tag it with the "Enemy" tag.



6. When playing, the level should reload the first three times the player gets hit. After the fourth hit the *Player* sprite will remain gone. The score is not visible at the moment - you need a GUI to display it.

7. In SourceTree, stage all the changes and commit.



Simple GUI

Unity provides a powerful system for creating User Interfaces (UI). However, it takes some time to learn how to use it, and an entire lab (../lab07/index) is dedicated to it later. For this game the legacy GUI system should suffice.

8. Every script component can override the *OnGUI()* (http://docs.unity3d.com/ScriptReference/MonoBehaviour.OnGUI.html) method. This method gets invoked by the game engine a number of times in-between rendering of the frames, and is meant to provide the programmer with a means of creating a simple user interface. For details on the execution order of different methods in the game loop see the article on Execution Order of Event Functions (http://docs.unity3d.com/Manual/ExecutionOrder.html) in the Unity manual.

9. Update the "LevelMaster" script with the following code:

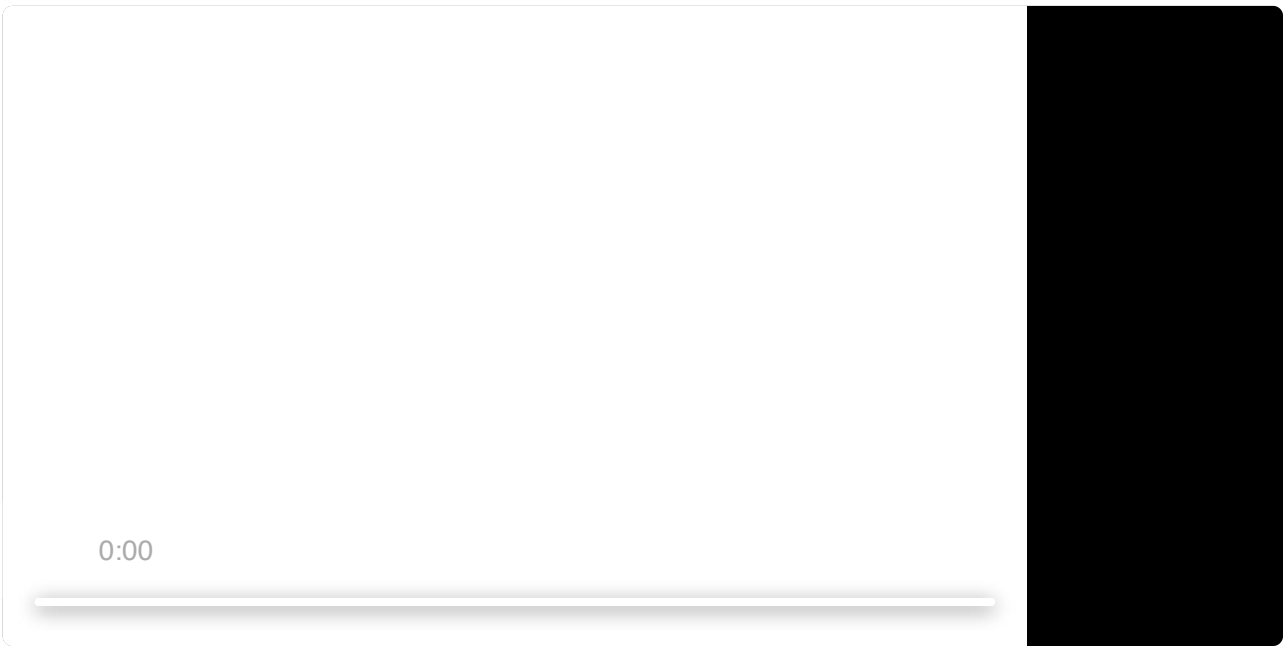
LevelMaster.cs

```
01: using UnityEngine;
02:
03: public class LevelMaster : MonoBehaviour {
04:
05:    // Variables referencing two edge colliders
06:    public EdgeCollider2D leftWall;
07:    public EdgeCollider2D rightWall;
08:
09:    // Use this for initialization
10:    void Start () {
11:       // Get the width and height of the camera (in pixels)
12:       float screenW = Camera.main.pixelWidth;
13:       float screenH = Camera.main.pixelHeight;
14:
15:       // Create an array consisting of two Vector2 object
16:       Vector2[] edgePoints = new Vector2[2];
17:
18:       // Convert screen coordinates point (0,0) to world coordinates
19:       Vector3 leftBottom  = Camera.main.ScreenToWorldPoint(new Vector3(0f, 0f, 0f))
20:       // Convert screen coordinates point (0,H) to world coordinates
21:       Vector3 leftTop = Camera.main.ScreenToWorldPoint(new Vector3(0f, screenH, 0f)
22:
23:       // Set the two points in the array to screen left bottom
24:       // and screen left top points
25:       edgePoints[0] = leftBottom;
26:       edgePoints[1] = leftTop;
27:
28:       // Position the left wall edge collider
29:       // at the left edge of the camera
30:       leftWall.points = edgePoints;
31:
32:       // Convert screen coordinates point (W,0) to world coordinates
33:       Vector3 rightBottom = Camera.main.ScreenToWorldPoint(new Vector3(screenW, 0f,
34:       // Convert screen coordinates point (W,H) to world coordinates
35:       Vector3 rightTop = Camera.main.ScreenToWorldPoint(new Vector3(screenW, screenH
36:
37:       // Set the two points in the array to screen right bottom
38:       // and screen right top points
39:       edgePoints[0] = rightBottom;
40:       edgePoints[1] = rightTop;
41:
42:       // Position the left wall edge collider
43:       // at the left edge of the camera
44:       rightWall.points = edgePoints;
45:    }
46:
47:    // HUD
48:    void OnGUI() {
49:       // Show player score in white on the top left of the screen
50:       GUI.color = Color.white;
51:       GUI.skin.label.alignment = TextAnchor.UpperLeft;
```

```
52:         GUI.skin.label.fontSize = 40;
53:         GUI.skin.label.fontStyle = FontStyle.Bold;
54:         GUI.Label(new Rect(20,20,500,100), "Score: " + GameMaster.playerScore);
55:
56:         // Show the player lives in red on the top right of the screen
57:         GUI.color = Color.red;
58:         GUI.skin.label.alignment = TextAnchor.UpperRight;
59:         GUI.skin.label.fontSize = 40;
60:         GUI.skin.label.fontStyle = FontStyle.Bold;
61:         GUI.Label(new Rect(Screen.width - 320,20,300,100), "Lives: " + GameMaster.pla
62:     }
63: }
```

The *OnGUI()* method will be invoked by the game engine a number of times per frame. In this implementation of the method, the built-in *GUI* class is used to display text on the screen using its Label() (http://docs.unity3d.com/ScriptReference/GUI.Label.html) method. The look of the label is defined by changing the context of the *GUI* class before the call to the *Label()* method. Position and size of the label are defined with the *Rect* object passed in to *Label()* as the first argument. The text to display goes in as the second argument. C# concatenates strings with the "+" operator. It also does automatic conversion from numbers to strings, which is why *playerScore* and *playerHealth* variables from *GameMaster* can be concatenated to strings with the "+".

10. If you play the game, the score and number of lives should get displayed near the top of the screen.



Game over

At the moment, once the player loses all their lives or kills all the aliens, nothing happens. Let's create a game over screen.

11. Start by creating a new scene. From the main menu select *File* | *New Scene*.

12. Save the scene into *Assets* | *Scenes* as "GameOver".

3. Create a new C# script in *Assets / Scripts*. Name it *GameOver*. Double-click to open in VSC editor and paste in the following code:

GameOver.cs

```
01: using UnityEngine;
02:
03: public class GameOver : MonoBehaviour {
04:
05:    // Display game over message
06:    void OnGUI() {
07:
08:       // Show player score in white on the top left of the screen
09:       GUI.color = Color.white;
10:       GUI.skin.label.alignment = TextAnchor.MiddleCenter;
11:       GUI.skin.label.fontSize = 40;
12:       GUI.skin.label.fontStyle = FontStyle.Bold;
13:       GUI.Label(new Rect(0,Screen.height/ 4f,Screen.width,70), "Game over");
14:    }
15: }
```

In the OnGUI() method a label is displayed somewhere around the centre of the screen with "Game Over" as the text.

4. In order for the *GameOver* object to get instantiated, and the *OnGUI()* method to get executed by the game engine, the script must be a component of some game object in the scene. Instead of creating an object just for that purpose, just add the script to the "Main Camera" - it's a game object too.

5. Add the "GameOver" script component to the *Main Camera* game object.

6. You might also want to change the colour of the *Background* in the *Camera* properties of the *Main Camera* - try setting it to black.

7. Press run to see how the scene looks like.

8. Press stop and don't forget to save the scene.

9. How do you load the "GameOver" scene from within the game? There are two conditions for game over: when a player has no more lives left, or when they shoot down all the aliens. The *PlayerHit()* and *EnemyHit()* methods in the "GameMaster" script are ideal for detecting those conditions. Edit the "GameMaster" script and add the following changes:

GameMaster.cs

```
01: using UnityEngine;
02: using UnityEngine.SceneManagement;
03:
04: public class GameMaster : MonoBehaviour {
05:
06:
07:    // Static variables - there's only one instance
08:    // of this variable for the entire game
09:
10:    // Player health - always start with 3 lives
11:    public static int playerHealth = 3;
12:    // Player score
13:    public static int playerScore = 0;
14:
15:    // Method to call when enemy is hit
16:    public static void EnemyHit(Alien alien) {
17:       // Add enemy points to player's score
18:       playerScore += alien.points;
19:
20:       // Get the reference to alien's parent, the wave object
21:       Transform enemyWave = alien.transform.parent;
22:
23:       // Get an array of references to all children of the wave game object
24:       // who have an Alien component (so, we're looking for all the
25:       // aliens remaining in the wave)
26:       Component[] aliensLeft = enemyWave.GetComponentsInChildren<Alien>();
27:
28:       // If only one alien is left, that's the alien that just has been
29:       // hit and is about to be deleted...so no more aliens will be left
30:       if(aliensLeft.Length == 1) {
31:          SceneManager.LoadScene("GameOver");
32:       }
33:    }
34:
35:    // Method to call when player is hit
36:    public static void PlayerHit() {
37:       playerHealth--;
38:       // Reduce player's lives
39:       if(playerHealth > 0) {
40:          // If more lives left, then reload the
41:          // level
42:          SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex);
43:       } else {
44:          //No more lives left, load the GameOver scene
45:          SceneManager.LoadScene("GameOver");
46:       }
47:    }
48: }
```

Game over due to the player running out of lives is easy. All that is needed is a condition in the *PlayerHit()* method that detects if *playerHealth* is zero, at which point the "GameOver" scene is loaded.

Game over due to all aliens being gone is a bit more complicated. There's many ways of doing it. In this script this is done by checking the number of *Alien*s still left in the wave. When only a single alien (the one that got hit and invoked the *EnemyHit()* method) is left, it's game over. The reference to the hit alien is passed in as an argument of the *EnemyHit()* method. Reference to the *EnemyWave* game object is recovered via its *parent* variable. Calling the GetComponetnsInChildren() (http://docs.unity3d.com/ScriptReference/Component.GetComponentsInChildren.html) method of the *EnemyWave* game object returns an array with the list of all its children that also have an *Alien* component. The number of references in the resulting array corresponds to the number of *Alien*s left in the wave. If all that's left is the currently hit alien, the "GameOver" scene is loaded.

10. Test whether it works. When you press the play button the Unity Editor will run the currently open scene, which at the moment is the "GameOver" scene. Therefore, you need to open the "Level1" scene (by double-clicking it in *Assets | Scenes*) first. Don't forget to save the "GameOver" scene before the new one is loaded.

11. Play the game, and get killed three times (don't try to shoot all the aliens just yet - it takes too much time). The new screen doesn't load. There should be an error message at the bottom of the Unity Editor window: "Level 'GameOver'(-1) couldn't be loaded because it has not been added to the build settings." If you want see the message in detail go to the "Console" tab in the *Project panel*.

12. Whenever things don't work as expected, check for errors is the *Console*. It turns out that in order to move between the scenes, you need to configure the game's *Build Settings*.

13. Go to *File | Build Settings*. Press *Add current* to add "Level1" to *Scenes in the Build*. To add "GameOver" scene, you can drag it form *Assets |Scenes* to the *Scenes in the Build* window.

14. Close the Build window and try playing (and dying three times) again. This time the "GameOver" scene should load fine.

15. It would be nice to have some kind of indication in the "GameOver" scene whether the player did well or not. Add the following code to the "GameOver" script:

GameOver.cs

```
using UnityEngine;

public class GameOver : MonoBehaviour {

   // Display game over message
   void OnGUI() {

      // Show player score in white on the top left of the screen
      GUI.color = Color.white;
      GUI.skin.label.alignment = TextAnchor.MiddleCenter;
      GUI.skin.label.fontSize = 40;
      GUI.skin.label.fontStyle = FontStyle.Bold;
      GUI.Label(new Rect(0,Screen.height/ 4f,Screen.width,70), "Game over");

      string message;

      // Check player's lives left...if more than 0,
      // then player won, otherwise the game was lost
      // Generate appropriate final message
      if(GameMaster.playerHealth <= 0) {
         // The lost message will be shown in red
         message = "You lost :(";
         GUI.color = Color.red;
      } else {
         // The won message will be shown in white
         message = "You won!!!";
         GUI.color = Color.white;
      }
      // Show lost/won message
      GUI.Label(new Rect(0,Screen.height/ 4f + 80f,Screen.width,70), message);
   }
}
```

This code displays additional text in the "GameOver" screen. Depending on the value of *GameMaster.playerHealth* the text will indicate whether the game was won or lost.

6. In SourceTree, stage all the changes and commit.

Main menu

Time to create a scene that will serve as the main menu.

7. Create a new scene. Name it "MainMenu" and save to *Assets | Scenes*.

8. Create a new C# Script. Name it "MainMenu" and paste in the following code:

MainMenu.cs

```
using UnityEngine;
using UnityEngine.SceneManagement;

public class MainMenu : MonoBehaviour {

   // Update is called once per frame
   void Update () {
      if(Input.anyKey) {
         // Start the new game

         // Reset the player lives and
         // score
         GameMaster.playerHealth = 3;
         GameMaster.playerScore = 0;
         // Load the first level
         SceneManager.LoadScene("Level1");
      }
   }

   // Display main menu message
   void OnGUI() {
      GUI.color = Color.white;
      GUI.skin.label.alignment = TextAnchor.MiddleCenter;
      GUI.skin.label.fontSize = 40;
      GUI.skin.label.fontStyle = FontStyle.Bold;
      GUI.Label(new Rect(0,0,Screen.width,Screen.height), "Press any key to start")
   }
}
```

The *OnGUI()* method displays a message to "Press any key to continue...". The *Update()* function checks for key input and loads "Level1" once any key (http://vimeo.com/37714632) is hit. Recall that the variables of the *GameMaster* are static and will not get destroyed when new scenes are loaded. Hence, player lives and score need to be reset before the game is started.

9. Add the "MainMenu" script component to the *Main Camera* in the "MainMenu" scene. Change the *Background* of the *Main Camera* to black.

10. In *File | Build Settings* make sure to add "MainMenu" (current scene) to *Scenes in Build*. Drag the "MainMenu" scene to the top - it's the first scene to be loaded when the game starts.

11. Add the following code to the "GameOver" script:

GameOver.cs

```
01: using UnityEngine;
02: using UnityEngine.SceneManagement;
03:
04: public class GameOver : MonoBehaviour {
05:
06:    // Update is called once per frame
07:    void Update () {
08:       if(Input.anyKey) {
09:          // Go back to main menu
10:          SceneManager.LoadScene("MainMenu");
11:       }
12:    }
13:
14:    // Display game over message
15:    void OnGUI() {
16:
17:       // Show player score in white on the top left of the screen
18:       GUI.color = Color.white;
19:       GUI.skin.label.alignment = TextAnchor.MiddleCenter;
20:       GUI.skin.label.fontSize = 40;
21:       GUI.skin.label.fontStyle = FontStyle.Bold;
22:       GUI.Label(new Rect(0,Screen.height/ 4f,Screen.width,70), "Game over");
23:
24:       string message;
25:
26:       // Check player's lives left...if more than 0,
27:       // then player won, otherwise the game was lost
28:       // Generate appropriate final message
29:       if(GameMaster.playerHealth <= 0) {
30:          // The lost message will be shown in red
31:          message = "You lost :(";
32:          GUI.color = Color.red;
33:       } else {
34:          // The won message will be shown in white
35:          message = "You won!!!";
36:          GUI.color = Color.white;
37:       }
38:       // Show lost/won message
39:       GUI.Label(new Rect(0,Screen.height/ 4f + 80f,Screen.width,70), message);
40:
41:       // Last line will be shown in white
42:       GUI.color = Color.white;
43:       GUI.Label(new Rect(0,Screen.height/ 4f + 240f,Screen.width,70), "Press any ke
44:    }
45: }
```

The code adds the "Press any key to continue..." message to the "GameOver" scene. The *Update()* method will load the "MainMenu" after user presses a key.

2. In SourceTree, stage all the changes and commit.

Sound

A few finishing touches - sound and music.

3. In the Unity Editor load the "Level1" scene.

4. Add *Audio | Audio Source* component to the *LevelMaster* game object. Set the *Audio Clip* property of the *Audio Source* component - there are two audio files in *Assets | Sounds*, so you should get two choices. Use the *testloop* asset. Enable *Loop* attribute and set *Volume* to 0.3.

5. Edit the "Attack" script and make the following changes:

Attack.cs

```
using UnityEngine;

public class Attack : MonoBehaviour {

    // Variable storing projectile object
    // prefab
    public Transform shotPrefab;

    // Probability of auto-shoot (0 if
    // no autoshoot)
    public float autoShootProbability;


    // Cooldown time for firing
    public float fireCooldownTime;

    // How much time is left until able to fire again
    float fireCooldownTimeLeft = 0;

    // Variable storing a reference to an audio clip
    public AudioClip shotSound = null;

    // Per every frame...
    void Update () {
        // If still some time left until can fire again
        // reduce the time by the time since last
        // frame
        if(fireCooldownTimeLeft > 0) {
            fireCooldownTimeLeft -= Time.deltaTime;
        }

        // If auto-shoot probability is more than zero...
        if(autoShootProbability > 0) {
            // Generate number a random number between 0 and 1
            float randomSample = Random.Range(0f, 1f);
            // If that random number is less than the
            // probability of shooting, then try to shoot
            if(randomSample < autoShootProbability) {
                Shoot();
            }
        }
    }

    // Method for firing a projectile
    public void Shoot() {
        // Shoot only if the fire cooldown period
        // has expired
        if(fireCooldownTimeLeft <= 0) {
            // Create a projectile object from
            // the shot prefab
            Transform shot = Instantiate(shotPrefab);
            // Set the position of the projectile object
```

```
52:             // to the position of the firing game object
53:             shot.position = transform.position;
54:             // Set time left until next shot
55:             // to the cooldown time
56:             fireCooldownTimeLeft = fireCooldownTime;
57:
58:             // Check if shotSound variable has been set...if yes,
59:             // then play it
60:             if(shotSound != null) {
61:                AudioSource.PlayClipAtPoint(shotSound, transform.position);
62:             }
63:         }
64:     }
65: }
```

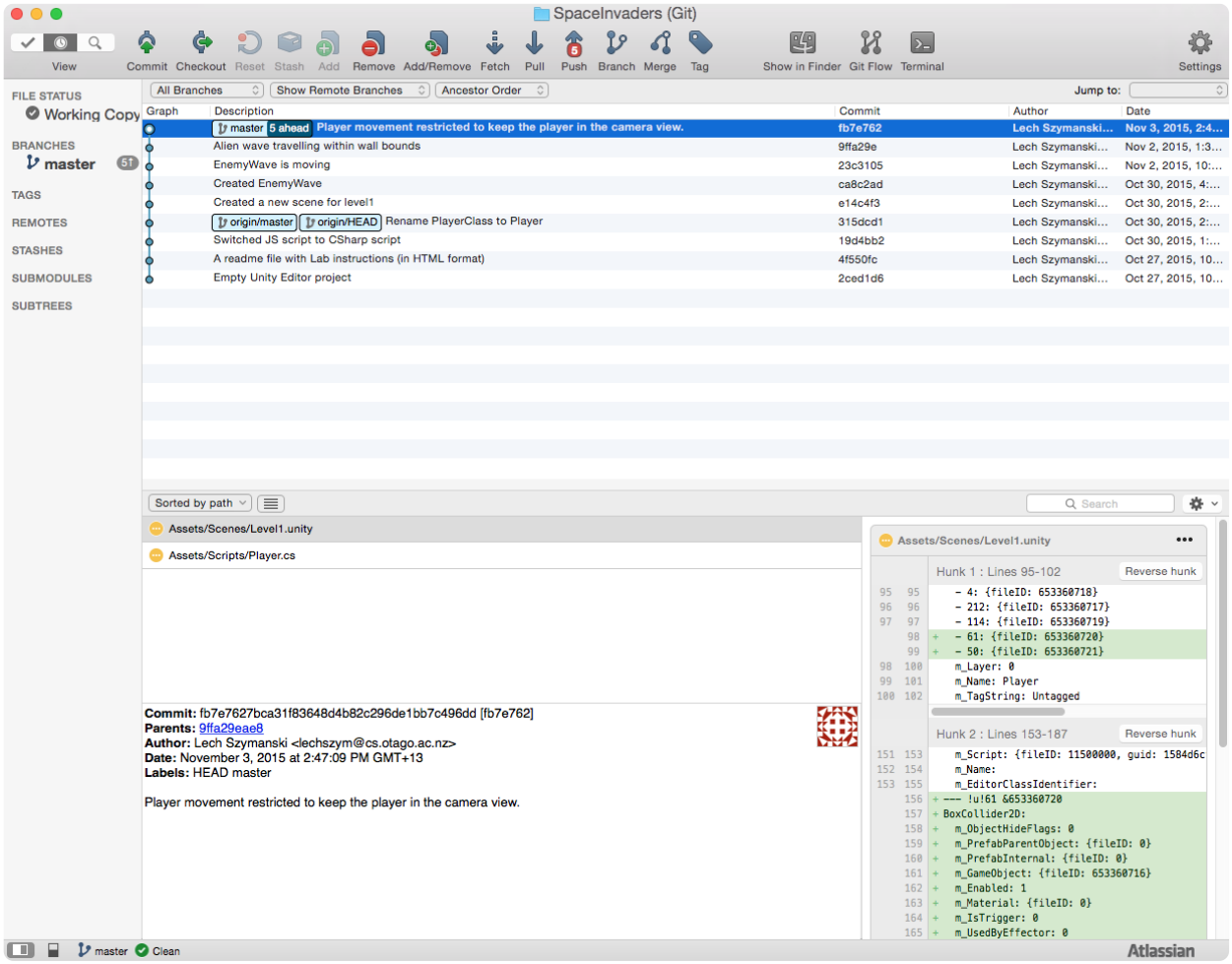
Whenever the *Shoot()* method is invoked and the *shotSound* variable is not null (which it is by default), the referenced audio clip will by played.

6. The shot sounds are meant for the player shots, and not for alien bombs. Select the *Player* game object and find its "Attack" script component in the **Inspector panel**. It should now list *Shot Sound* as one if its attributes (because the new variable has been declared as public). If it doesn't, make sure your "Attack" script has been saved with the changes made above.

7. Drag the *laser* sound from *Assets* | *Sounds* onto the *Shot Sound* property of the *Player* object.

8. That's it - you have music and shot sounds.

9. In SourceTree, stage all the changes and commit.



Pushing your project back to GitLab

10. Did you notice, in SourceTree, that with every commit, a new number is displayed over the "Push" button in the toolbar? This is SourceTree telling you that your local, cloned project is out of sync with the one on on GitHub. Of course it is, you've been making changes to it. Now it's time to sync the projects - in GIT terminology, this is pushing your changes back to the origin.

11. Pushing the changes to a remote server is like publishing your results. The main reason is so that others can see the changes. In this case, the project is private, so no one can see the changes at the moment. However, for Assignment 1 (which is a continuation of this project), you will submit your game exactly this way (detailed instructions can be found in the Assignment description). Another reason to push is so that you can clone the project somewhere else. You can make multiple clones of the project (on different machines), make changes, push them back, and synchronise.

12. In Source tree, press the Push button. Follow the instructions. Once it's done, you can go to your project webpage at "https://altitude.otago.ac.nz/<your user name>/SpaceInvaders", from the left-hand sidebar select "Repository" and then "Commits" to see the history of your commits. Your entire project, with all the snaphsots (if you didn't forget to commit any files or changes) is now on the GitLab server.

Build the final game

The game is finished, what remains is to build the final product. The beauty of Unity is that you can build for many different platforms. Let's build a web version (like the one you've seen at the beginning of this tutorial).

3. Go to *File | Build Settings*.
4. Select the *WebGL* and (if you haven't done this earlier) click the "Switch Platform" button. Remember, for your own game later on, it's a good idea to do this as soon as you create your project, so that Unity is aware of the target platform for your project.
5. If you want to adjust the resolution of the built product, click on *Player Settings* button. Select the *Player* tab and set resolution under the *Resolution and Presentation* section (I've used 800x600).
6. In the *Build Settings* window press the *Build and Run* button. You will need to select the location where you want to final game to go. After that Unity will compile your game. This might take a bit of time.
7. Once the build process is finished, your default browser will be launched playing the new game (security settings of most browser won't run a webGL script from a local file, so Unity creates a local server and makes the game accessible through HTTP on a local port).

Done.