Lab - Unity Basics

Goals

- Familiarise yourself with some of the basic functions of the Unity editor.
- Set up a Unity2D project.
- Create and display a simple scene.

Resources

- Lab01.zip (Lab01.zip).
- Unity Manual (http://docs.unity3d.com/Manual/index.html)

Preparation

- Watch the following video tutorial:
    - Unity editor interface overview (https://learn.unity.com/tutorial/using-the-unity-interface)

What to do

This lab will walk you through the initial set up of a Unity2D project. You're going to create a simple scene, do a bit of scripting to interact with the game engine, and Unity will do the rest.

Log on and open Unity Hub

. Log in to OS X with your CS credentials.

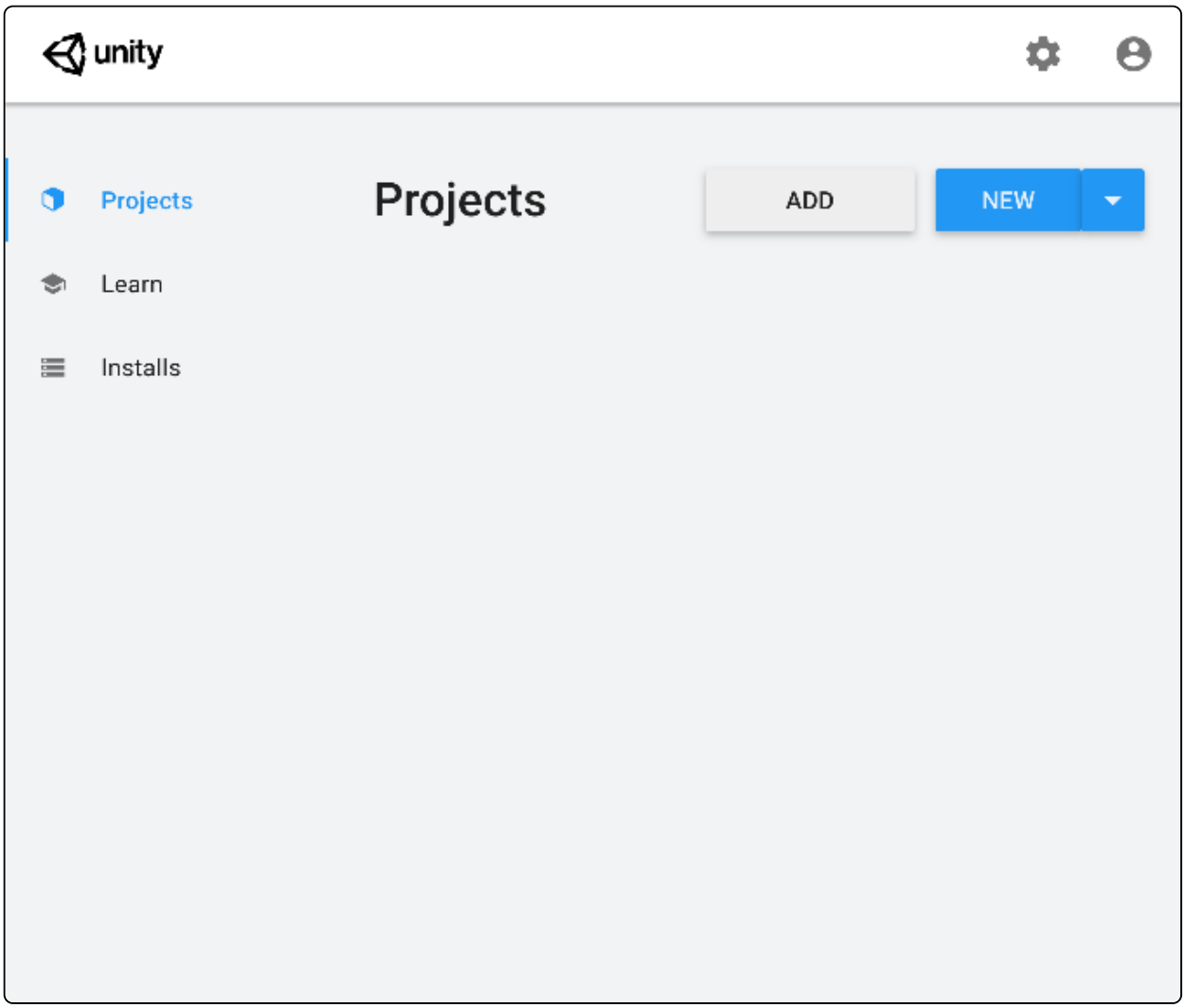
. Run Unity Hub by clicking on the  icon in the Dock or Launchpad.

Unity version

In this paper we're using Unity version 2020.3.23f1 - the latest version at the time of lab setup for this edition of the COSC360 paper. If you are working at home, make sure you use the same version of Unity, otherwise there might be issues with marking and/or collaborating with your group. Unity versions **are not** backwards compatible - if you open a project in a newer version, it will get converted and not work for the old version. Unity Hub makes the management of different projects with different versions quite easy - just makes sure that for anything to do with this paper you use the same version of Unity that we have in the lab.

Create a Unity 2D project

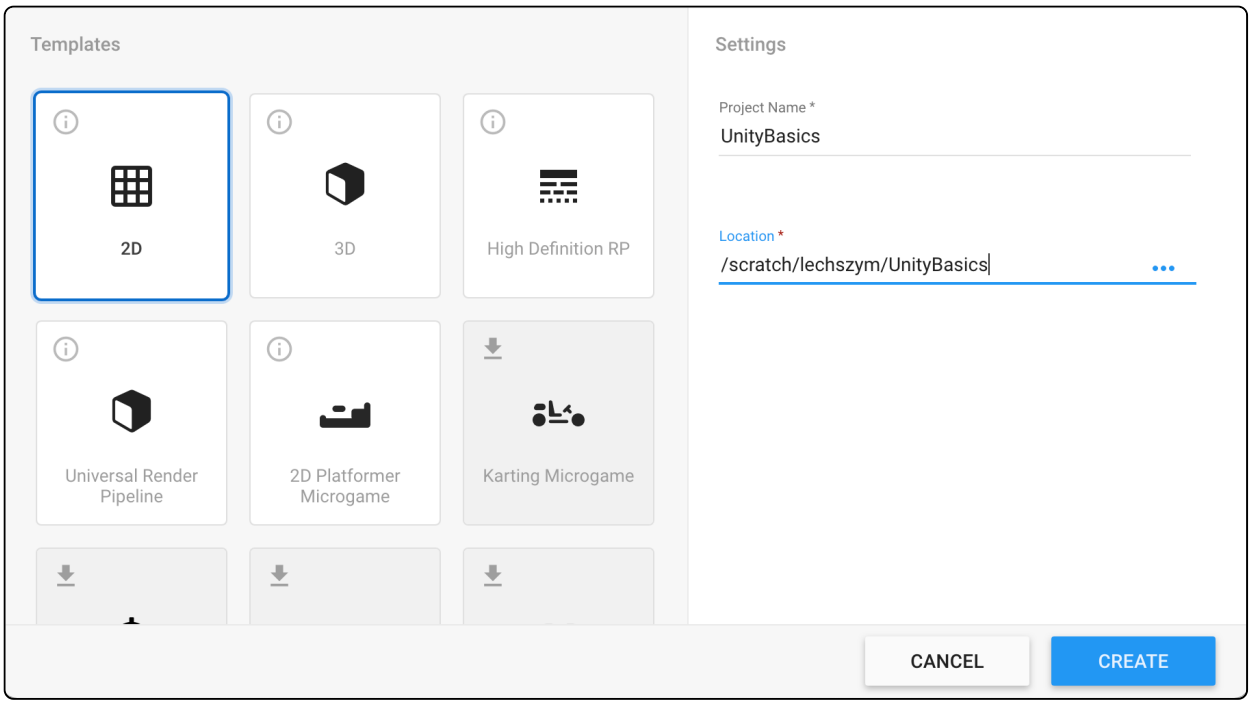
. A window should pop up with a list of recently opened projects (there might be none at this time).



. In the *Projects tab*, click on the *New* button. Set *Project name* to *Lab01* and set the *Location* to where you want to save your project.

Project location (lab machines only)

Unfortunately, the Unity Editor has a problem reading data from our network mounted drives. Don't create Unity projects in your home directory (which is a networked drive). Instead, create a folder with your cs name in the */scratch* directory, which will be your work space on the local hard-drive. For instance, for me the directory would be */scratch/lechszym*. It's probably a good-idea to right-click on that folder, select "Get Info" and change permissions for "Everyone" to "No Access", so that no one else who logs into that machine can read your files. All your Unity projects and your game project should be saved into that directory. Remember, this directory is local, so you won't see it if you login to a different machine.



. Make sure to select the *2D* template - we will be working in 2D mode.

. Press the **Create Project** button - the application will create a new 2D project. There are few panels on screen, which will introduce one by one throughout this exercise.

File flags (lab machines only)

Because your home directory is on a networked drive, caching of files in Unity might be affected and you may see a flood of messages about Failure to change file flags once the project is created. This is a problem due to Unity collecting statistics (which cannot be disabled in the free version of Unity) and the temporary folders for your home directory being located on a networked drive (which Unity doesn't like). A workaround for this problem is to move certain folders from the network drive to the */scratch/<your username>* folder and create symlinks in their original location. The following script: unity_cache_lib_to_scratch.sh (scripts/unity_cache_lib_to_scratch.sh) will do this for you. Download it, and then in the Terminal issue the following commands:

```
$ chmod +x unity_cache_lib_to_scratch.sh
$ ./unity_cache_lib_to_scratch.sh
```

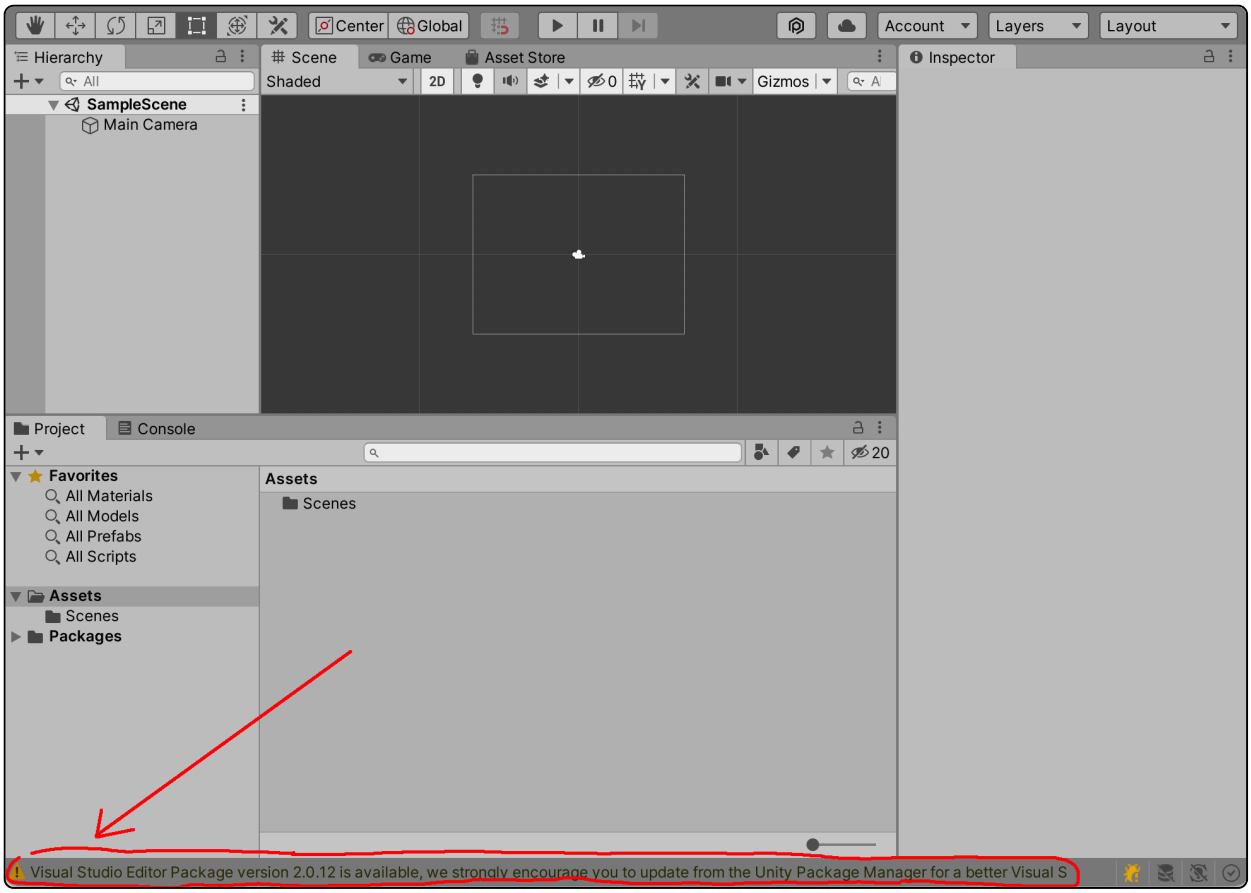
from the folder where the script is located.

Remember that the need to use */scratch/* for your project files as well as the Unity cache ties you to a particular machine. If you need to change machines, you can move the cache files back to it's proper place on home drive using this script: unity_cache_scratch_to_lib.sh (scripts/unity_cache_scratch_to_lib.sh). Then, on the new lab machine, you can run the *unity_cache_lib_to_scratch.sh* again, to get rid of the change flags failures in Unity.

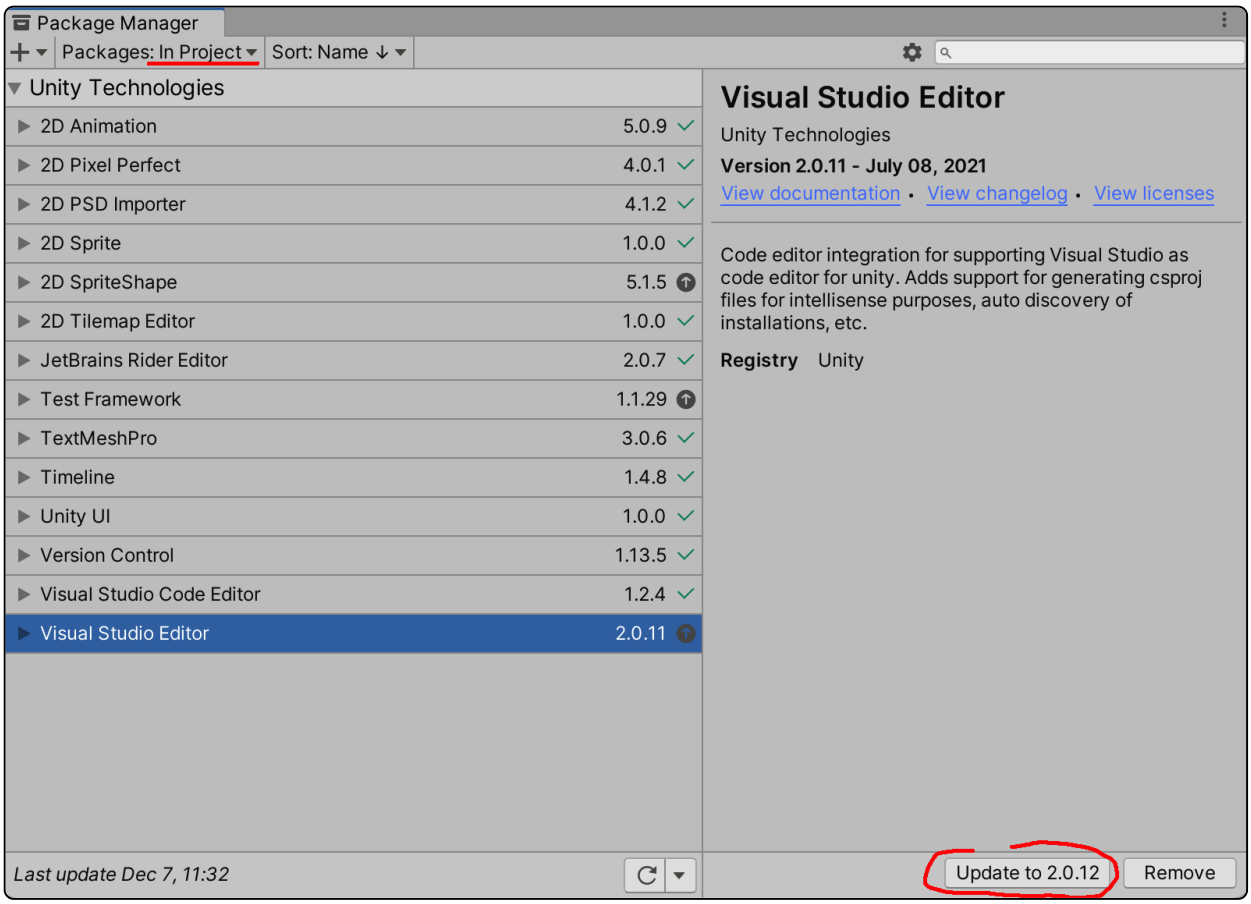
You will need to do this every time you switched to a new computer in the lab.

. There is one more potential thing to take care of in terms of the setup. Take a look at the bottom of the Unity Editor window. Do you see a message about a package upgrade? The Unity game engine is very

modular and allows varios comibations of packages that provide different functionalities (that may or may not be desirable for different games). The result of that is that it contains a package system that seems to undergoe a separate versioning to the main game engine releases. Thus, it seems that one of the packages, that has to do with script editing (that is going to be critical for us) is out of data. It's best to update it.



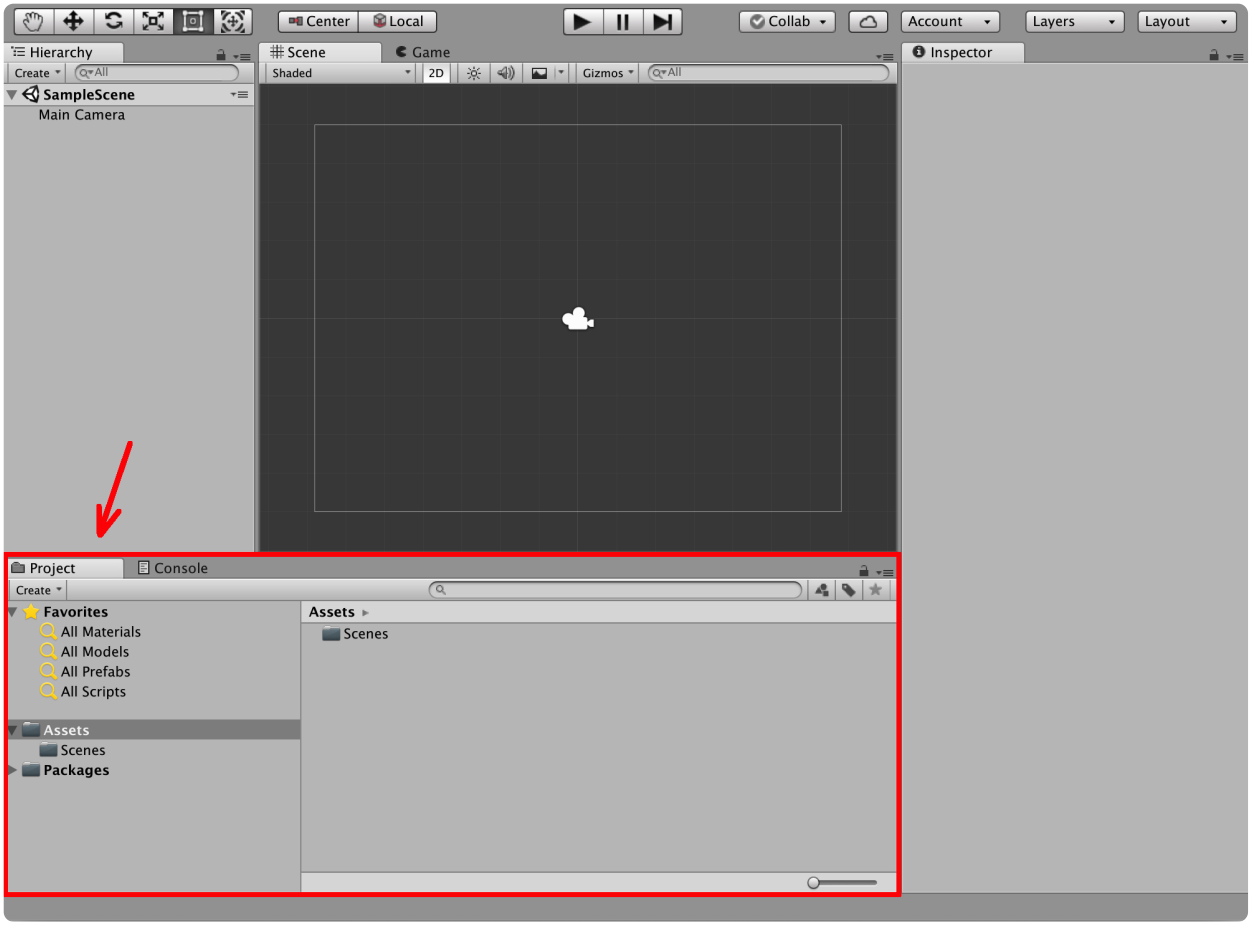
. From the main menu select *Window->Package Manager*. Make sure you're looking at the "In Project" packages and note the "Visual Sutdio Editor" packages on the list. Update it to the latest version...and feel free to update an other packages in that list.



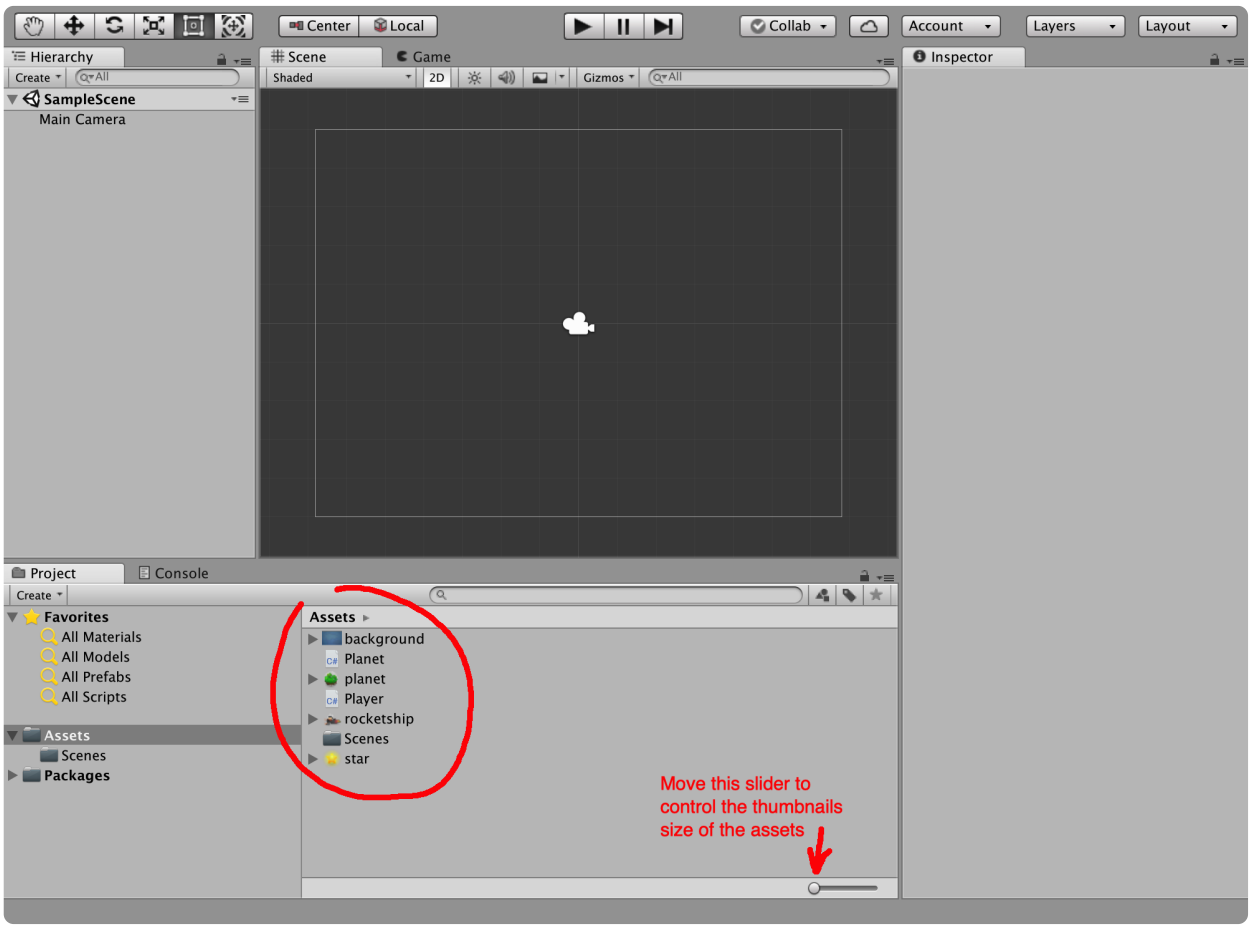
. Close the package manage window. We'll return to the package manager in other labes (when extra packages/functionality will be required) but for now you sould be all set.

Import Assets

). First, take a look at the **Project panel** (outlined in red in the image below). Make sure the *Project* tab is selected. This panel shows a list of your game assets. Currently there's nothing there. Assets are things that make up your game: images, scripts, audio files and such. The Unity authoring tool provides the means of setting up, configuring, and specifying the behaviour of those assets in a scene - but it does not help you with creation of those assets. Hence, the game content (that is, the assets) have to be created in other programs and imported into Unity.

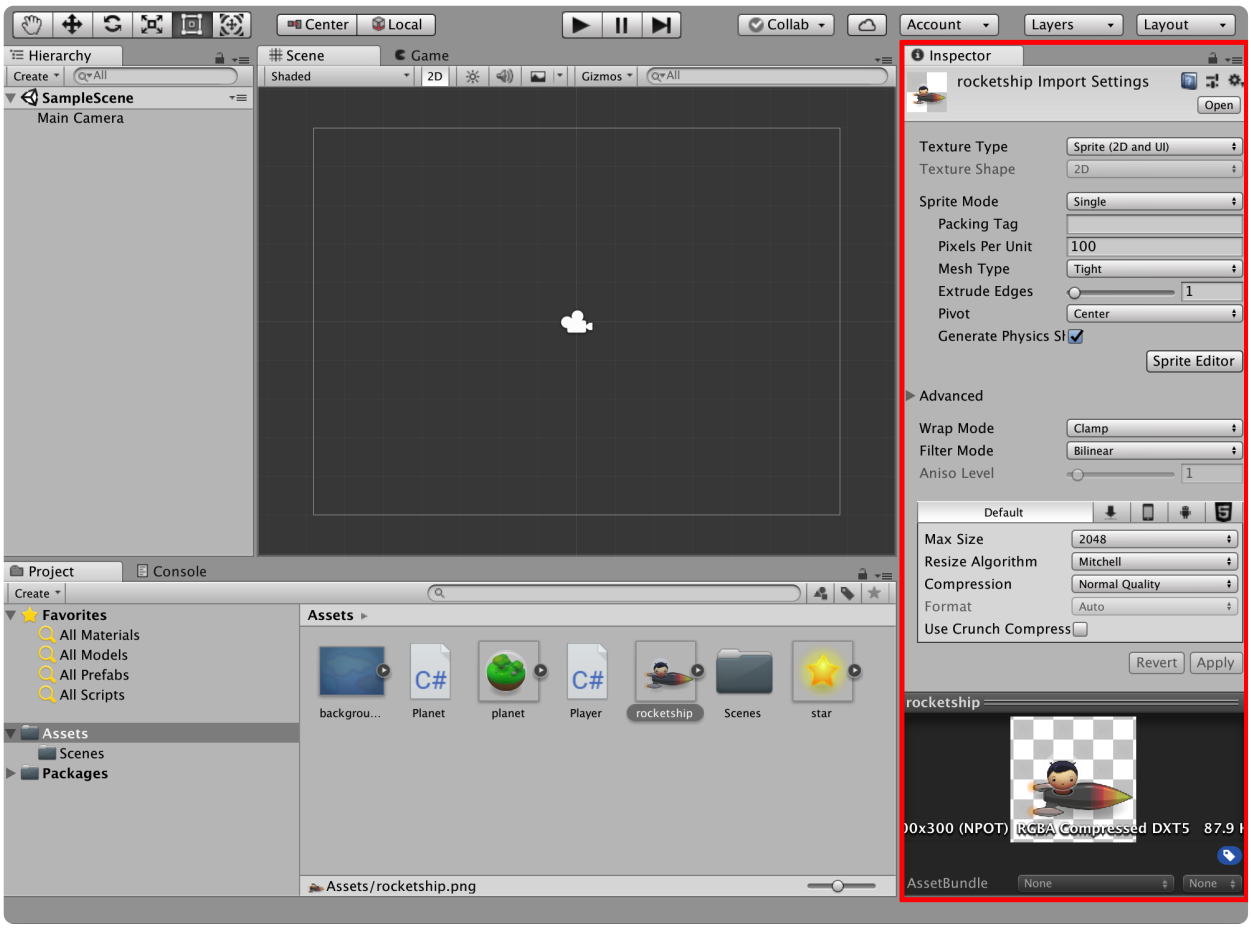


1. In this lab you'll be using pre-made assets. Grab Lab01.zip (Lab01.zip) - it contains four *png* and two *cs* files. Unzip them, select all in the Finder and drag them over the *Assets* window in Unity.



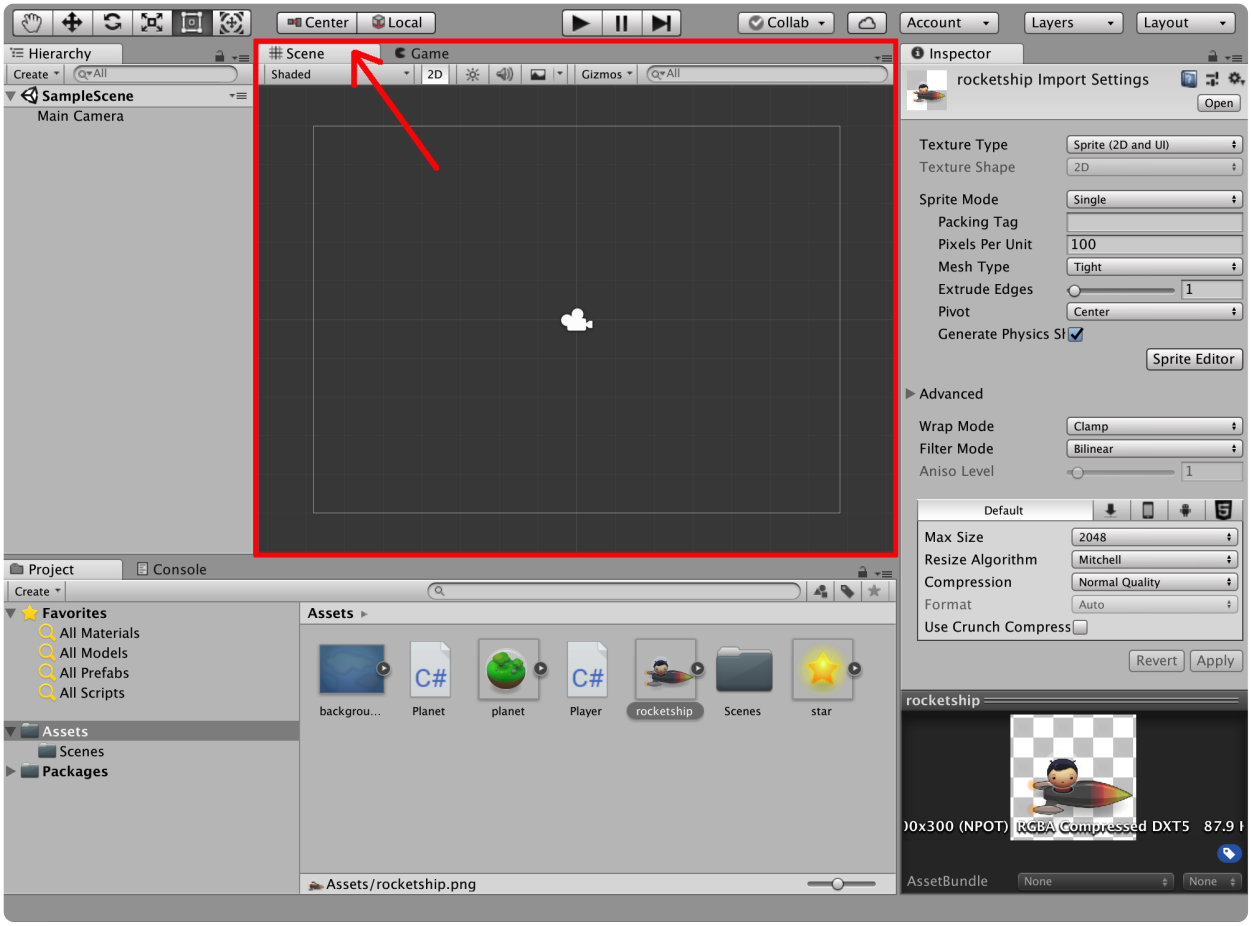
2. In the Finder take a look at the *Labs/Lab01* directory (where your project is saved). There's an *Assets* folder there, where the files you've imported just now should reside.

3. Next, take a look at the **Inspector panel**. This panel is context dependent - it displays, and allows you to configure, properties of a selected object. Select the *rocketship* image from the *Assets* window. The **Inspector panel** should display information about the image. Note that *Texture Type* property says "Sprite". Sprite (https://docs.unity3d.com/Manual/Sprites.html) is computer graphics term for a 2D image that is integrated into the scene - how and where exactly it is displayed, depends on the setup of your scene.

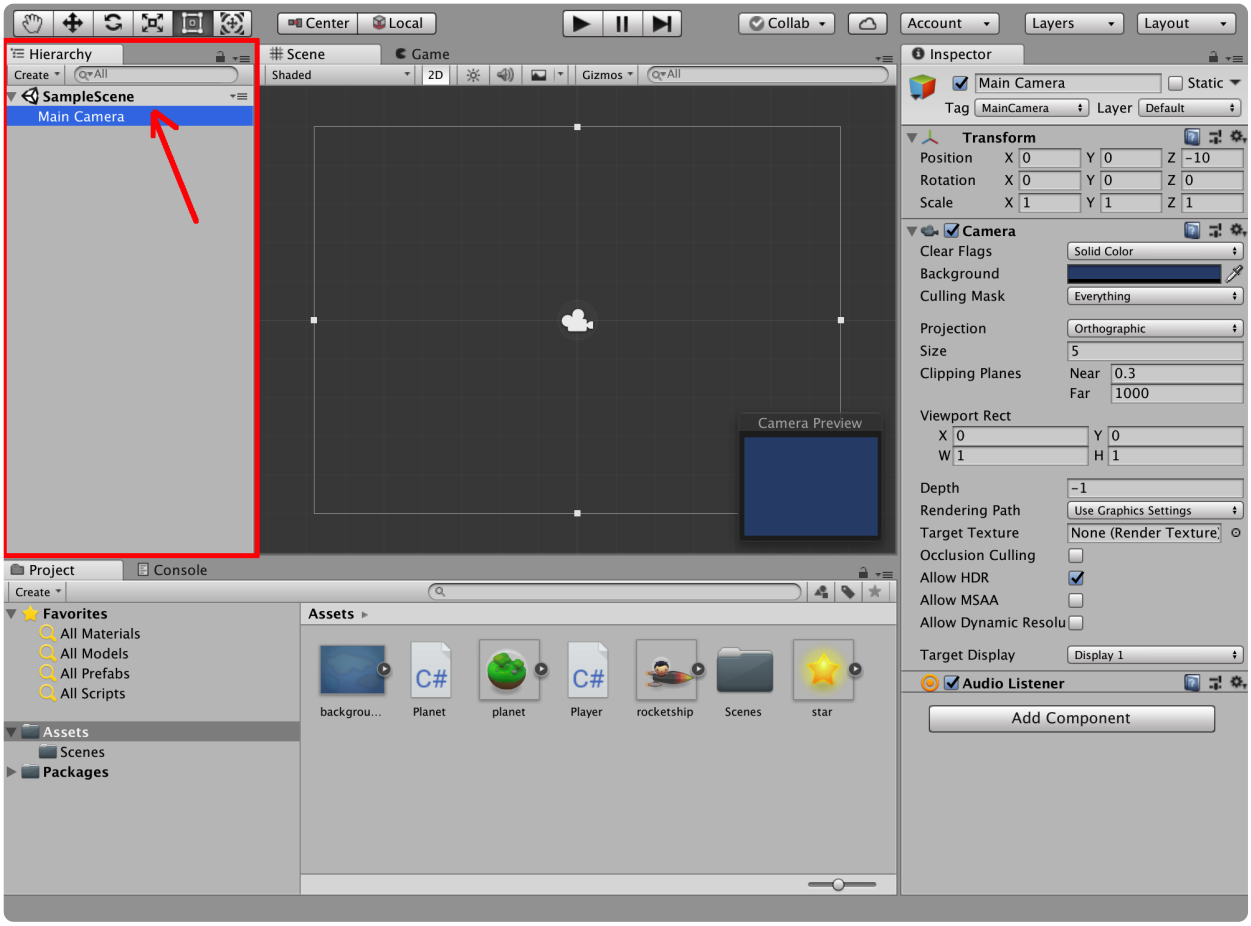


Place game objects in the scene

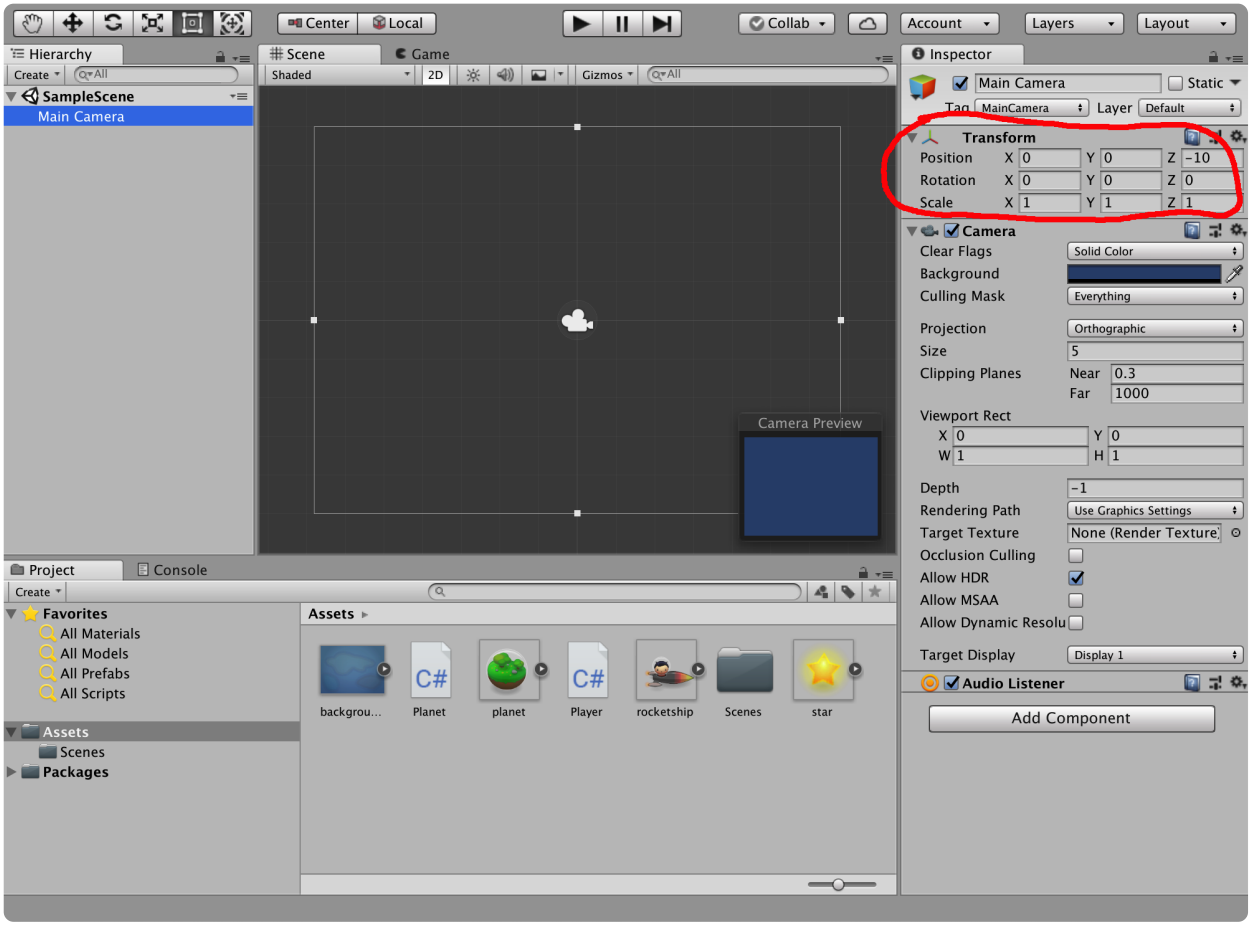
1. The scene is where game objects (http://docs.unity3d.com/Manual/class-GameObject.html) are placed and animated. It's a simulated virtual world, which the player gets to see while playing the game. The player will only see a view of the scene from a virtual camera (https://docs.unity3d.com/Manual/class-Camera.html), which you, the game maker, can configure to display different parts of the scene. At this point the **Scene panel** should display a white camera icon - if it doesn't, make sure *Scene view* is selected. Currently there's only one game object in the scene – the *Main Camera*.



; All the game objects placed in the scene are listed in the **Hierarchy panel** (outlined in red in the image below). At this time there should be only one object, called *Main Camera*, there. Select it. In the Scene panel you should now see the outline of the region visible to the camera and the arrows indicating the axis in the camera coordinate system. If you don't see the camera outline, double-click on *Main Camera* - this should zoom out the scene to show the entire game object. You also should see a smaller *Camera Preview* window in the *Scene*, which gives you a preview of camera's view – right now just a solid background.



3. Take a look at the **Inspector panel**, which now should display the properties of the camera. Every game object (that's been placed in the scene) has a *Transform* (http://docs.unity3d.com/Manual/class-Transform.html) component, which specifies the location, rotation, and scale of that object. The *Position* attribute of the *Transform* component specifies object's location in "world units" along the X, Y and Z axis with respect to "world origin". Note that the *Main Camera* is located at (0,0,-10),that is: 0 units from the "world origin" on X axis, 0 units from the "world origin" on the Y axis, and -10 units from the "world origin" on the Z axis.



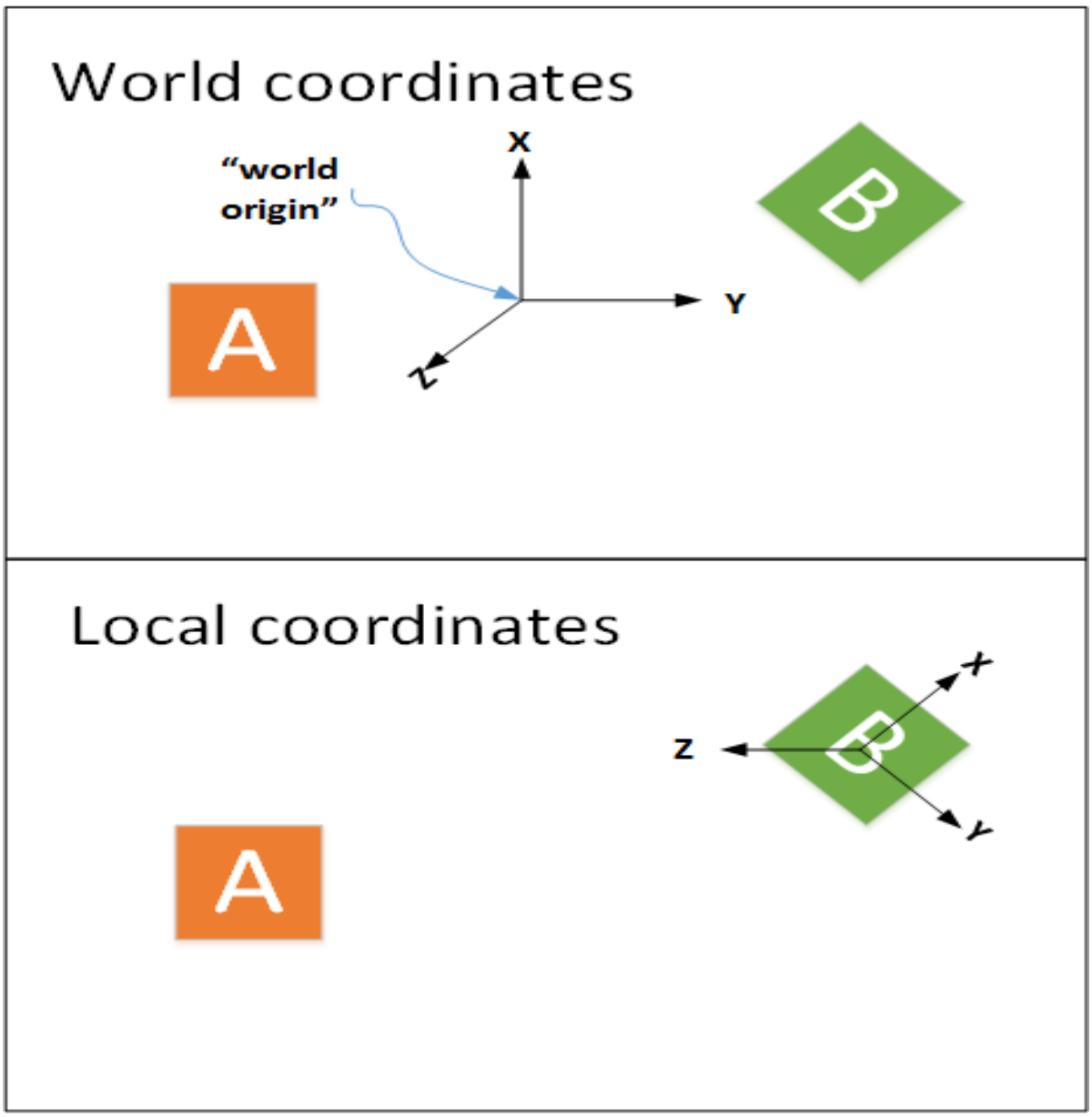
About coordinate systems

There are a number of coordinate systems in Unity - depending on the task, different coordinate systems are easier to work with. At this point let's just worry about the main two coordinate systems.

World coordinates give the location in the virtual 3D game space with respect to the "world origin", the point (0,0,0). These coordinates give an absolute position of an object in the game space.

Local coordinates give a relative position of an object with respect to another object, usually the parent object.

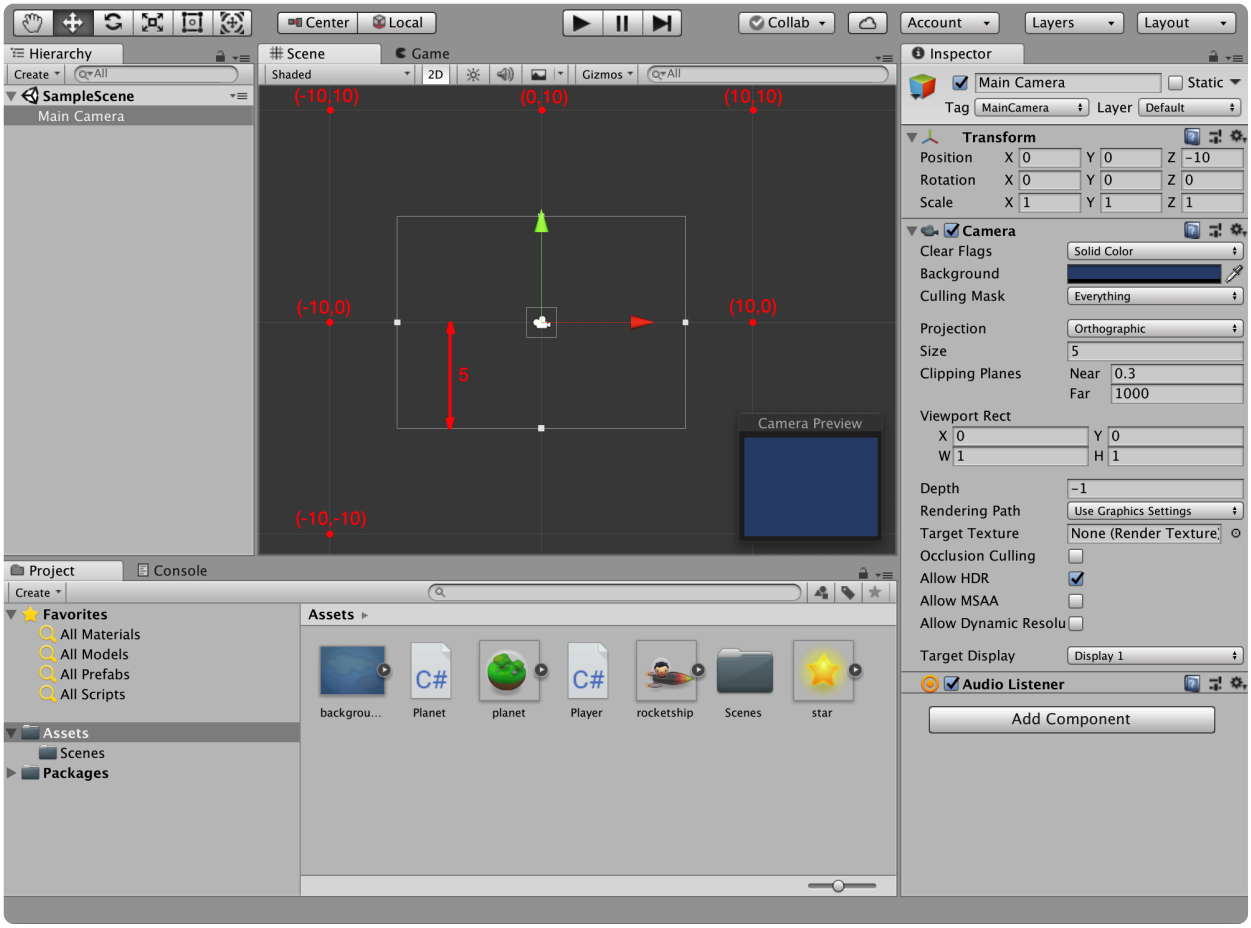
The figure below demonstrates how spatial relationship between two objects, A and B, can be defined in two different coordinate systems. In the "world coordinates" the position of A and the position of B are given with respect to the "world origin". In the second scenario the position of A can be specified with respect to B's local coordinate system.



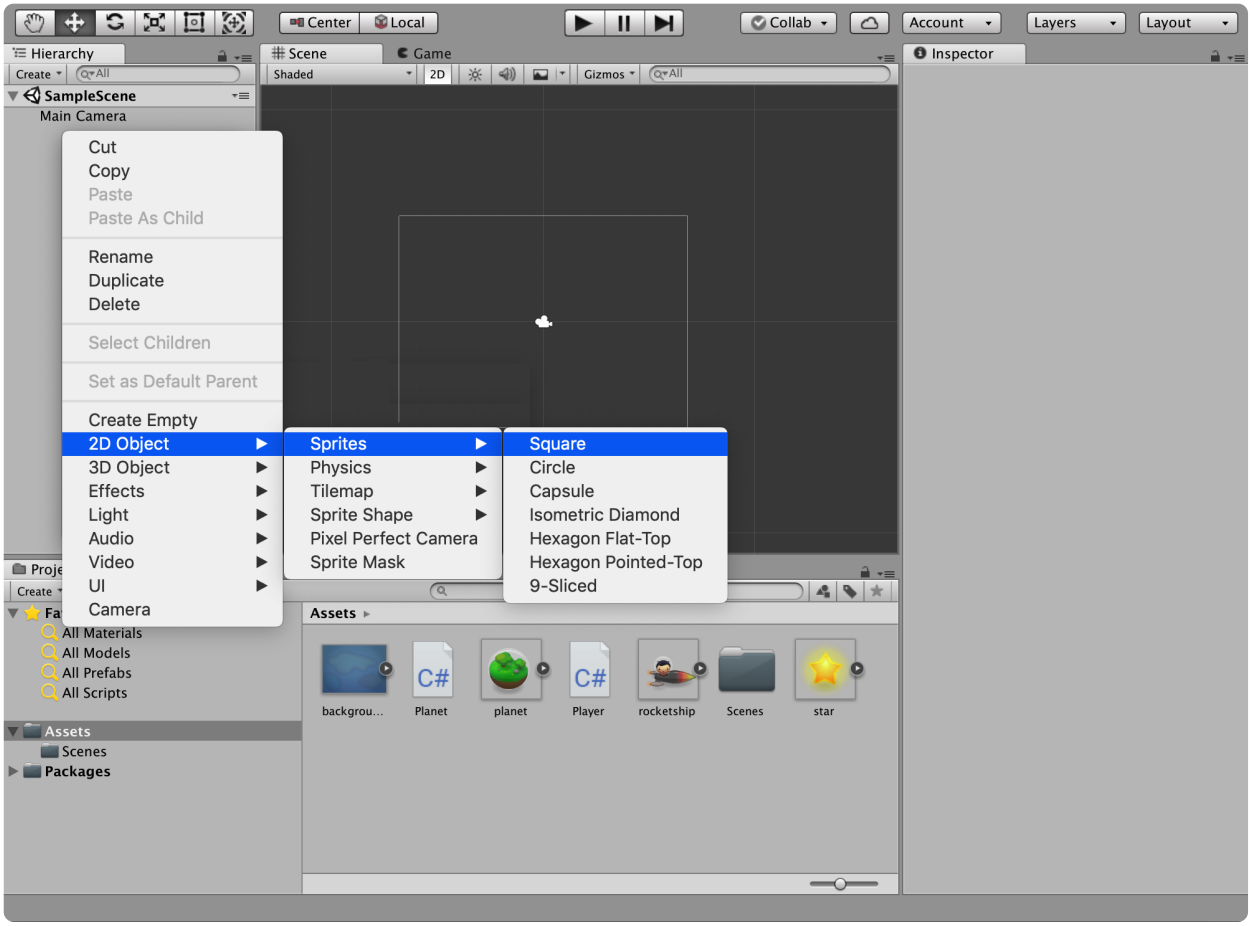
You might be wondering why you need 3D coordinates, that is (X,Y,Z), for location when making a 2D game. The main reason is that Unity is really a 3D engine, 2D mode being just a special engine configuration. But there are advantages to having a third coordinate. Did you notice the arrows originating from the camera icon in the *Scene view* (when the *Main Camera* is selected in the **Hierarchy panel**)? These show the local coordinates with respect to the camera - camera local coordinates are kind of important, since they define what the player sees. The green arrow (pointing up) is the Y axis, the red arrow (pointing right) is the X axis...and there's also Z axis - it's the blue circle, right over the camera icon, representing a blue arrow pointing away from you (into the screen). Because the *Main Camera* doesn't have any rotation about the world axis (see the "Transform" component) its local coordinates system is perfectly lined up with the world coordinate system (though *Main Camera*'s centre is shifted 10 units on the Z axis). When the game engine renders a scene in a 2D project, it flattens everything along the camera's Z-axis...and so all the physics and collisions are done in the XY plane. However, the Z-axis is still useful for specifying the rendering order - which object are in front, and which in the back, of the scene.

The size of the *Main Camera* is defined by the *Camera | Size* attribute, which specifies the height of the camera from it's centre. Currently it's set to 5, which means that camera is 10 units high (5 going up, and 5 going down from its centre). The width is relative to height and it follows the desired screen ratio. The

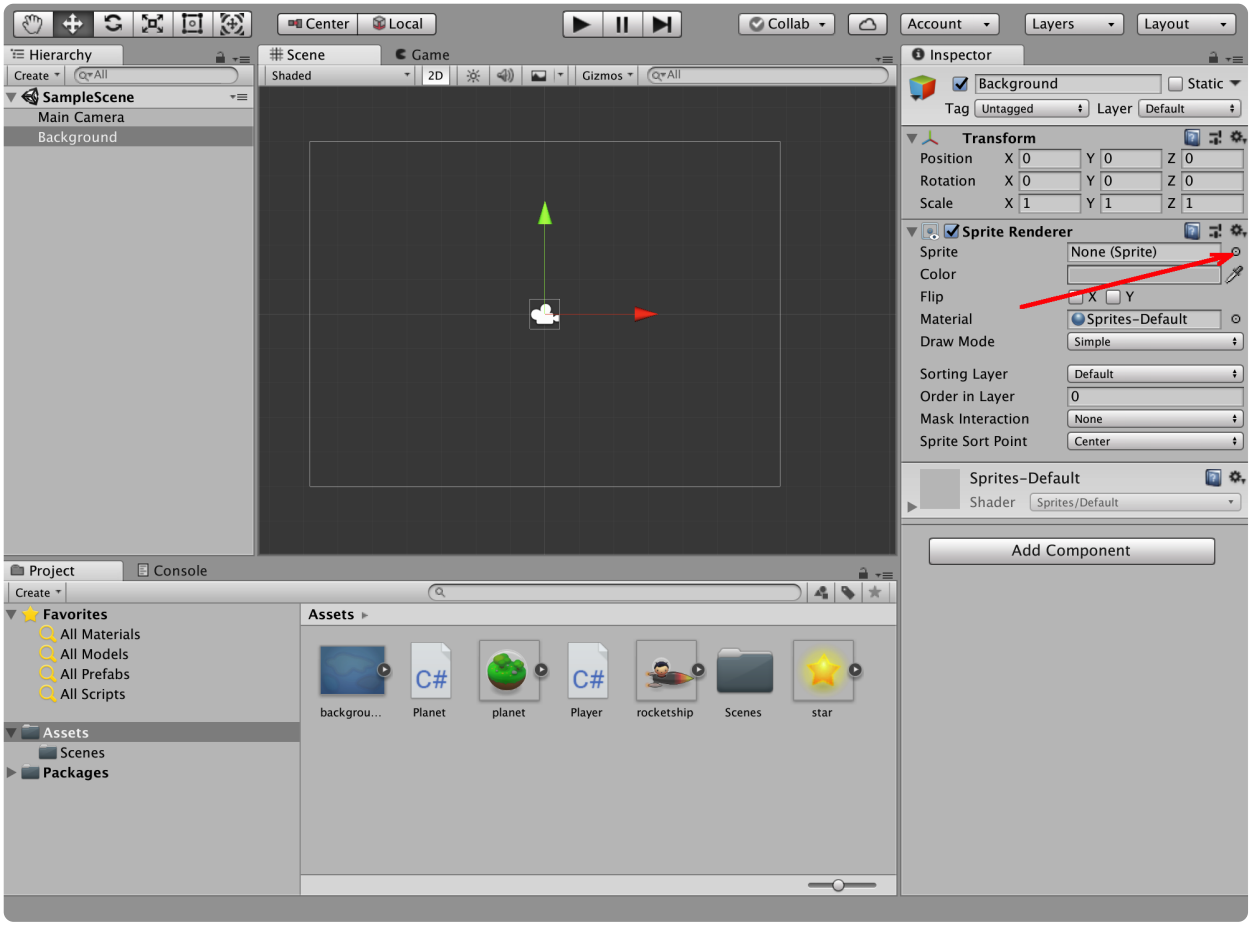
screenshot below shows the size and position of the camera in the scene in "world units".

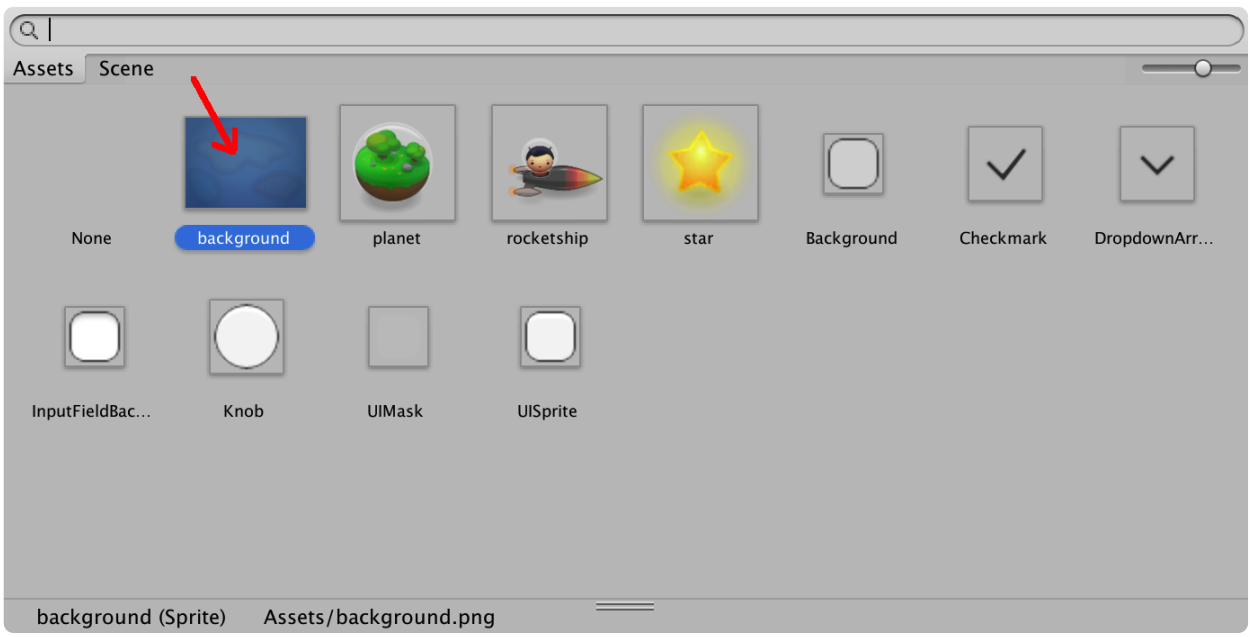


1. Time to place your first game object in the scene. In the **Hierarchy panel** right-click empty space and select a *2D Object* | *Sprites* | *Square*. The new object will appear in the panel as a *New Sprite*. Click it once rename it to *Background*.

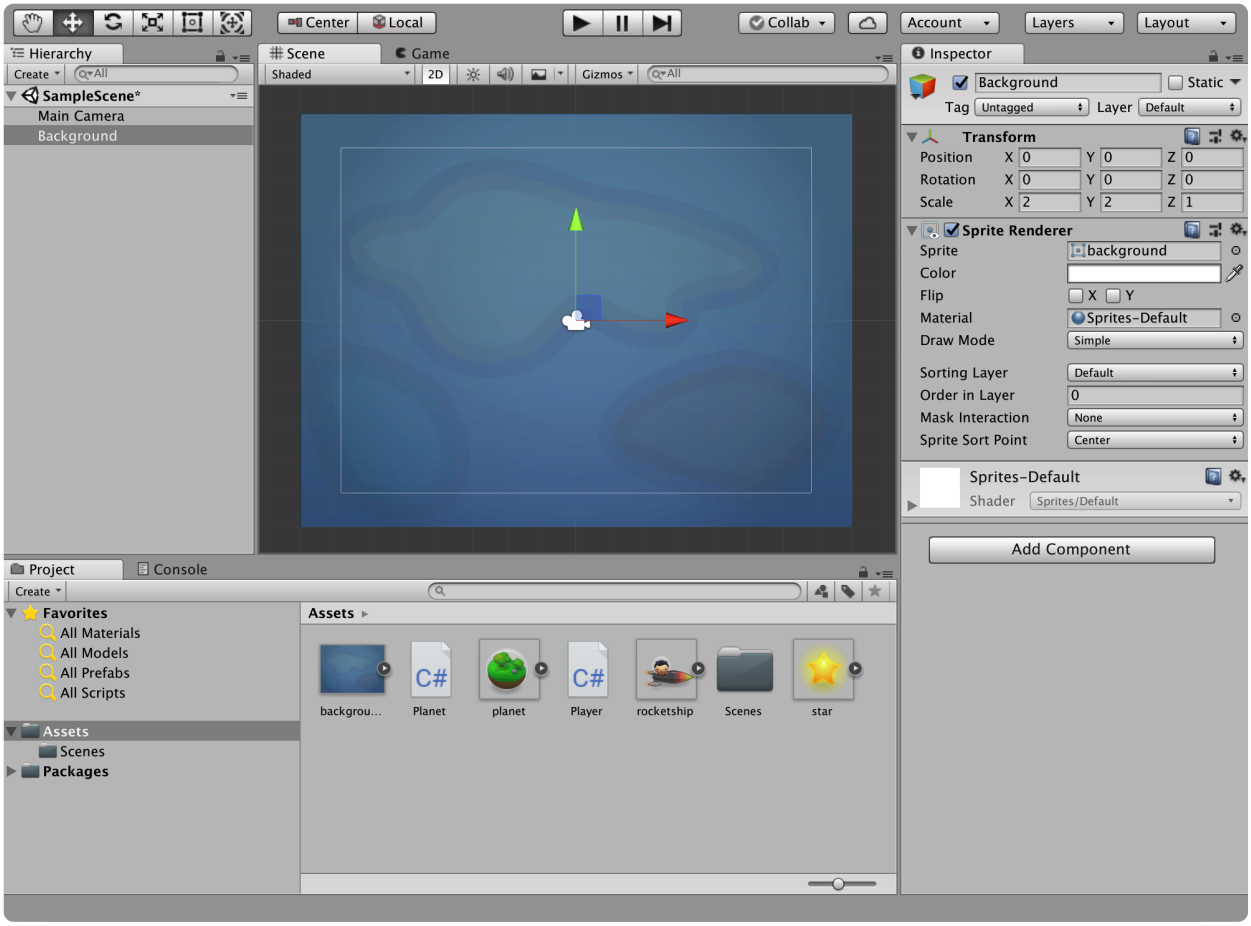


3. Select the newly created *Background* game object in the **Hierarchy panel**. Check its *Transform* component in the **Inspector panel** – it should be positioned at (0,0,0). Note the *Sprite Renderer* component. Click on the circle next to the box which says "None (Sprite)". A dialog should appear listing your image game assets - select the *background* image.

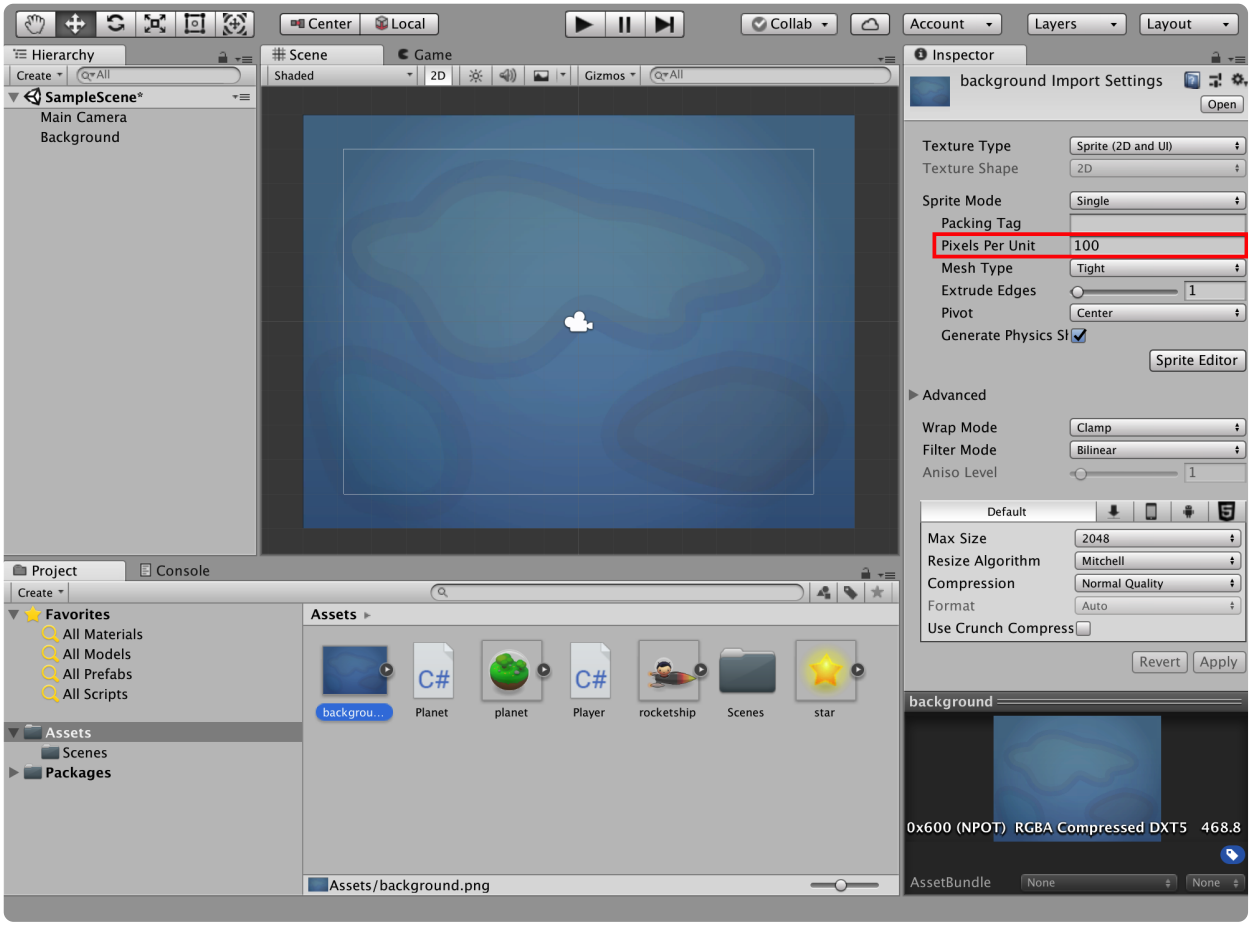




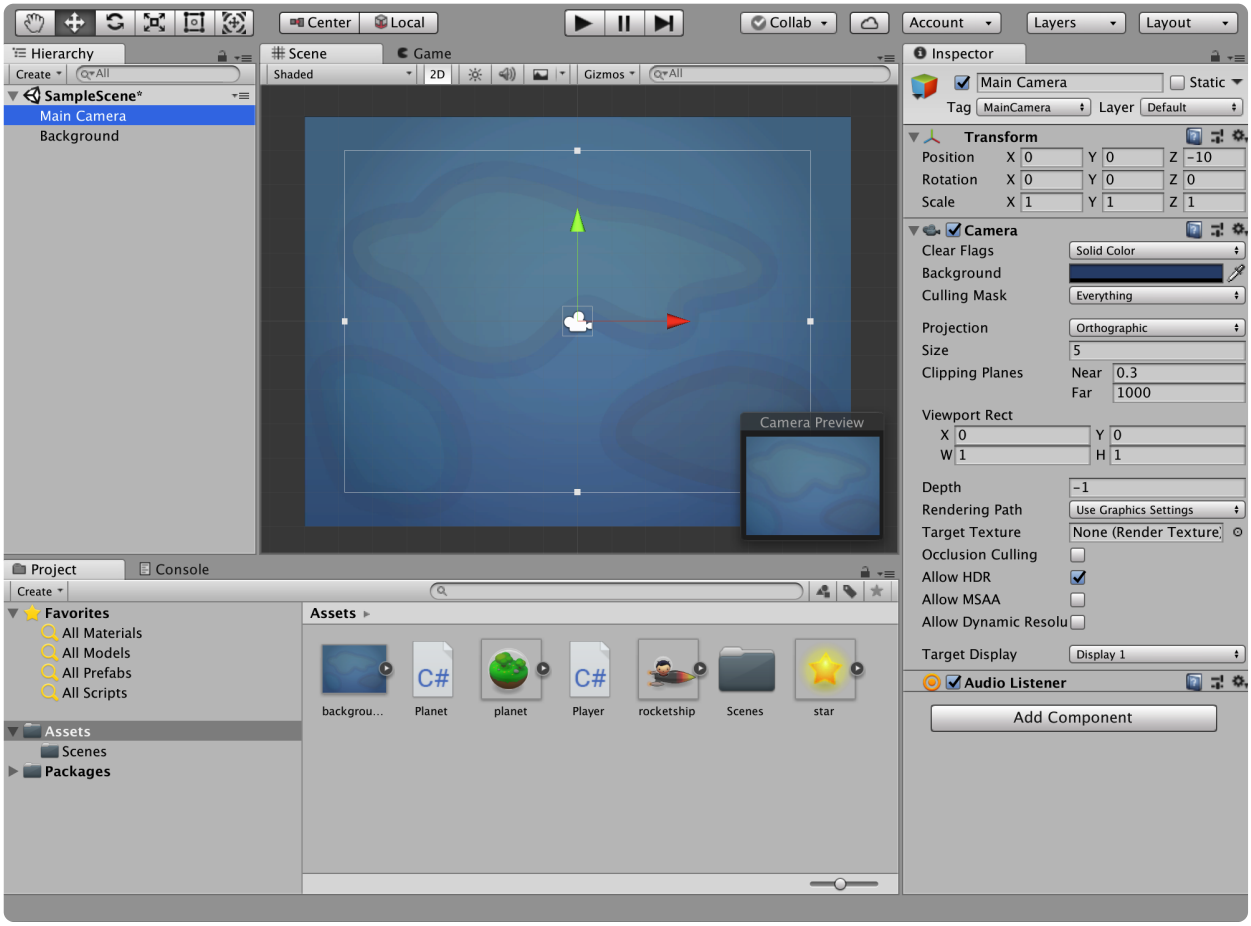
). Set the *Transform* properties of the newly created object as follows: *Position* to (0,0,0) and *Scale* to (2,2,1).



). The scaling factor is now doubling the sprite in the scene along the horizontal and the vertical directions. What's the reason for doing this? If you click on the *background* image asset (in the *Assets* window) you'll notice in the **Inspector panel** the *Pixel to Units* setting. Right now it is set to 100. This specifies the translation of the image size from pixels to size in "world units". Since this particular image is 800x600 pixels, it ends up measuring 8x6 "world units" in the **Scene view** at the scale of 1.

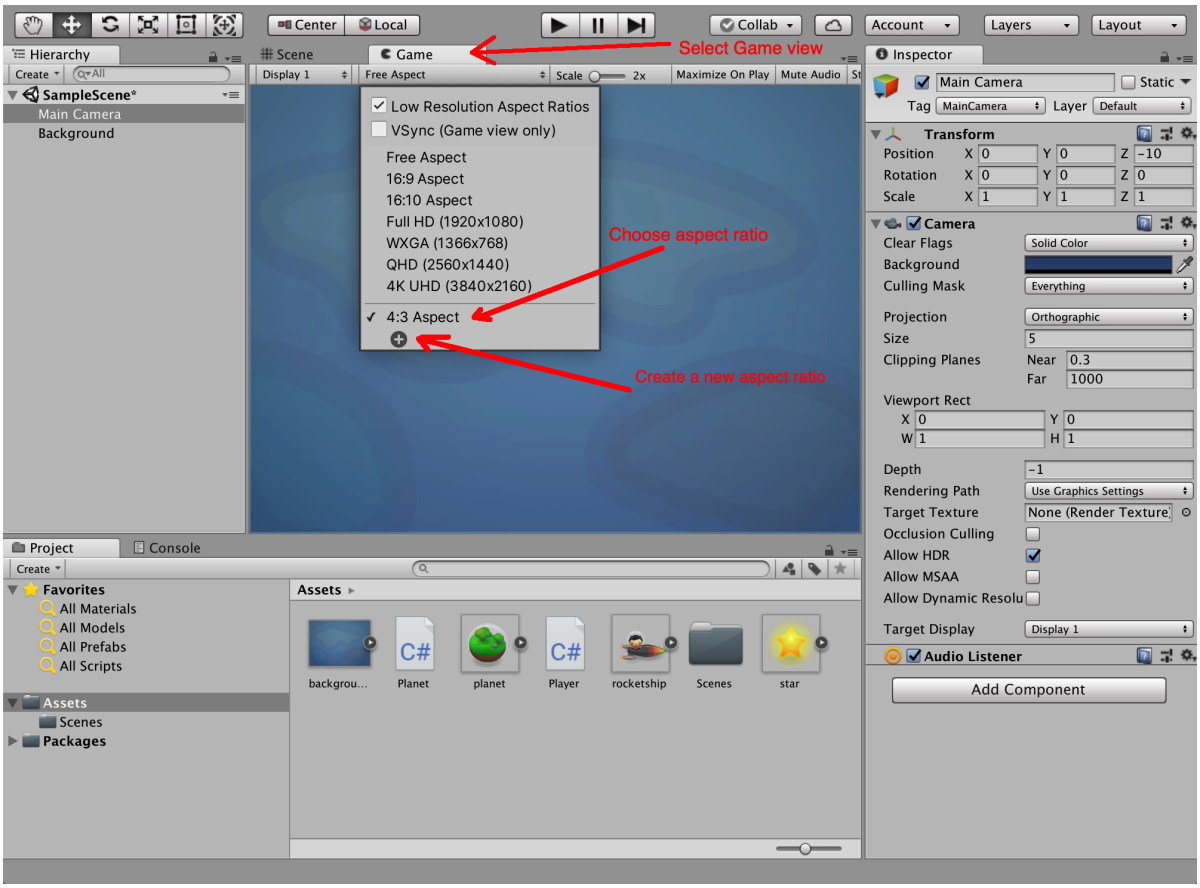


Located at the "world origin" this makes the background stretch from -4 to 4 "world units" on the X axis and -3 to 3 "world units" on the Y axis. Recall that *Main Camera* stretches from -5 to 5 "world units" on Y axis. In order for it to fill the camera's entire view, the background sprite needs to be bigger. One way would be to change the "Pixel to Units" setting, but that setting would affect the scaling of the image in every object that uses it as a sprite (a given asset can be used by multiple objects in the game). It's better to leave the *Pixel to Units* setting as is, and scale the particular game object that you are working with. After doubling the X and Y scale, the sprite will stretch from -8 to 8 "world units" on the X axis, and -6 to 6 "world units" on the Y axis. The figure below shows the *Main Camera* outline over the scaled background.

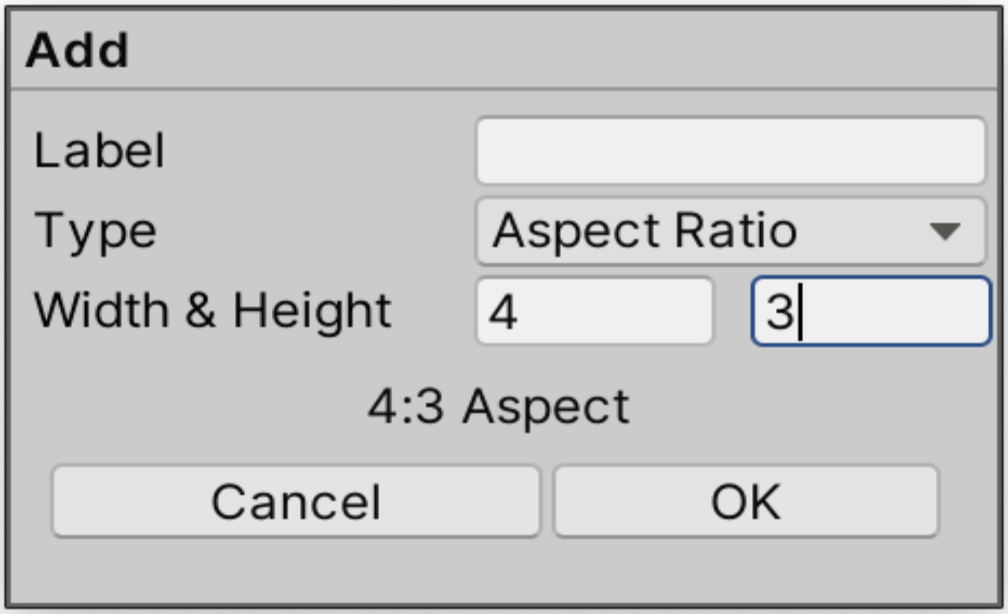


If your camera outline doesn't fit within the background image (like in the screenshot above), it because your camera aspect might be set differently to mine. You have two choices. You can keep your camera aspect and scale the Background game object a bit more to fill out the entire camera view. Alternately, you can change your camera aspect ratio.

To change the camera aspect, select the **Game view** and change the camera aspect tor 4:3, as shown in the screenshot below (if you don't see the 4:3 option follow the instructions below):

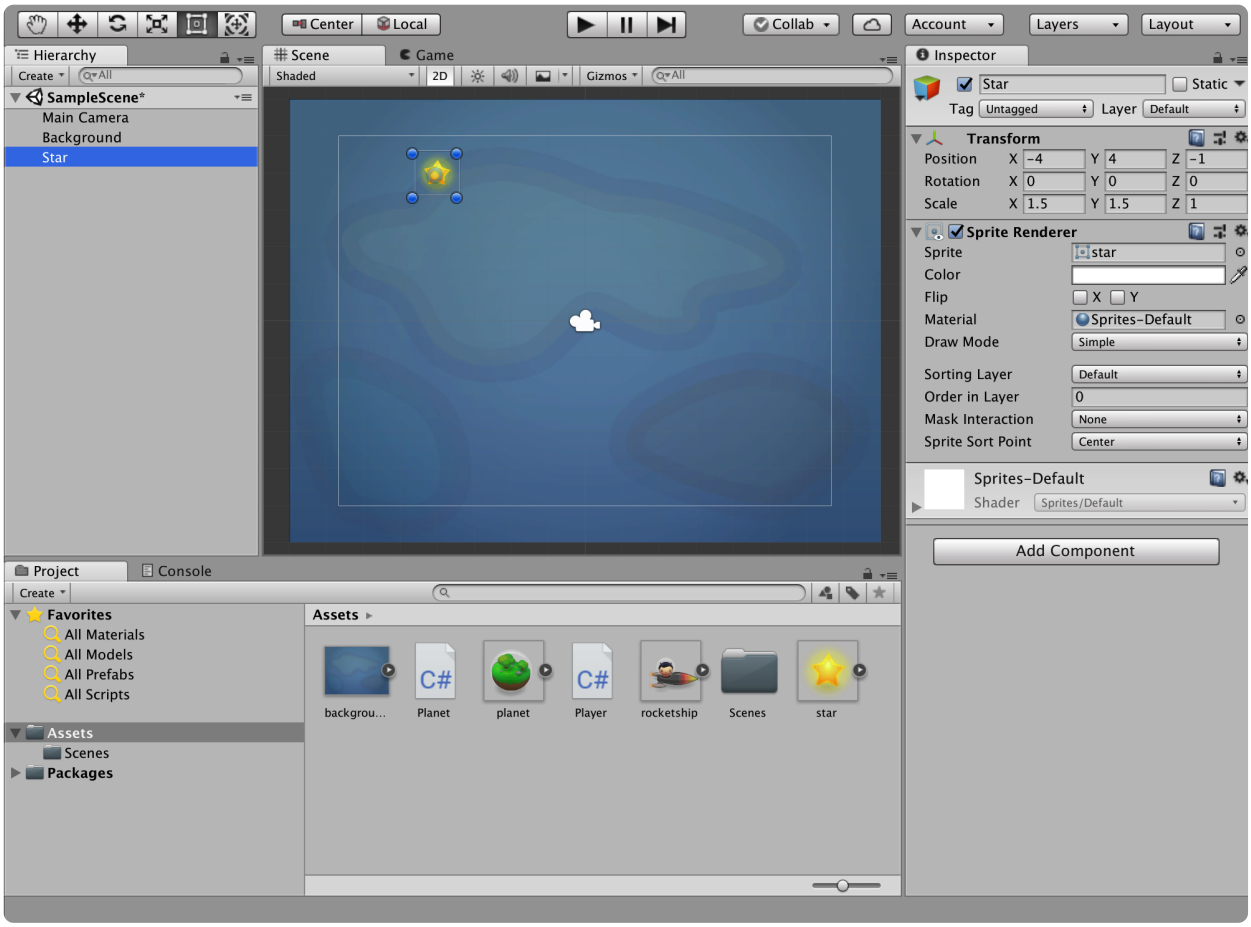


If you don't see an option for 4:3 aspect ratio, you need to create one first, by clicking on the plus icon (as shown in the screenshot above) and creating the new aspect ration as shown in the screenshot below:



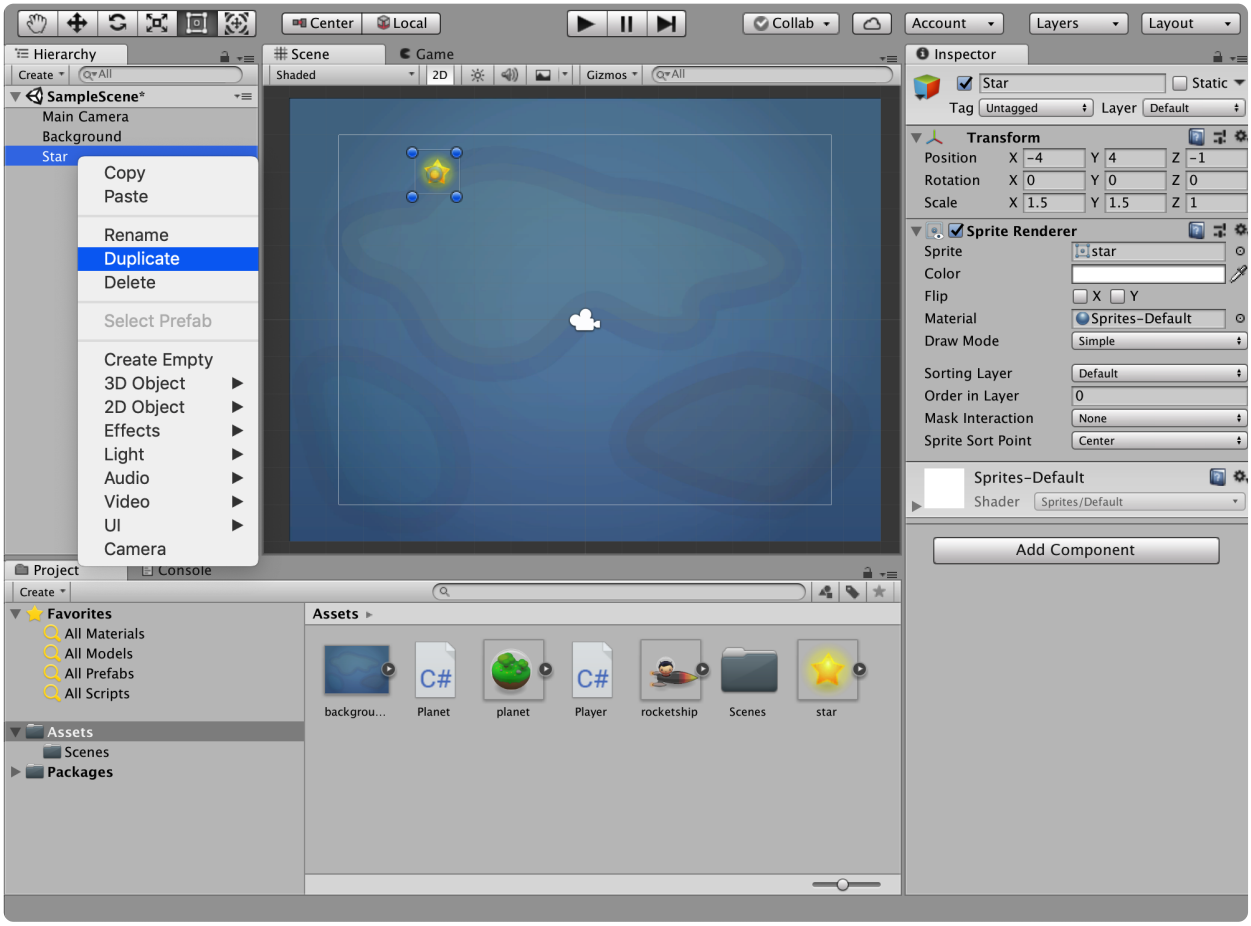
When you're done go back to **Scene view** and check if the camera outline now fits within the background.

1. Create a new *2D Object | Sprite* in the **Hierarchy panel**. Name it *Star*. Set its *Sprite Renderer | Sprite* to the "star" image. Set the *Transform* attributes as follows: *Position* to (-4,4,-1), and *Scale* to (1.5,1.5,1).



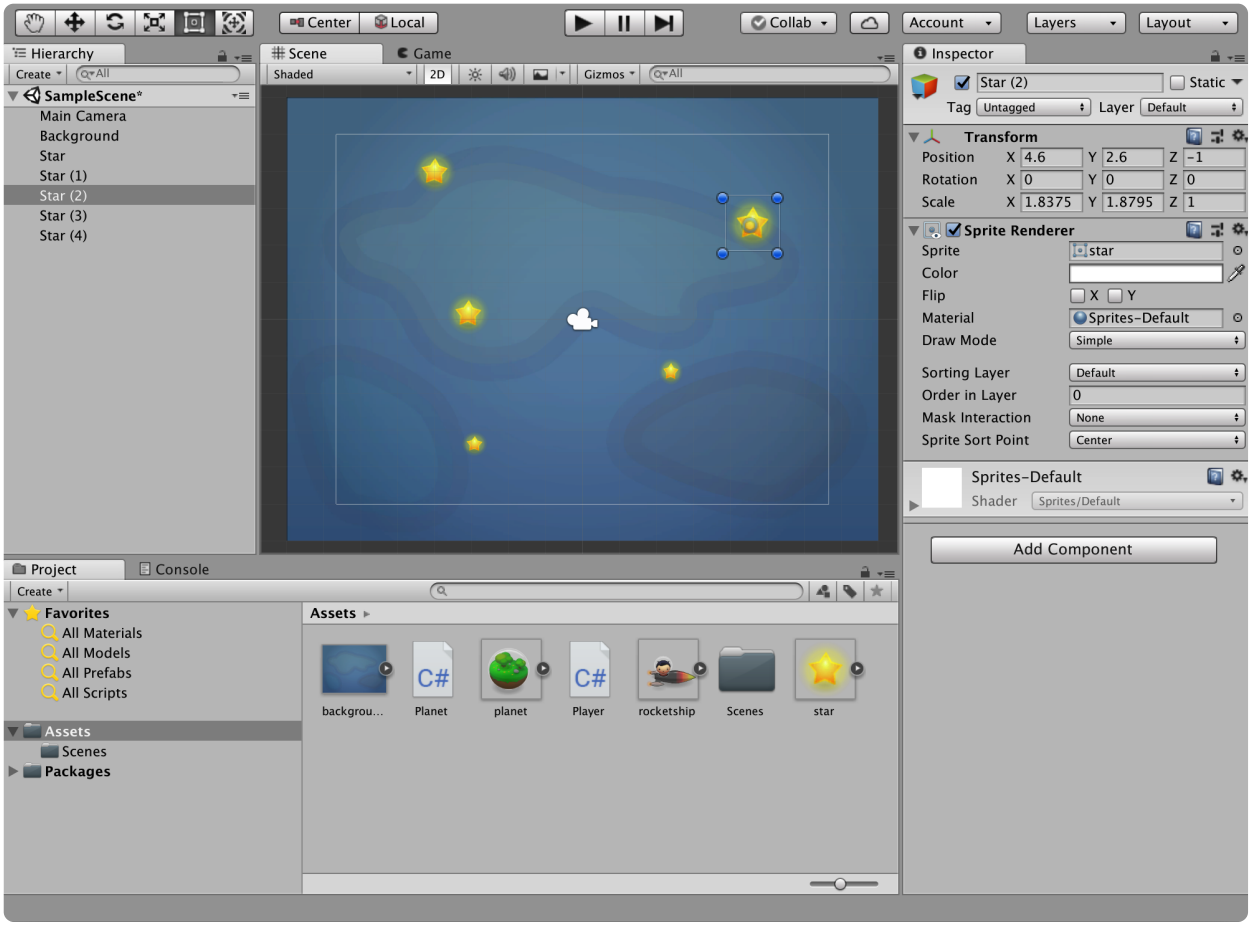
Recall that the *Main Camera*'s Z-coordinate was set to -10, and the *Background*'s to 0. You have just set the *Star*'s Z position to -1. The figure below shows the position order of these game objects along the Z-axis. The *Main Camera* (as shown in the image) is looking in the positive direction of the Z-axis, and so the *Star* will be rendered in front of the *Background*. When it comes to collisions and physics, the Z position does not matter in 2D games (it's just ignored). However, it does affect the rendering order. If it's helpful, you can think of the Z-dimension as depth from the camera, but only for drawing. There are other ways of selecting rendering order of sprites (such as using the *Order in Layer* property of the *Sprite Renderer* component), but for now stick with the Z-axis method.

2. Time to put more stars in the scene. Right-click on the *Star* game object in the **Hierarchy panel** and select *Duplicate*. A copy of the game object, named *Star (1)* will be created in the scene.



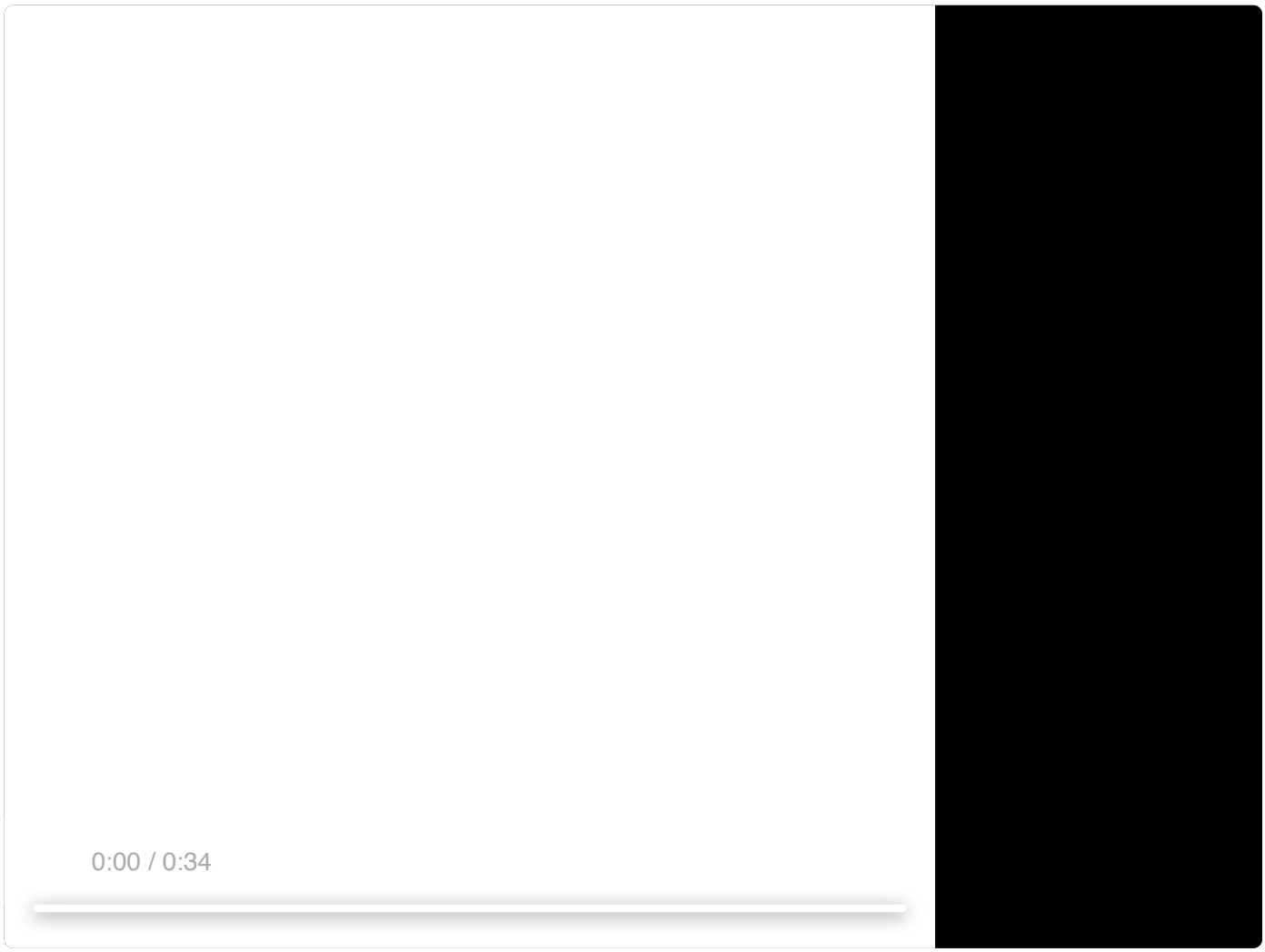
3. Select the **Translate control button**  (keyboard shortcut W) from the "Scene view" control (http://docs.unity3d.com/Manual/PositioningGameObjects.html) toolbar, and click on the *Star (1)* game object in the **Hierarchy panel**. You should see a rectangular box around the selected game object in the *Scene view* (the two stars are probably right on top of each other right now). Drag that rectangle to some other part of the scene - the Z-coordinate of the object's position will remain unchanged. If you prefer, you can also change the position of one of the *Stars* by changing its *Transition* | *Position* coordinates in the **Inspector panel**.

4. Duplicate a few more stars and place them anywhere you wish in the scene. To make it look less generic you can scale the stars differently (or set different *Transform* | *Rotation* around the Z-axis).

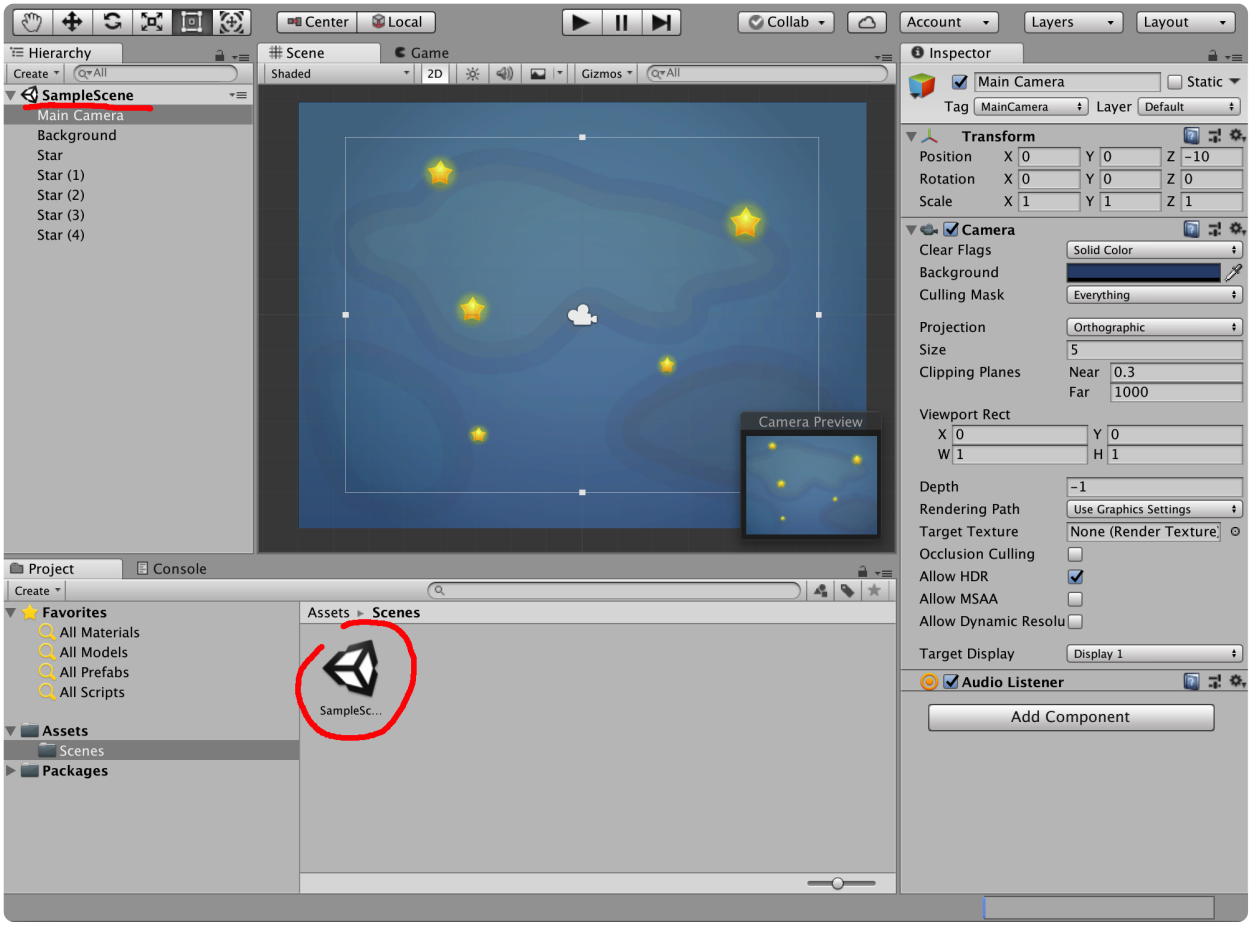


It is possible to switch the *Scene view* to 3D mode. This is useful for checking the order of the sprites along the Z-axis. In the video below:

- the *Scene view* is switched to 3D mode (http://docs.unity3d.com/Manual/ViewModes.html)
- the hand (move) tool (http://docs.unity3d.com/Manual/SceneViewNavigation.html)  is selected from the *Scene view* control toolbar
- the view of the scene is panned by left-clicking and dragging the mouse in the *Scene view* window
- the view of the scene in rotated by pressing the *Alt* key and left-clicking+dragging the mouse in the *Scene view* window - note the change of the mouse pointer to an "eye" icon when *Alt* is pressed
- the *Scene view* is switched back to 2D mode

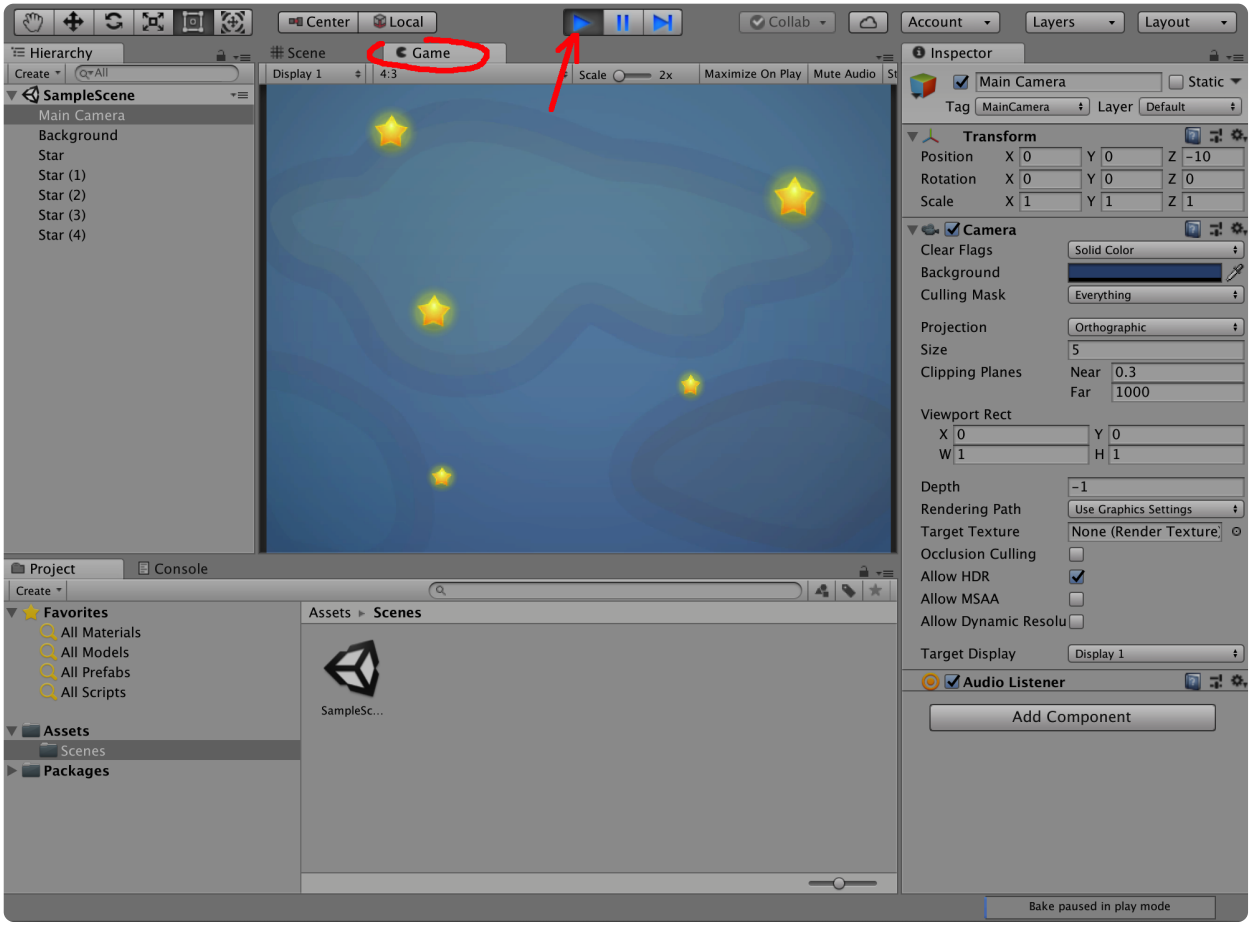


5. You've done enough in the scene that you probably don't want to lose it at this point. From the menu select *File | Save*. By default the first scene is called "SampleScene" and it gets saved to *Assets/Scenes* in the **Project panel**. If you close Unity and open it again, the *Scene view* and the **Hierarchy panel** might be blank - just need load the saved scene by double-clicking its icon in the *Assets* window. Make sure to save your scenes often so you do not lose your changes.



Get things moving

5. Try playing the game. Click on the play button  in the play control toolbar. The **Scene panel** will switch to *Game view* (https://docs.unity3d.com/Manual/GameView.html) and the editor will go into the game mode.

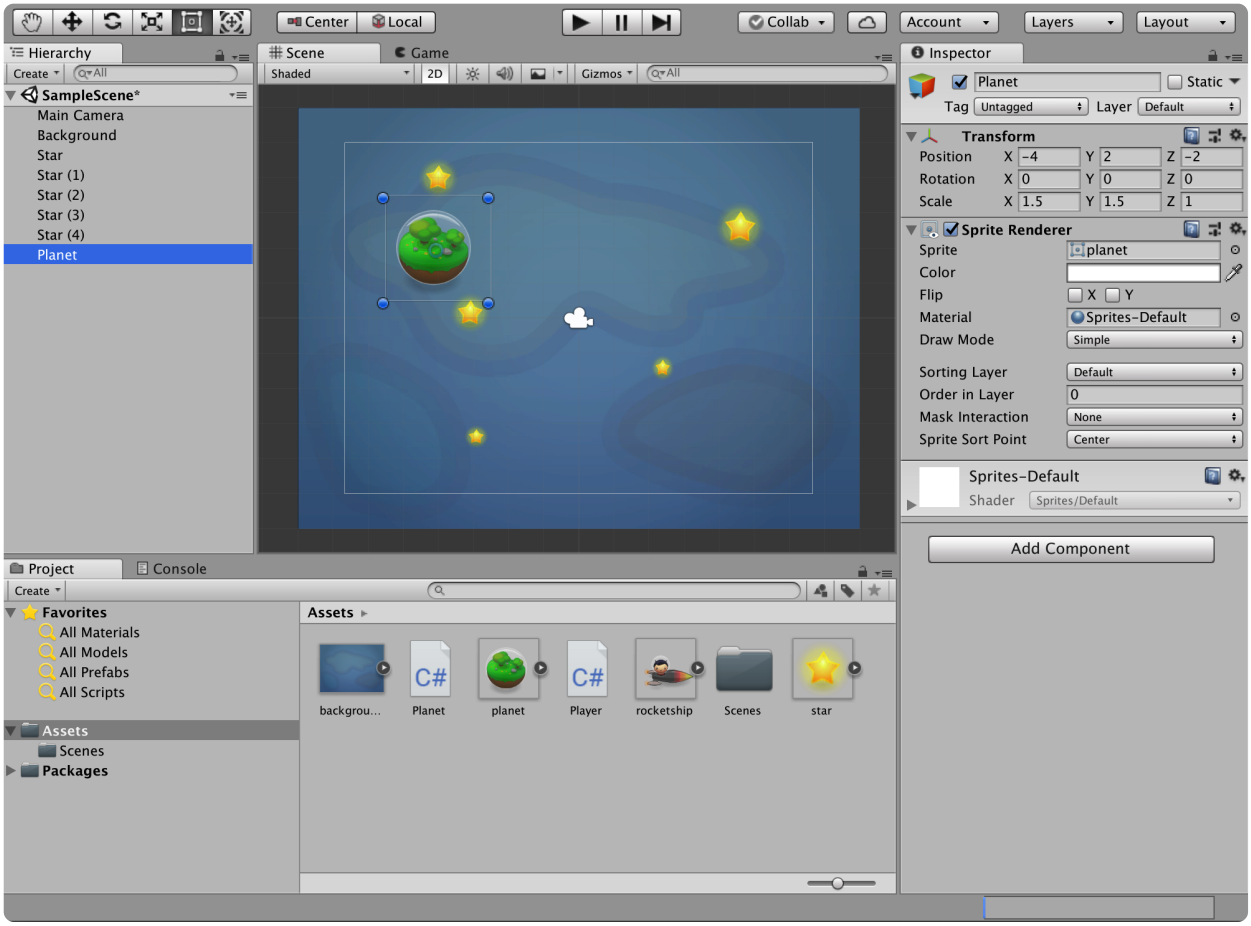


Game mode

In the game mode you get to play the game. You'll know when you're in game mode because the play control buttons turn blue and all the panels become dimmer. You can pause the game and step through it frame by frame. It is also possible to switch to the *Scene view* while still in the game mode (sometimes that is useful to see what is happening with the game objects outside the view of the "Main Camera"). However, changes made to the scene while in the game mode are temporary, and will disappear after you exit game mode. **Make sure to turn off game mode (by clicking the play button again) before continuing to work on your scene.**

7. OK, it looks nice, but nothing is happening. That's because you've just placed the objects in the scene and haven't scripted anything to happen. Time to get objects moving in the scene.

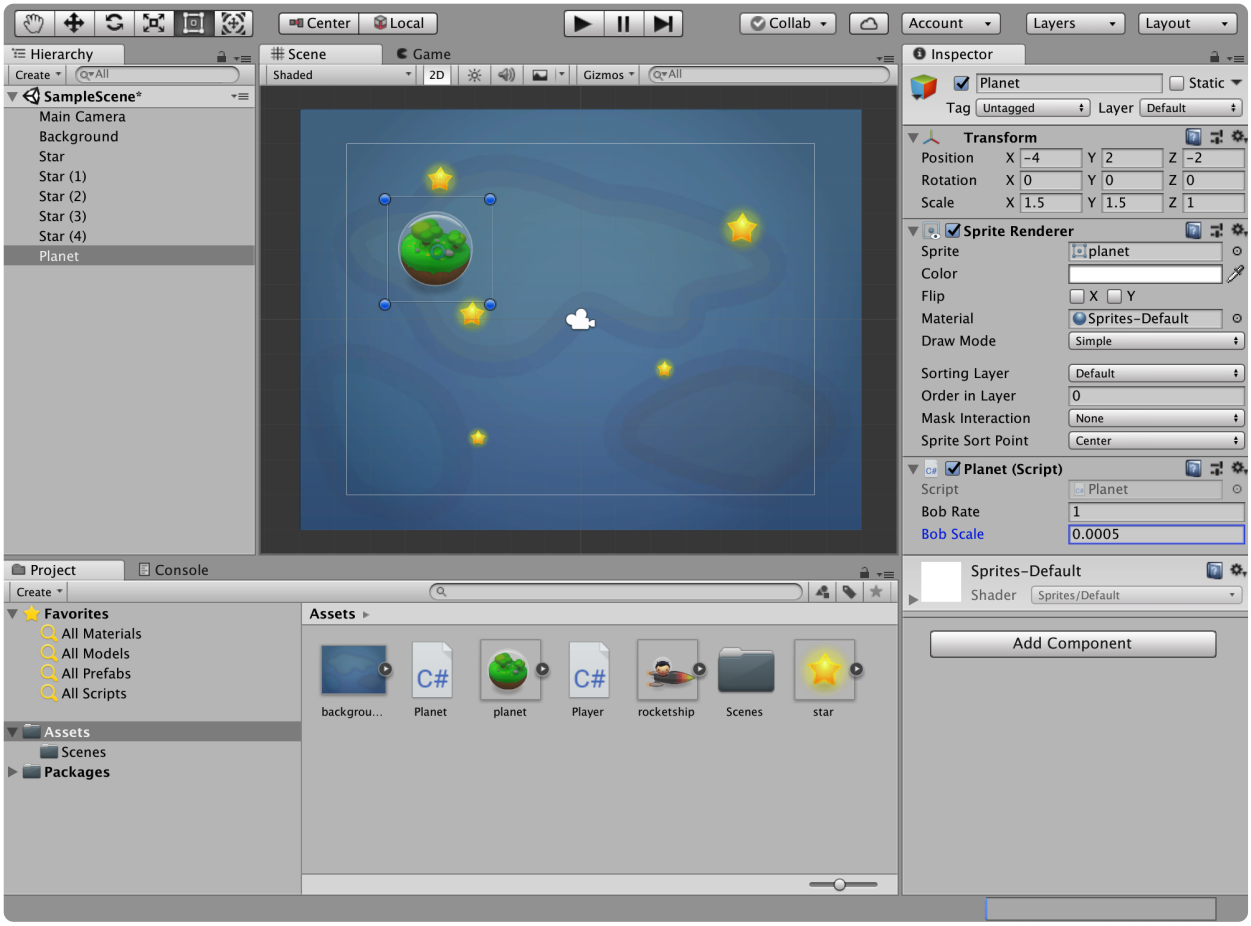
8. Switch off the game mode. Create another *2D Object* | *Sprite* in the **Hierarchy panel**. Name it *Planet*. Set its *Sprite Renderer* | *Sprite* attribute to "planet" image from *Assets*. Set the *Transform* as follows: *Position* to (-4, 2, -2), *Scale* to (1.5,1.5,1).



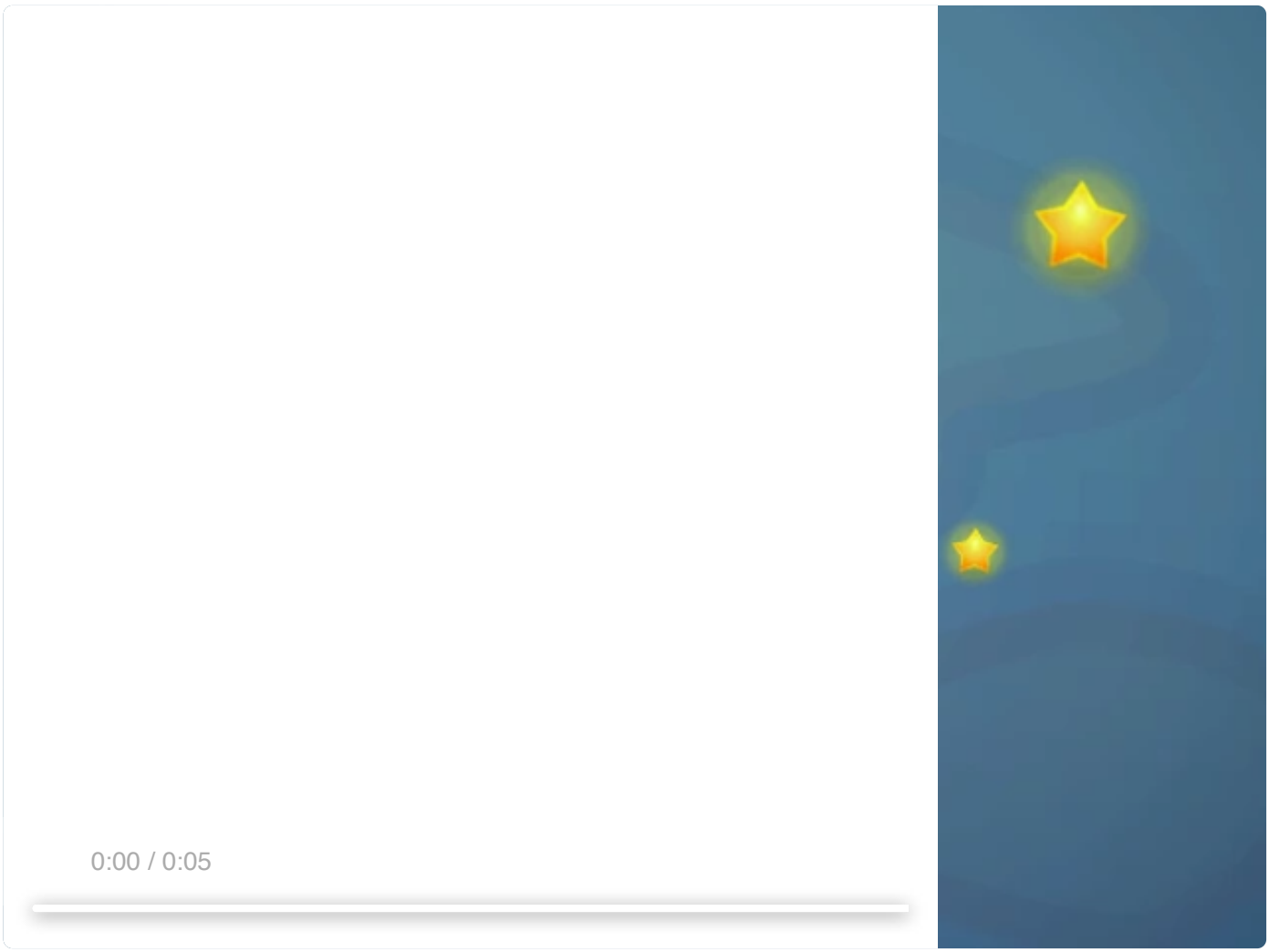
Note the Z-position of the *Planet* game object - it will get rendered in front of the *Star* sprites.

3. While the **Inspector panel** is still showing the attributes of the *Planet* game object, click the **Add Component** button. Select *Scripts / Planet*.

4. When the "Planet (Script)" component shows up in the **Inspector panel** set its *Bob Rate* attribute to 1 and *Bob Scale* attribute to 0.0005.

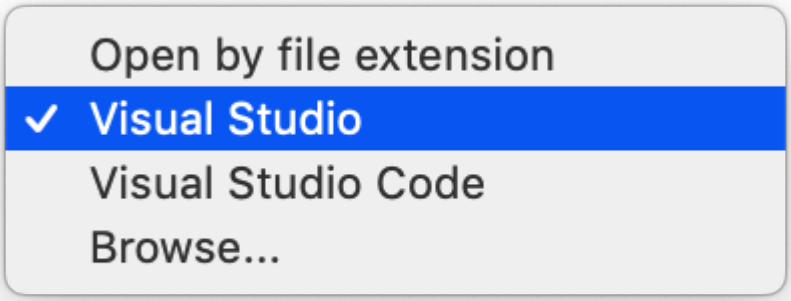


1. Press play to go into play mode. Hopefully, you get to see something like the video below:

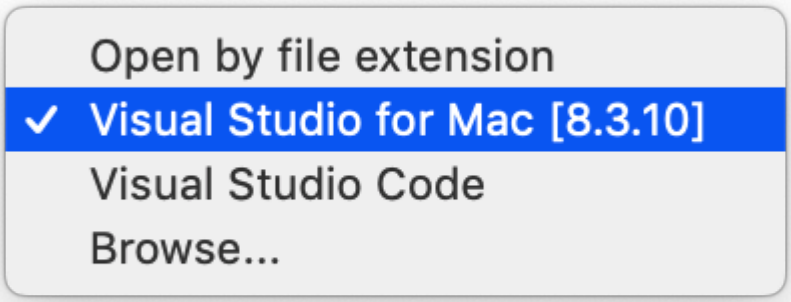


2. How does this work? Time to look at the code (http://docs.unity3d.com/Manual/CreatingAndUsingScripts.html). However, first you need to make sure the code editor is setup properly (just need to do it once).

3. The script editor that comes with Unity is Visual Studio Code, but the lab machine habve two version of VSCode and Unity seems to pick up the wrong one (that is not integrated with Unity Editor). To set the correct editor, from the main menu select *Unity->Preferences*, select the "External Tools" tab and set tge "External Script Editor" option to "Visual Studio" or "Visual Studio for Mac [version]" (whichever of the options you get). Don't select the "Visual Studio Code" option, which points to the the wrong editor.



or



4. Once you've done that, close the preferences and open them again. The "External Script Editor" should be set to "Visual Studio for Mac [version>]"

5. Back to the *Assets* panel - Double-click on *Planet* script. Visual Studio should open and display its contents.

Programming in Unity

Unity uses C# script for programmig. C# is pretty easy to follow, and it has some similarities to Java (https://msdn.microsoft.com/en-us/library/ms836794.aspx). If you need a primer on C# syntax, have a look at some of the scripting tutorials (https://learn.unity.com/course/beginner-scripting) or scripting documentation (http://docs.unity3d.com/Manual/CreatingAndUsingScripts.html).

5. Let's examine "Planets.cs":

Planet.cs

```
01: using UnityEngine;
02:
03: public class Planet : MonoBehaviour {
04:
05:    // Rate of the 'bob' movement
06:    public float bobRate;
07:
08:    // Scale of the 'bob' movement
09:    public float bobScale;
10:
11:    // Update is called once per frame
12:    void Update () {
13:       // Change in vertical distance
14:       float dy = bobScale * Mathf.Sin(bobRate * Time.time);
15:
16:       // Move the game object on the vertical axis
17:       transform.Translate(new Vector3(0, dy, 0));
18:    }
19: }
```

C# is an Object Oriented scripting language. A .cs file is expected to implement a class of the same name as the file name (without the cs extension). Therefore, "Planet.cs" implements a *Planet* class. When you add the *Planet* script component to a game object, you instantiate an object of *Planet* type and associate it with that game object.

The first line of "Planet.cs" specifies the library containing the engine itself, so that you can reference engine types and classes. The definition of the *Planet* class follows. Object that hook into the game engine need to extend the *MonoBehaviour* class, which provides a set of instance variables and behaviour methods that the engine is going to invoke. You can add new variables and methods as well as overwrite existing methods, to customise the behaviour of your game object.

The *Planet* class shown above defines two new member variables: *bobRate* and *bobScale* of type *float*. These control the speed and amplitude of the planet's bobbing motion. Note that the values of these variables are not specified in the code. If you recall, you have set those values in the **Inspector panel**. All the *public* variables appear as properties of the script component, which can be set in the **Inspector panel**. For some reason Unity changes the format (separating tokens based on capitalisation), so "bobRate" appears as "Bob Rate" and "bobScale" as "Bob Scale" in the **Inspector panel**, but they are the same variables. Unless it's explicitly specified as *public*, a member variable is taken to be private

The *Planet* class has only one method called Update(). However, before explaining what the code inside that method does, you need to understand how the engine invokes this method and how the game loop operates.

The Game Loop

The fundamental operation of the game engine is to perform the following:

- Check for player input (and store input events for later processing)
- Update the game state (some of it in response to input events)
- Render the next frame and display

This sequence of steps is done over and over again and is referred to as the *game loop*. How fast it takes to go through one iteration of the game loop will determine how quickly the next frame can be rendered - which is what determines the frame rate. To produce 30 Frames Per Second (FPS), the entire loop has about 3.3ms to run.

In the "Check for input" phase, any events from input devices (such as pressing of a given key on the keyboard) are recorded for later processing

In the "Update the game state" phase, the engine iterates over all the game objects in the scene - if a game object happens to have a script component, The Update() method of that component will be invoked. The default implementation of Update() in *MonoBehaviour* does nothing. However, if a script implements this method, it overrides *MonoBehaviour*'s one. The Update() method will get executed on every iteration of the game loop before the next frame is rendered. Inside this function you can place code that changes the game state: such as what happens in response to user input

In the rendering phase, the engine figures out what game objects are in the *Main Camera*'s view, and how to draw them to create a frame for display

The Update() method of the *Planet* class has only two lines of code, and they are responsible for animation of the planet object:

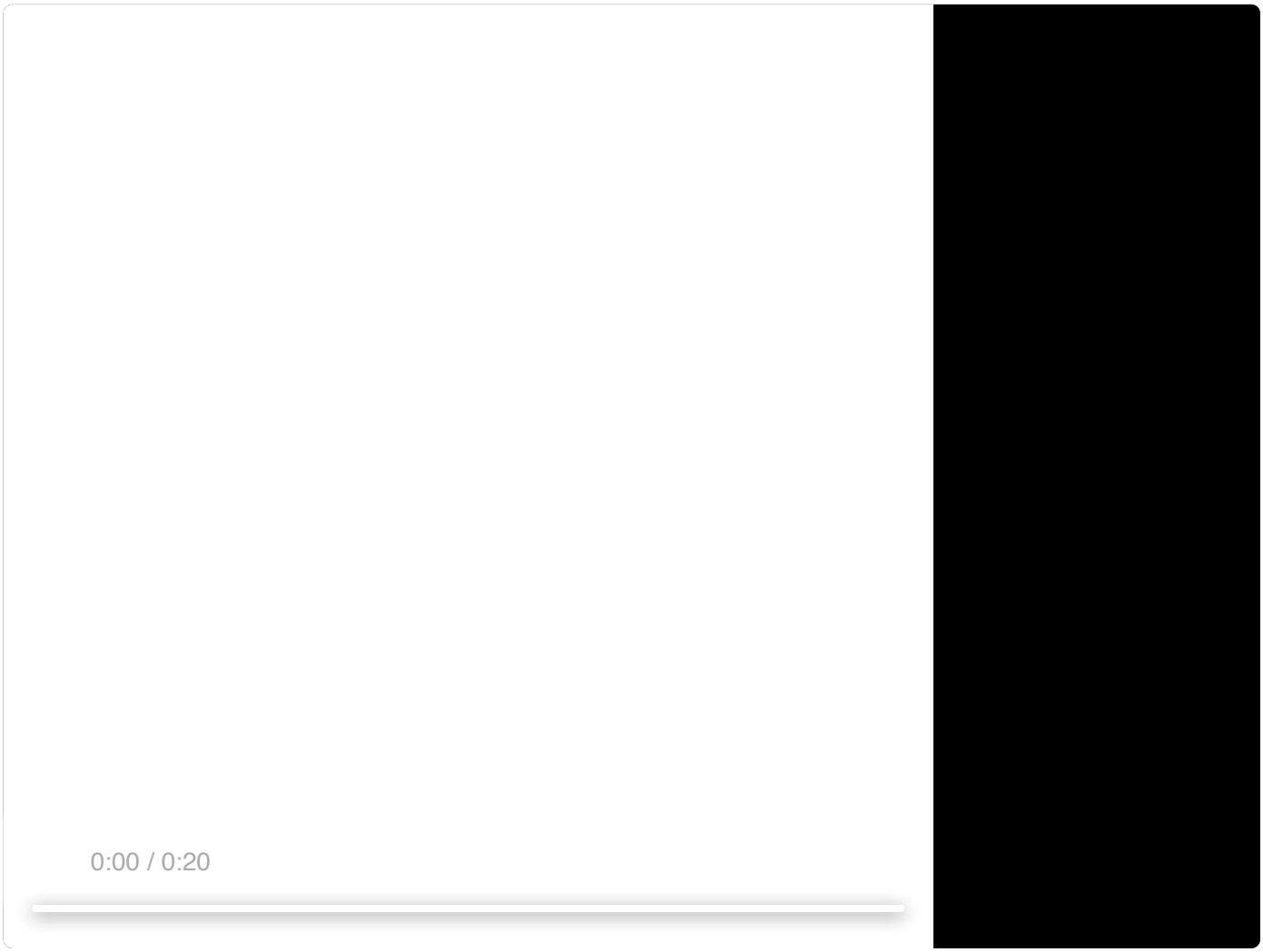
```
void Update () {
   // Change in vertical distance
   float dy = bobScale * Mathf.Sin(bobRate * Time.time);

   // Move the game object on the vertical axis
   transform.Translate(new Vector3(0, dy, 0));
}
```

The animation of the planets is a simple up and down bobbing motion based on a sine wave calculated from the *Time.time* variable. The Time (http://docs.unity3d.com/ScriptReference/Time.html) class is a built-in class used by the engine to record information about various aspects of the passage of time. *Time.time* gives the number of seconds since the start of the game. Mathf (http://docs.unity3d.com/ScriptReference/Mathf.html) is another built-in class that provides useful mathematical functions. The *bobRate* variable is the phase, and the *bobScale* variable is the amplitude of the sine wave that dictates the change in distance, *dy*.

The last line of the Update() method uses inherited member variable *transform*, which is a reference to the *Transform* component of the game object. It invokes the *Translate()* method of the Transform (http://docs.unity3d.com/ScriptReference/Transform.html) object, which changes the position of the game object by a specified vector. Vector3 (http://docs.unity3d.com/ScriptReference/Vector3.html) is another built-in class that allows the programer to specify a vector with x,y,z coordinates. The computed change in distance is passed in as a Y coordinate of a new *Vector3* object, hence the change position will be along the vertical axis.

7. Make two duplicates of the *Planet* game object. Position them anywhere you like in the scene. Each of the duplicate objects has the same script component, which will correspond to different instance of the *Planet* object. Hence, the logic of the bobbing motion will be the same. However you can specify different values for *bobRate* and *bobScale* for each instance. Go ahead, have a go at setting different values in the **Inspector panel** to get something similar to the following motion (of slightly varying bobbing rates and scales):



Add a player

8. Time to put in a character for the player to control. Create a new *2D Object | Sprite*. Name it *Player* and set its *Sprite* to the "rockethsip" image asset.

9. In the "Transform" component, set the "Player" position to (0,0,-3), no rotation, and leave scale at (1,1,1). The new Z-axis order of the game objects in the scene is as given in the diagram below. The *Player* sprite will get rendered in front of other sprites.

). Add the *Script | Player* script component to the *Player* game object. You'll notice a number of variables that need to be set, but before setting them, take a look at the script code.

l. Double click on *Player* script in *Assets* to edit the code.

Player.cs

```
01: using UnityEngine;
02:
03: public class Player : MonoBehaviour {
04:
05:    public KeyCode moveLeft;
06:    public KeyCode moveRight;
07:    public KeyCode moveUp;
08:    public KeyCode moveDown;
09:
10:    public float speed;
11:
12:    // Update is called once per frame
13:    void Update () {
14:       float delta = speed * Time.deltaTime;
15:
16:       if(Input.GetKey(moveRight)) {
17:          // Move to the right
18:          transform.Translate(new Vector3(speed * delta, 0, 0));
19:       } else if(Input.GetKey(moveLeft)) {
20:          // Move to the left
21:          transform.Translate(new Vector3(-speed * delta, 0, 0));
22:       } else if(Input.GetKey(moveUp)) {
23:          //Rotate counterclockwise
24:          transform.Rotate( new Vector3(0, 0, speed * delta * 10));
25:       } else if(Input.GetKey(moveDown)) {
26:          //Rotate clockwise
27:          transform.Rotate( new Vector3(0, 0, -speed * delta * 10));
28:       }
29:    }
30: }
```

The script defines four public variables of type KeyCode (http://docs.unity3d.com/ScriptReference/KeyCode.html), which is a built-in class that contains codes for keyboard keys. Representing key codes as variables allows for arbitrary key mapping. The four variables in the above script correspond to left, right, up and down movement.

There's another variable of type *float*, which determines the movement speed of the *Player* game object.

There is an Update() function in this script, which will be called once per frame. The *delta* variable calculates the range of the movement in this frame.

The length of time it takes the game engine to complete one iteration of the game loop is not constant. As the state of the game changes, the number of things to do in each iteration of the loop may vary. Hence, Update() gets called at varying time intervals. In order to move an object at a constant speed, say some number of "units" per second, the change in position needs to be integrated over time since the last Update(). The Time (http://docs.unity3d.com/ScriptReference/Time.html) class provides the *deltaTime* variable that records time in seconds since the completion of the last frame. This value is multiplied by the speed to get the distance.

For example, if *speed=2* ("units"/s), and *Time.deltaTime=0.33* (s), then the computed distance is:

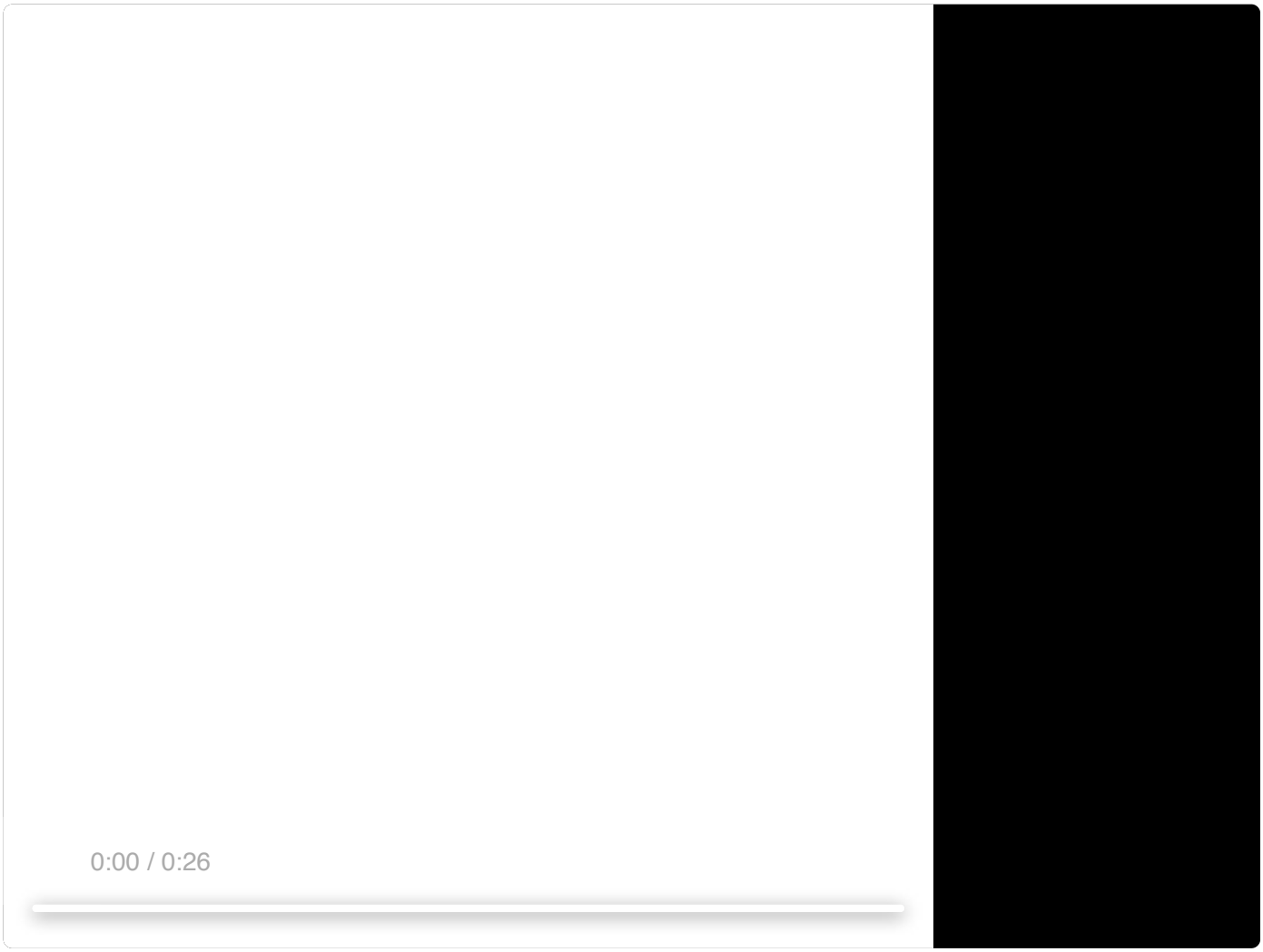
`2` ("units"/s) `* 0.33` (s) `= 0.66` ("units").

When the `moveRight` key is pressed, the *Player* game object will be translated by `dist` "units" in the x direction. If the `moveLeft` key is pressed, the *Player* game object will be translated by `dist` "units" in the negative x direction.

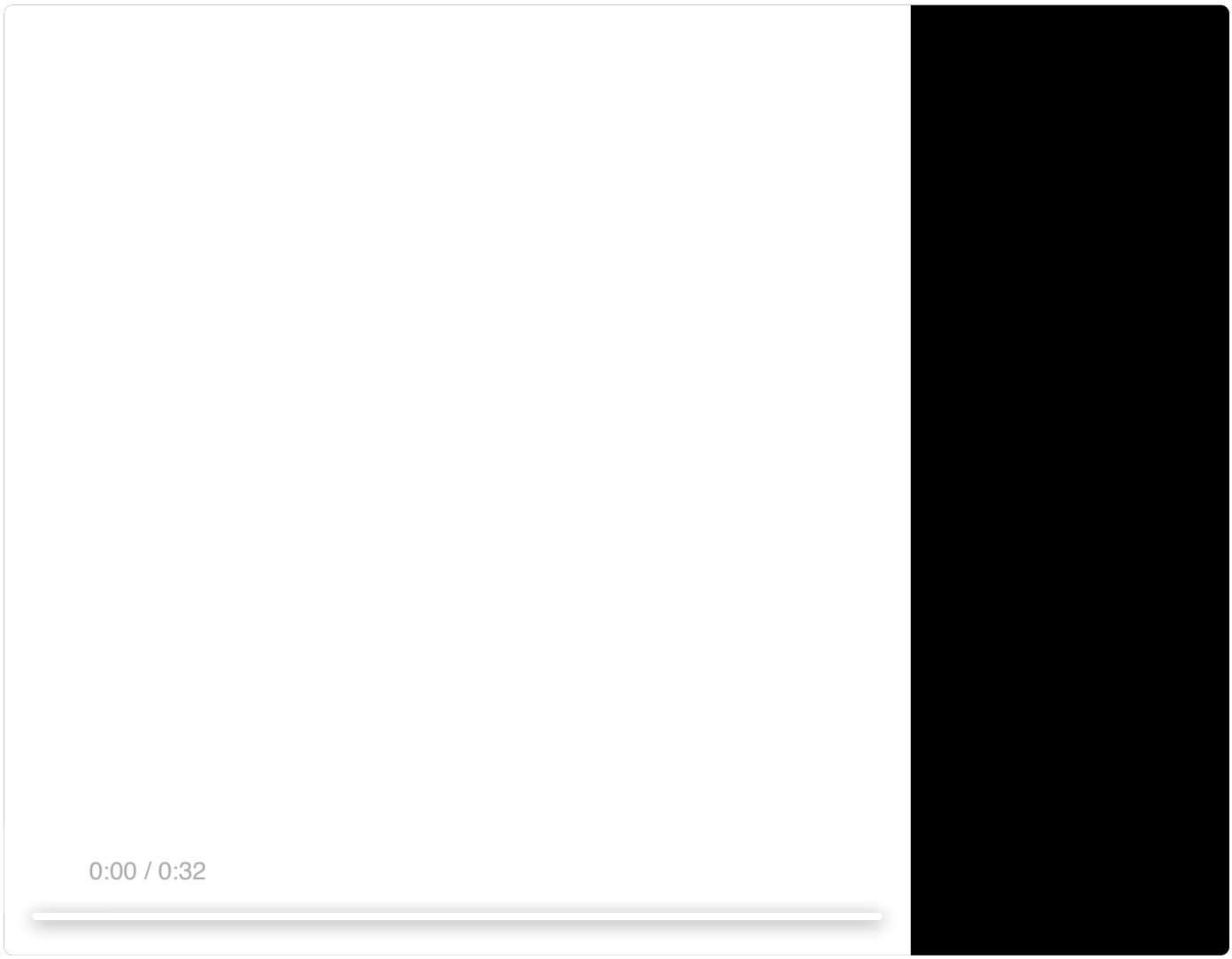
If the `moveUp` key is pressed, the *Player* game object will be rotated about Z-axis by `dist`. If the `moveDown` key is pressed, the *Player* game object will be rotated in the opposite direction about Z-axis by `dist`.

These movement mechanics might seem odd - it's best to get a feel for it by playing the game. However you still need to define the keys and the speed of movement

2. In Unity editor, in the **Inspector panel** set the *Move Left* attribute of the *PlayerScript* to "LeftArrow". Unity recognises that you're setting a variable of *KeyCod* type, and it helps you by providing a list of options. Set *Move Right* to "RightArrow", *Move Up* to "UpArrow" and *Move Down* to "DownArrow". Set the *Speed* to 2.
3. Press the play button. You should get something like the video below (controlling the player with the arrow keys):



What is going on there? Initial left and right movement should be straight forward. Rotation by going up and down is a bit strange, but still it's doing exactly what the *Player* script prescribed. But after a bit of rotation the left/right movement is not horizontal with respect to the camera. Instead, it's horizontal with respect to the the new rotated position of the player. Take a second to ponder why this is happening (before reading the answer below).



The left/right movement of the *Player* sprite is relative to its own rotation, because the *Translate()* method (by default) takes its argument vector to be in local co-ordinate system of the game object. The translation into the shift in world coordinates is computed internally. It is possible to specify the desired coordinates system of the shift vector with an extra argument to the Translate (http://docs.unity3d.com/ScriptReference/Transform.Translate.html) function, but most of the time (as in this case) the local co-ordinate system is what we want.

4. Take some time to play with the code and try to get different movement mechanics.

Debugging

5. At various times during development, things are not going to work as they should. Most of the time it will have to do with a mistake in script code. Whenever something is wrong, Unity will try to alert you to the problem by displaying error messages in the *Console*. Whenever there is an error, one of the messages will also appear at the bottom of Unity window - so you can see that there is a problem, even if the *Console* view is not currently open. Try it - in the first line of the *Update()* method in *Player* script change *Time.deltaTime* to something else - like *Time.deltatime* (this kind of a mistake is very common). *deltaTime* is a built-in variable, so you can't spell it any way you want, and capitalisation matters. Save the script. Unity should sync up, detect that there is a problem, and alert you with a message like in the screenshot below:

Unity will refuse to play the game when there is something wrong with the script. But, at least in this case, it's pretty good in letting you know what the problem is (and even how to fix it). Go ahead and change back that variable to *Time.deltaTime*.

Common scripting mistakes

When things don't work as you expect, even when scripts compile fine, there's a chance you got one of these common errors:

- Variable/class name name has been mistyped - remember C# is case sensitive.
- A method name has been mistyped - game engine looks for Update(), not update() method - if you get that wrong, your method will never get called.
- Value of a variable hasn't been defined before first use - did you remember to go back to the *Inspector panel* and initialise the public variables?

5. Sometimes, when the code compiles but things don't work as expected, it's useful to examine the state of your script while the game is running. In VSCode you can create breakpoints, which are places where execution of the code will stop during gameplay, allowing for inspection of variables. First though, you must enable debuggin in the Unity Dditor (running in debug mode slows down the program a touch, and by default Unity Editor has the debugging mode disabled). Switch to the Unity Editor window and click on the "bug" icon in the bottom-right of the Unity Editor Window and enable debugging.

   
   Script errors

7. Back to VSCode. To create a breakpoint at a specific line, click the column next to the line. For example, in the screenshot below, a breakpoint has been created on the line which moves the *Player* sprite to the right. You'll know that the breakpoint has been set, because you'll see a red-dot in place where you just clicked (you can remove it by clicking it again).

   
   Unity Editor enable debug option

8. Next, select the debugging tab in VSCode by clicking on the  button. If a window pops ups with options for process to which attach the debugger, select "UnityEditor [name of your current scene]" and click the "Attach" button.

   
   Connect debugger to current scene

9. Now, go back to Unity Editor and press play. As soon as you move to the right, the game will freeze. Go back to VSCode editor and you'll notice an arrow pointing at the line with the breakpoints - that means that execution of the script has been halted at this point. Down below the code you can examine the values of the variables (select the *Locals* tab). You can also step through the code line by line using the controls a at the top of the window. You can resume the game (play button) or stop the debugging session completely (the stop button).

   
   VSCode debug

Have fun

That is the end of this exercise. Try to modify the scene, have some fun with it. See if you can change the player movement, or maybe get the stars to move in the back. Perhaps have a go at adding some lighting effects. Learn more about the Unity Editor by tinkering with this project.

Assessment

Show your work to the demonstrator for assessment.