

PURPLE BELT



CODE NINJAS®

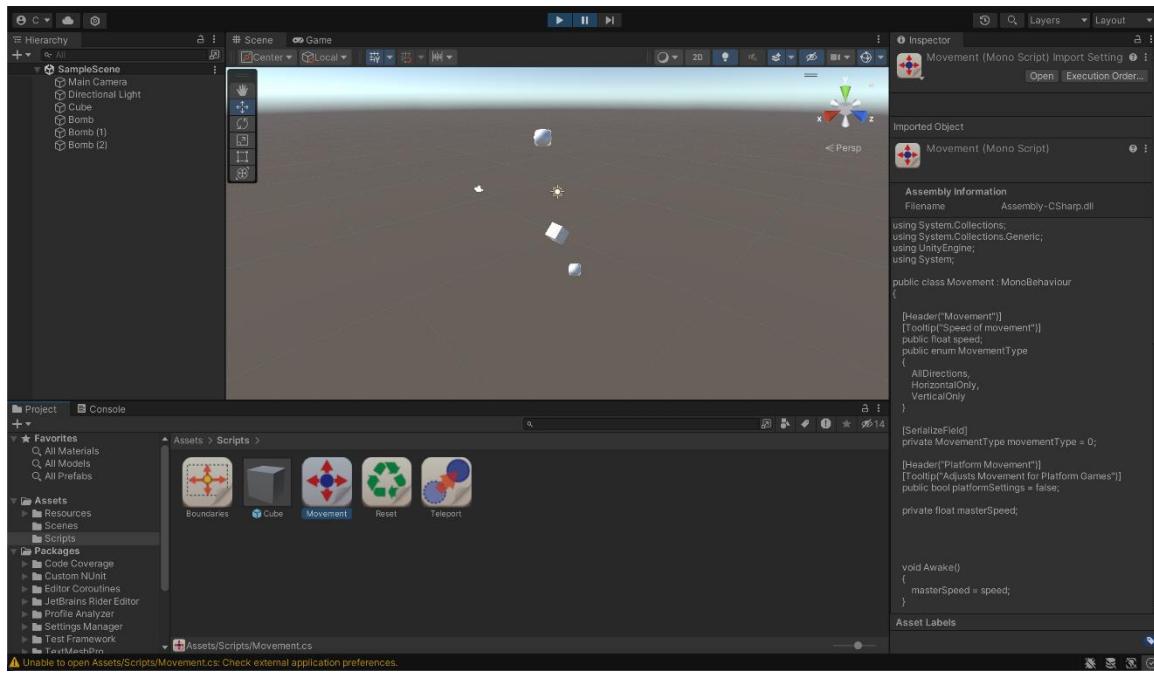
Welcome to Purple Belt!	3
Working in Unity	4
Getting Started with Unity	6
Activity 1: Dropping Bombs	11
Saving and Submitting Your Games	35
Prove Yourself: Color Drop	39
Activity 2: Scavenger Hunt	46
Prove Yourself: Particle Hunt	87
Activity 3: Meany Bird	91
Prove Yourself: Meaner Bird	118
Activity 4: Sketch Head	123
Prove Yourself: TrickHead	153
Activity 5: Don't Touch the Cubes	157
Prove Yourself: Don't Touch the Chopsticks	175
Activity 6: SuperShapes	179
Prove Yourself: Super Duper Shapes	201
Activity 7: PolyRun	202
Prove Yourself: PolyRun v2	228
Activity 8: Dropping Bombs Part 2	233
Activity 9: Dropping Bombs Part 3	268
Activity 10: Dropping Bombs Part 4	308
Activity 11: Dropping Bombs Part 5	350
GLOSSARY	357

Welcome to Purple Belt!

Welcome to Purple Belt! You've reached a significant milestone in your journey. It's a great accomplishment, and you should be proud of yourself. Everything that you've learned up to this point has been in preparation for this.

From now on, you'll be working with the same tools that professionals use to make games. This might seem a bit overwhelming at first. There are so many different options and ways to do things that it might make your head spin at first. Don't forget what you learned before in the previous belts. At the heart of every lesson are the same core programming concepts. You're still using variables, conditionals, and functions. The difference with Unity is that you have so many more opportunities than you had before.

But before we begin, what is Unity?



Working in Unity

Unity is the same tool used by professionals to make games such as Pokémon Go, Fall Guys, and Among Us. With Unity, you can bring various elements together to build a complete game that can be played on different devices and platforms.

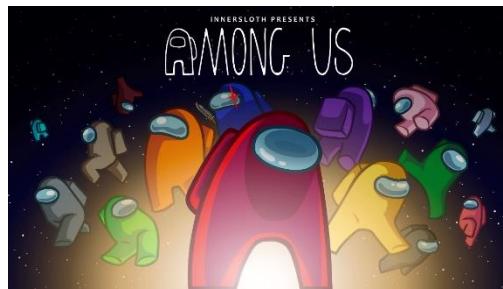


Image Sources (Left to Right): 1) Among Us: [Epic Games](#), 2) Fall Guys: [Fall Guys | Download & Play Fall Guys on PC for Free – Epic Games Store](#), 3) Pokémon GO: [Pokémon GO \(pokemongolive.com\)](#)

What do I need to know?

To get you started, we will concentrate on the Unity interface, how to find things and what to do with them. For the first few activities, you will use pre-built scripts to make your games. This approach will ease you into Unity's environment until you are familiar enough with Unity to start working with the programming language, "C Sharp" (C#).

Unity utilizes C# because it seamlessly aligns with the components of Unity game objects. Furthermore, as you'll soon discover, you can attach multiple C# scripts to a single object, facilitating the writing of code in manageable, bite-sized portions.

A Brief Word About 2D and 3D

Games are typically developed in either a 2-dimensional (2D) or 3-dimensional (3D) environment.

In a 2D environment, your focus lies in the x and y axes, governing an object's horizontal and vertical movement. The x-axis represents left and right movement, while the y-axis represents up and down movement. However, it's worth noting that the z-axis can also be employed in 2D to position or layer objects in depth, albeit without the perception of 3D space.

On the other hand, 3D introduces an additional dimension, the z-axis, which allows for forward and backward movement. Unity provides the flexibility to create games in either a 2D or 3D environment, even though Unity's core architecture remains 3D. You can initiate a Unity project in 3D and utilize a 2D (orthographic) camera and 2D colliders to craft a 2D game using 3D objects, as we'll demonstrate later in this section.

While all the activities in this book start as 3D projects, the initial ones exclusively involve the x and y axes, helping you understand that objects in Unity can take on various dimensions as needed!

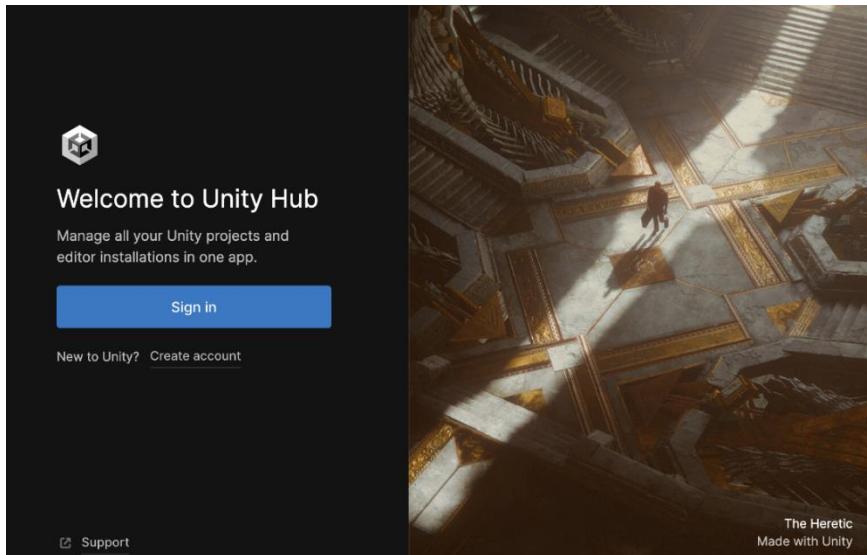
Getting Started with Unity

- 1** To start, locate **Unity Hub** on your desktop. The icon should resemble something like the image below. Move your cursor to Unity Hub and click it twice to open the app.

If you can't locate Unity Hub on your desktop, let a Code Sensei know and they will assist you!



- 2** Once you click Unity Hub it will appear, and you will be greeted by a screen that looks like this. Please note, the screen appearance may differ as Unity has a Black and White appearance option.



- 3** If you already have a Unity account then you can select the “**Sign in**” button otherwise click on “**Create account**”. Remember, if you are under the age of 13, you will need to get permission to create a Unity account. If you’re having trouble, have your Code Sensei help you.

When you click on the create account button, you will see a screen like the one below:

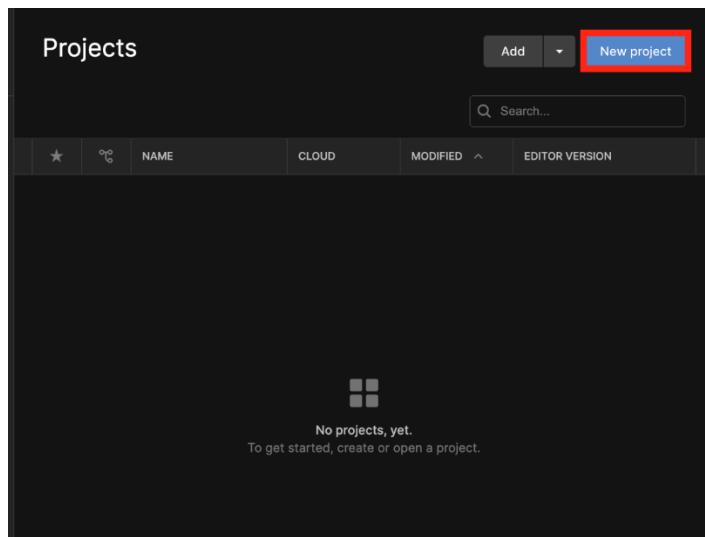
The screenshot shows the 'Create a Unity ID' page. At the top left is the Unity ID logo. Below it is the heading 'Create a Unity ID'. A link 'If you already have a Unity ID, please [sign in here](#)' is present. The form has two main sections: 'Email' and 'Password' on the left, and 'Username' and 'Full Name' on the right. Underneath these are several checkboxes for terms and conditions, followed by a reCAPTCHA field. At the bottom are buttons for 'Create a Unity ID' and 'Already have a Unity ID', and social sign-in links for Google, Facebook, and Apple. A 'OR' link is also visible.

You can fill in your details or have a Code Sensei help you do this. You can also sign on using a single sign-on provider, such as Google or Apple. Once you finish your details, click **Create a Unity ID**. Then return to Unity Hub.

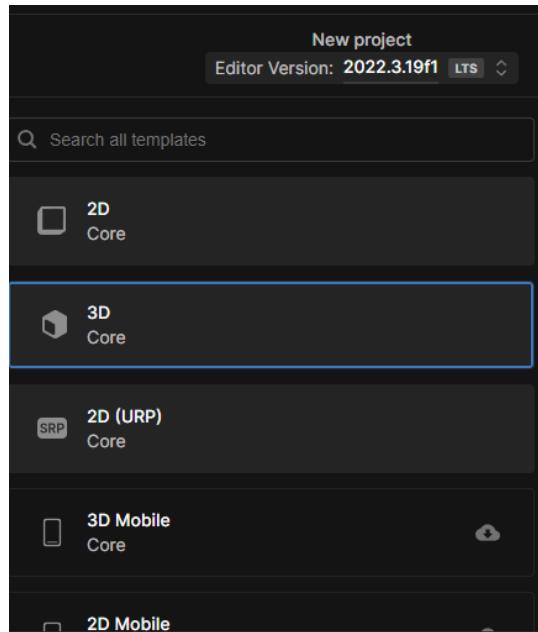
Once your account is created, sign into Unity Hub with it.

4 Once you are signed in, you will see the Projects window.

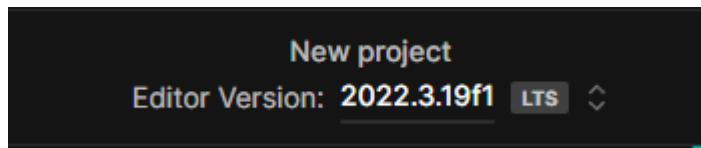
When you first start, you will not have any projects to open. That's okay. Make a new one by clicking the blue "**New project**" button.



5 On the left side of the screen are options for what type of project you will be creating. You will most often be choosing 2D or 3D. Go ahead and leave 3D selected.



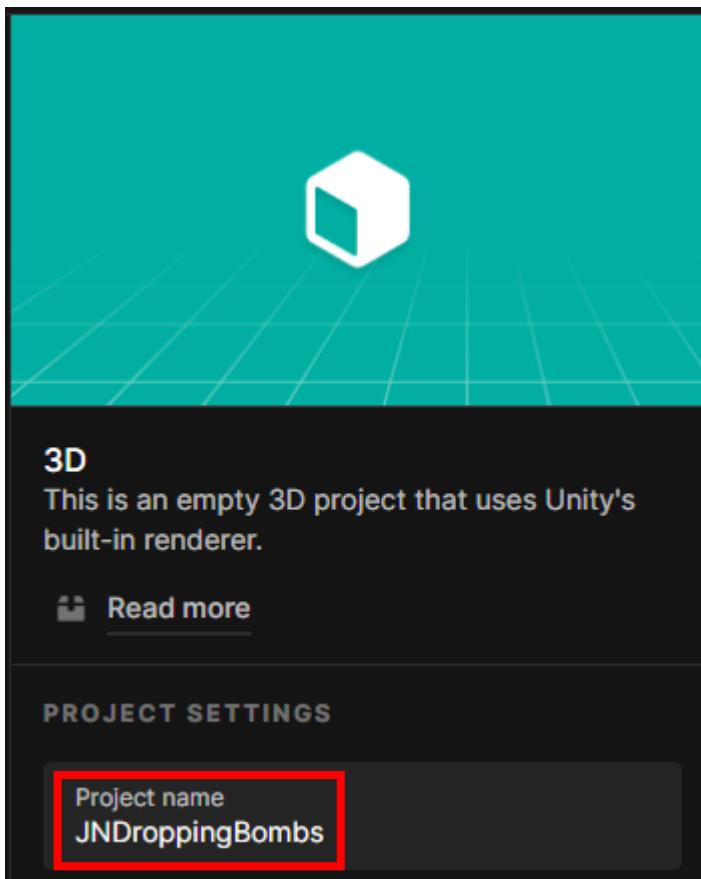
-
- 6** Ensure that the **Editor Version** at the top of the screen states “**2022.3 LTS**”



Using this version will ensure everything looks the same as you follow along in this guide.

-
- 7** You will also need to select a name for your project. This is very important: you *cannot* change the name of a project once it is created. Make sure that it describes the project well.

Your first project is **Dropping Bombs**, so select a name that reflects that. For example, if your name is John Ninja, why not use your initials and name your file **JNDroppingBombs**.



-
- 8** Finally, you need to choose where to store your Unity projects. It is best to create a new folder with a name like **John Ninja's Unity** or **JNUnity** to keep everything in one place.

However, your Code Ninjas location may handle Unity project storage differently. **Ask your Code Sensei** where to save your Unity projects before proceeding to the next step!

- 9** All that's left to do is click on **CREATE** and let Unity set up the files for your first project! Once that's done, move to the next page to create your first game in Unity!
-

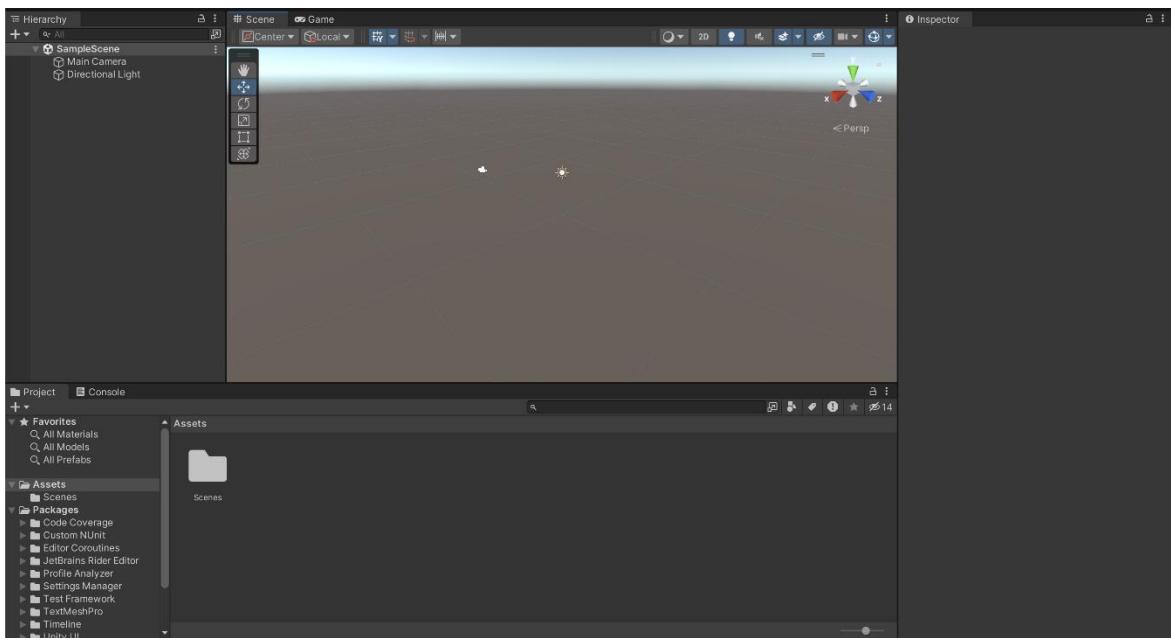
 **PRO TIP!**

Projects in Unity are saved in the location that you've chosen as a folder with the name of your project. If you ever need to duplicate a project, just make a copy of the folder and all its contents and paste it and give it a new name. To open the new folder in Unity, click on Add and select the new folder.

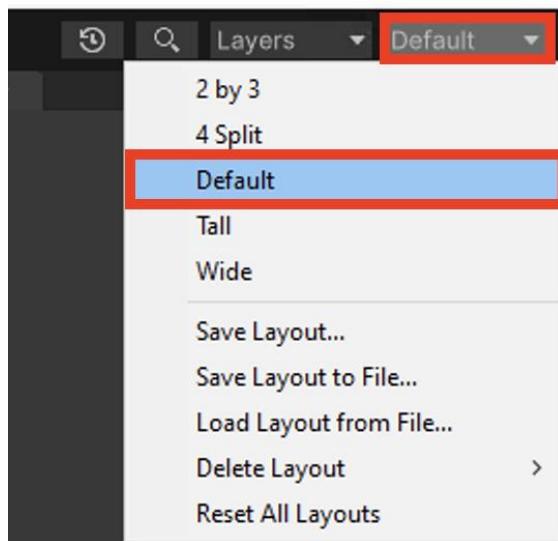
Activity 1: Dropping Bombs

This first project in Unity is meant to show you the possibilities achievable even without elaborate graphics, animations, or complex scripts. If you haven't done so already, open Unity and create a **new** 3D project, and give it a name like **MyInitialsDroppingBombs**.

- 1** After you have created your new project, Unity will open and show you something like the image below. Don't worry if it doesn't look exactly like this, we'll take care of that in the next step.

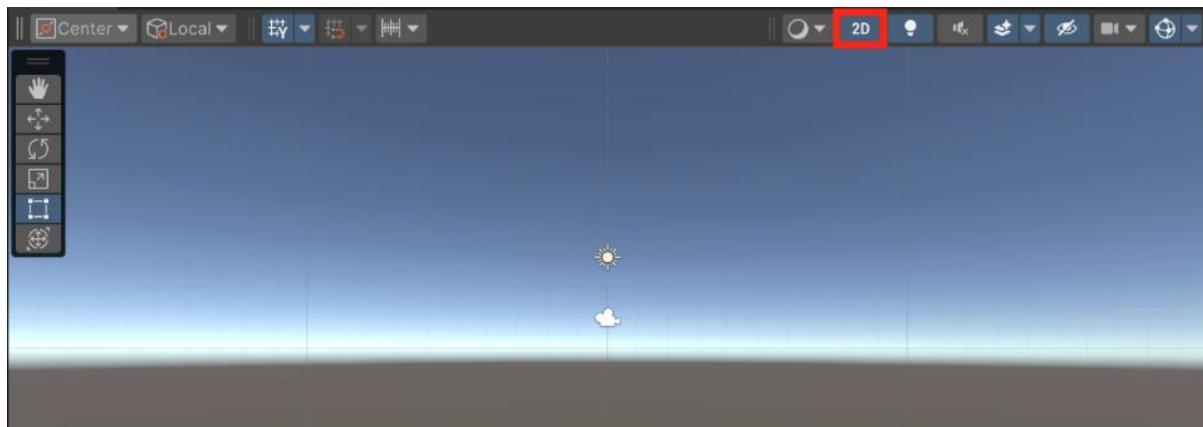


- 2** Unity gives you the option to rearrange the layout based on what works best for you. But for now, let's make sure you are using the default layout to avoid confusion. Click the **Layout** button located in the upper right corner. Select **Default** and your window should look just like the above image.

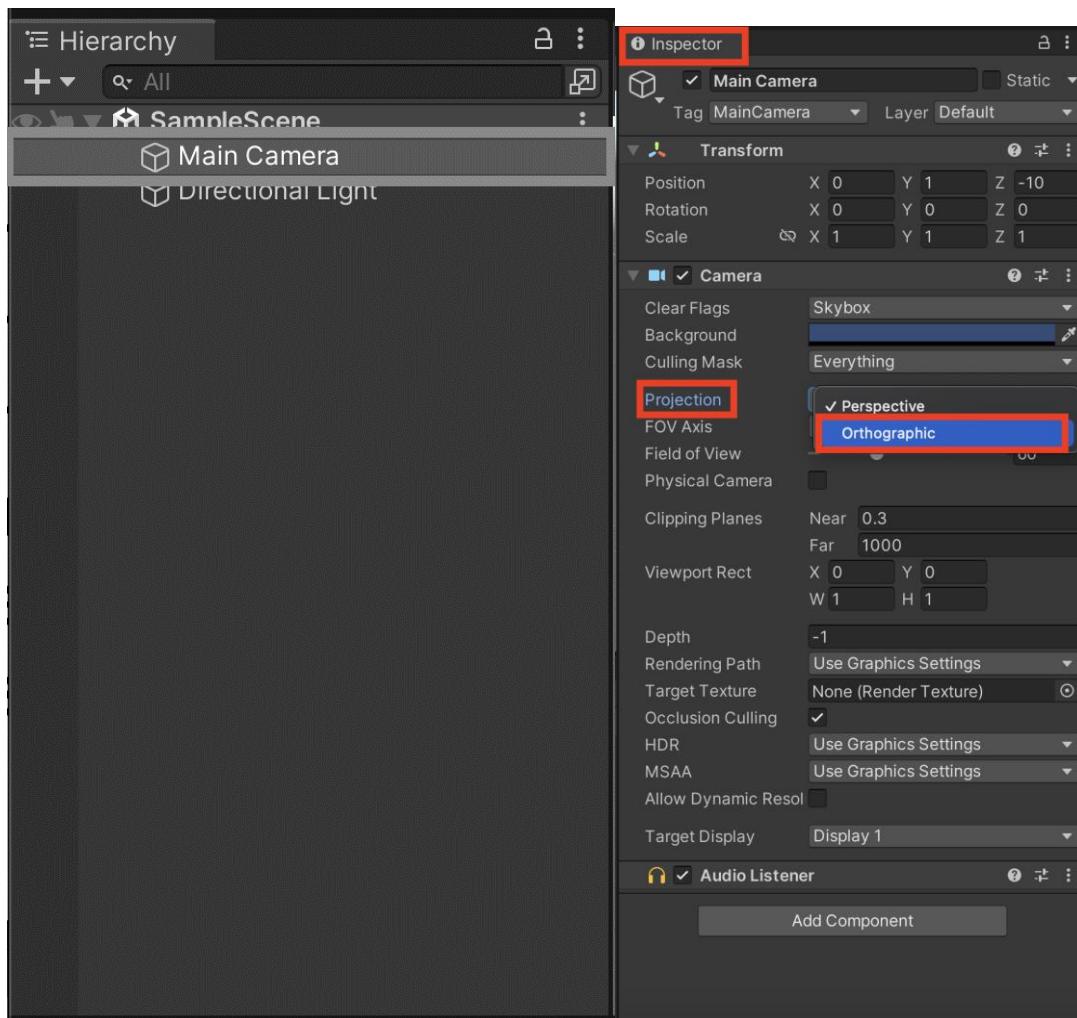


If you decide to rearrange the layout, you can use this same menu to save it for the next time you need it by clicking the "Save Layout..." button.

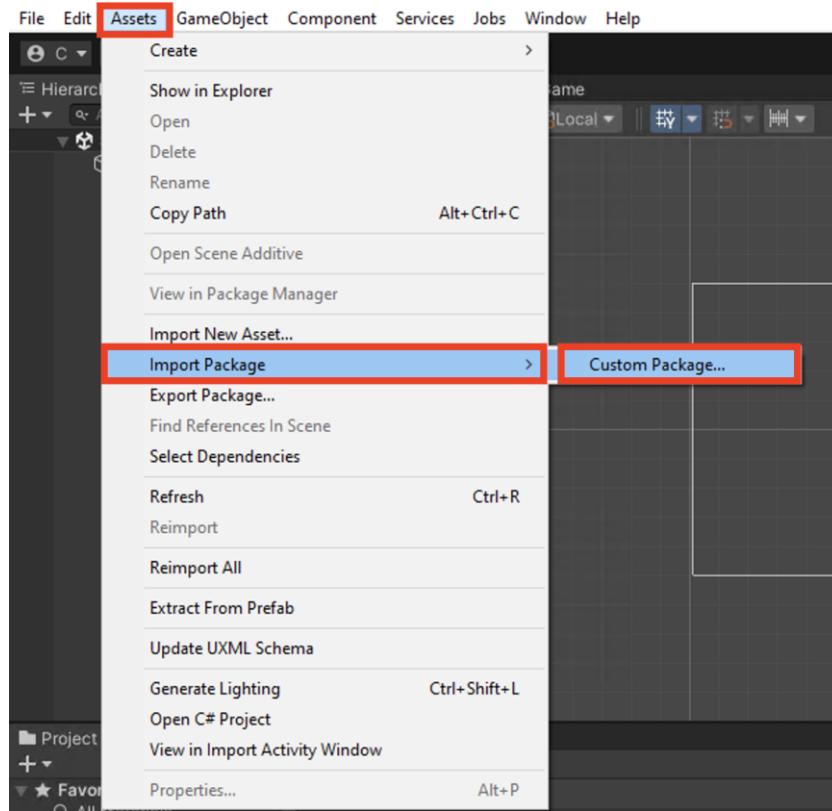
- 3** Even though this is a 3D Unity project, our game is only going to use movement in 2 dimensions (left to right, up and down). With that in mind, we can change the view so that it is strictly 2D. To do this, click the **2D** button above the scene window as shown below.



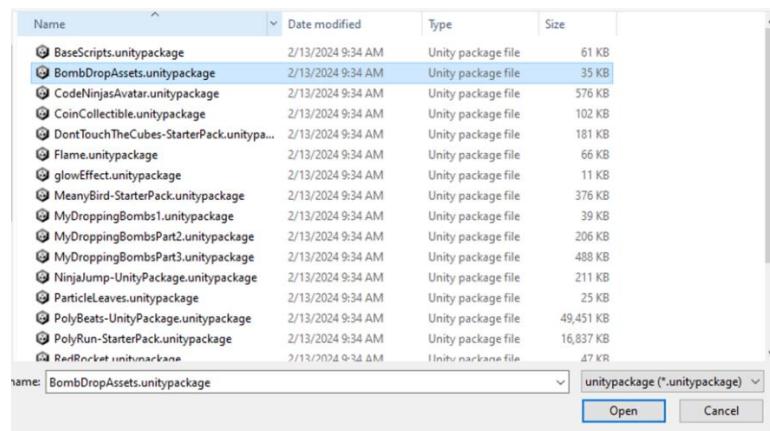
- 4** On the left side of the display is the **Hierarchy** panel. This shows you all of the **GameObjects** that are inside your scene. By default, Unity creates a Sample Scene and sets up a camera and light for you. Select the **Main Camera**. To the right of the scene, there is a panel called the **Inspector**. This panel contains all of the components associated with the selected **GameObject** in the **Hierarchy**. In this case, we want to change the **Projection** property of the camera from **Perspective** to **Orthographic** as shown below. We're doing this so that when we play the game, the objects will maintain their proportions and not get distorted by a 3D camera.



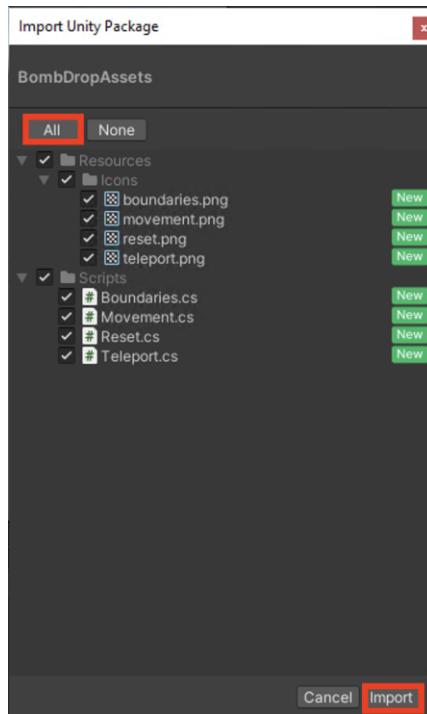
5 To make things easier, Code Ninjas has created some special scripts to use in this game. To load them, click **Assets** and select **Import Package** followed by **Custom Package**.



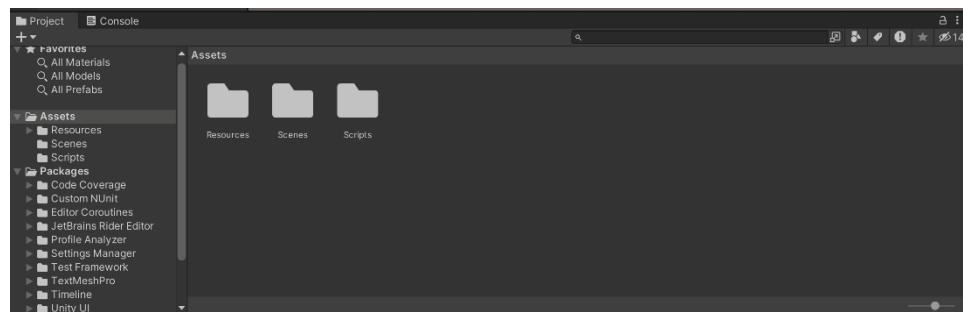
6 Navigate to the folder where your resource files are (ask your Code Sensei if you are uncertain) and select **Activity 01 - MyDroppingBombs1.unitypackage**.



7 Unity will show you an import window with a list of the assets in the package. This gives you the option of loading just some of the assets instead of all of them. In this case, we want them all, so click **All** and then click **Import**.

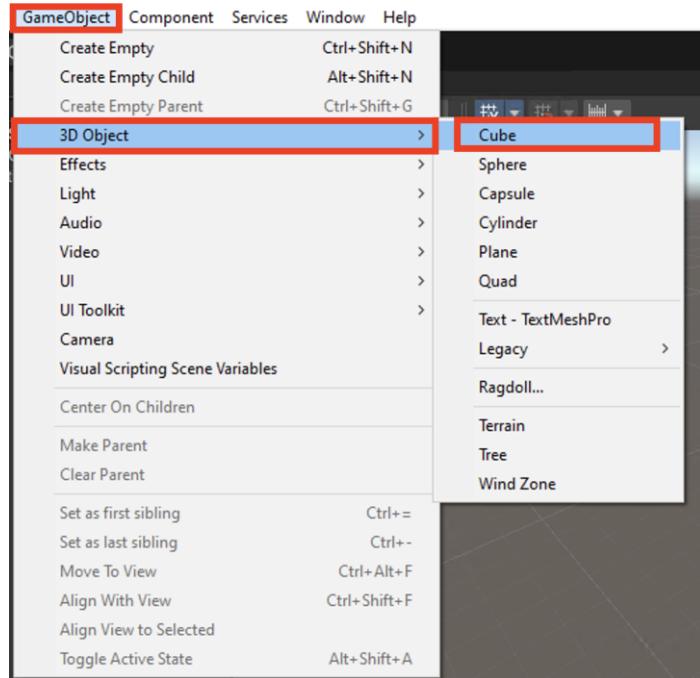


8 The assets that you just loaded have been placed in the **Project** panel at the bottom of the Unity editor. The Project panel shows you everything that is a part of your project in folders, just as if you were browsing through files on your computer. In fact, these folders have been created as part of your project. Notice that you now have a new **Resources** folder and a **Scripts** folder that were imported with your package. You'll be using the scripts later.

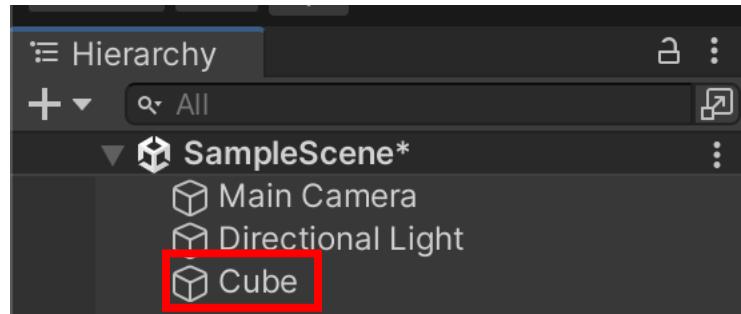


In Unity, the **Scene**, **Hierarchy**, **Inspector**, and **Project** panels are where you'll be doing most of your work, so feel free to take a few moments to look around each of these panels and become familiar with their functionalities.

- 9 At the moment, the scene is a bit empty. Let's add a **GameObject** to the scene. Click the **GameObject** menu, select **3D Object** and **Cube** to add a cube to the scene.

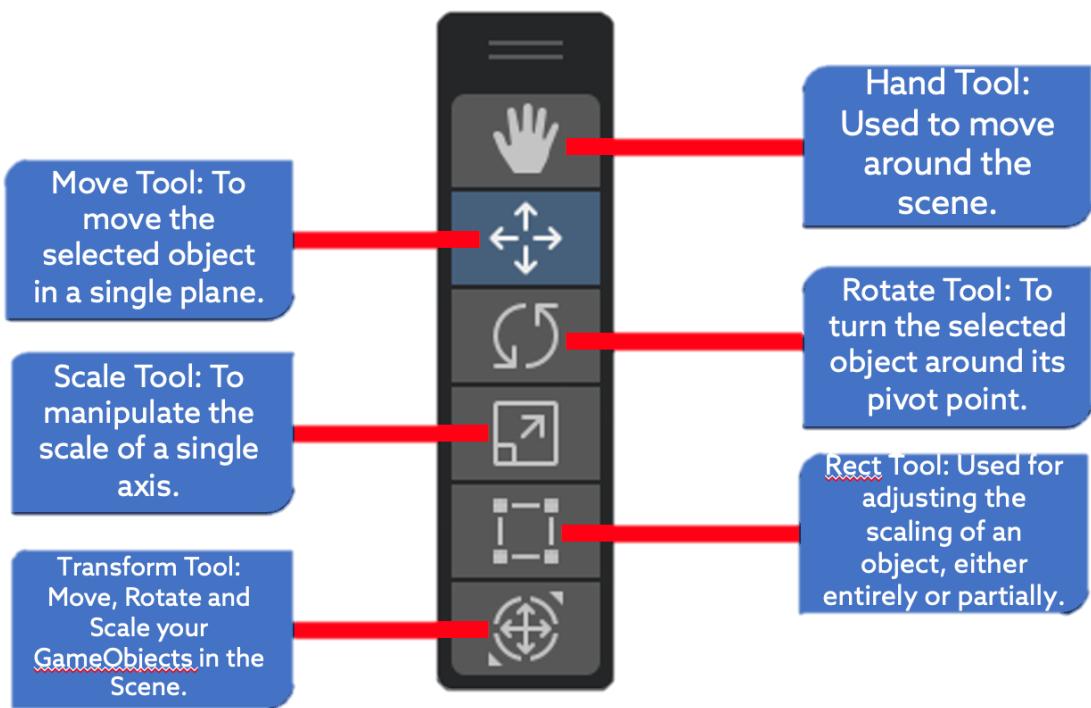


- 10 You should now see a cube added in the **Hierarchy** panel. Select the cube (if it isn't selected already) to see the components for the cube in the **Inspector** panel.



11

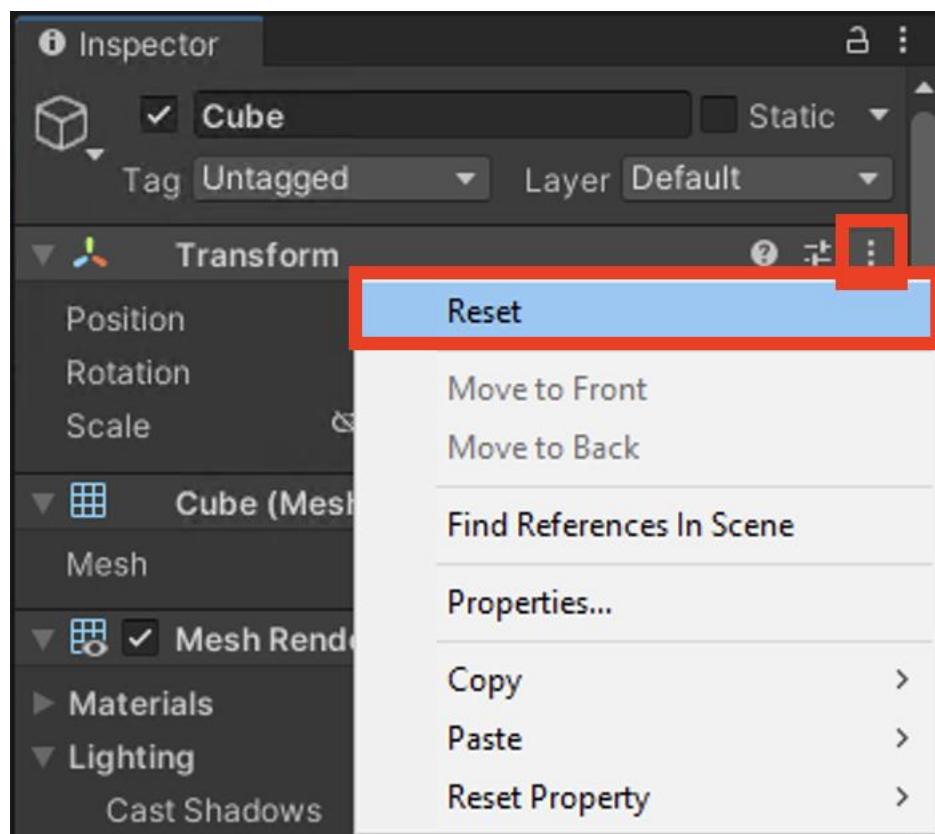
If you want to reposition, rotate, or make other changes to a **GameObject** within a scene, select the object in the scene by clicking on it and then using the tools above the scene to manipulate it. Go ahead and give it a try with the cube.



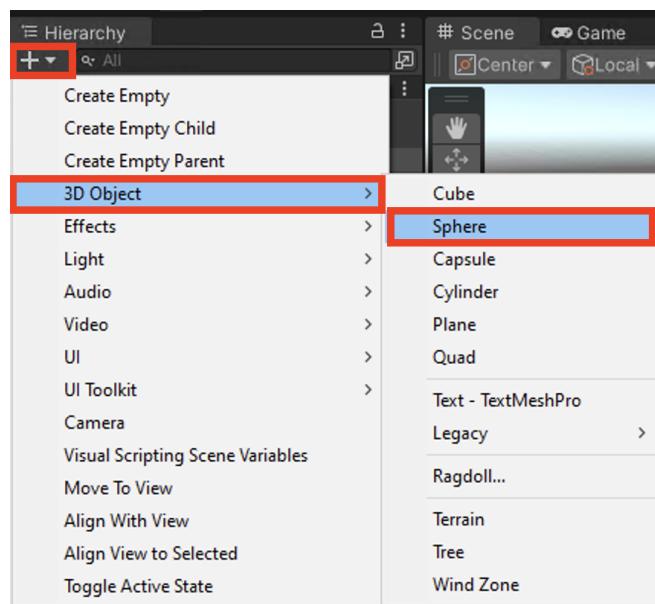
PRO TIP!

Regardless of the tool you're utilizing, navigating within the scene panel is easy using the mouse. Simply use the **mouse wheel** to zoom in or out of the scene. Furthermore, holding down the **right mouse button** and dragging allows you to move to the specific area of the scene you wish to interact with.

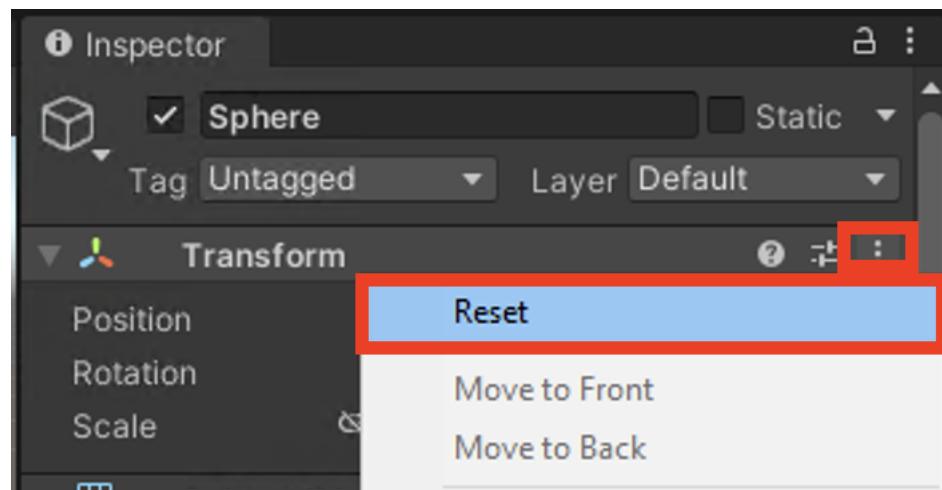
- 12** After you are done with the cube in the scene, it's best to return it to the middle of the scene at 0, 0, 0. With the cube selected, find the **Transform** component at the near top of the Inspector panel. This component shows the current position, rotation, and scale of the **GameObject**. A quick way to reset the transformations is to click on the three-dot menu in the upper right corner of the **Transform** component and then click on **Reset**. This should restore the cube's default settings.



13 Now to add another **GameObject** to the scene. You can follow the same procedure as in the previous steps, but here's an alternative approach. In the **Hierarchy** panel, click the **Plus** button in the upper left corner and then click **3D Object**. Finally, select **Sphere** so we won't confuse it with the cube.

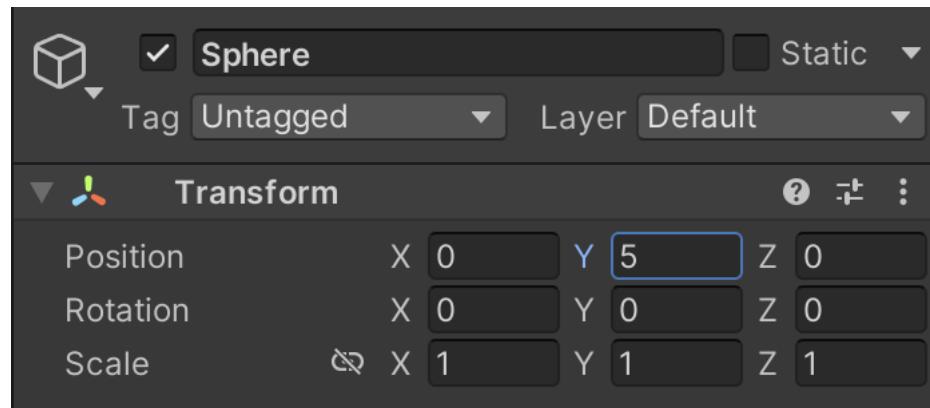


14 Let's make sure that the sphere starts out in the same spot as the cube. Use **reset** if necessary.



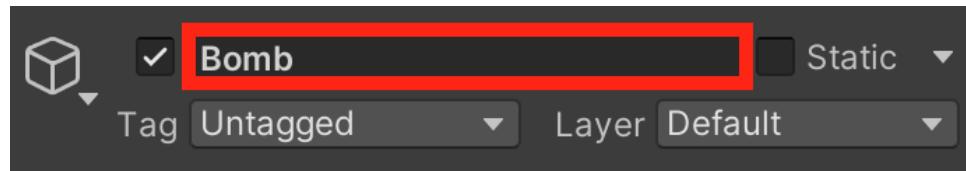
15

We want the sphere to start out above the cube. You can either use the move tool to move it in the scene or change the Y position of the sphere from 0 to 5 in the **Transform** component in the **Inspector** panel.

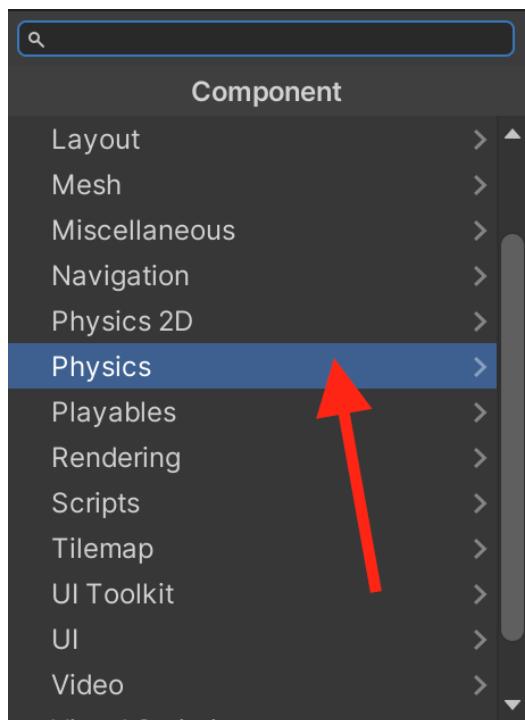


16

This game is called Dropping Bombs, not Dropping Spheres! Change the name of the sphere to **Bomb** by highlighting the name in the **Inspector** panel and typing in the new name.



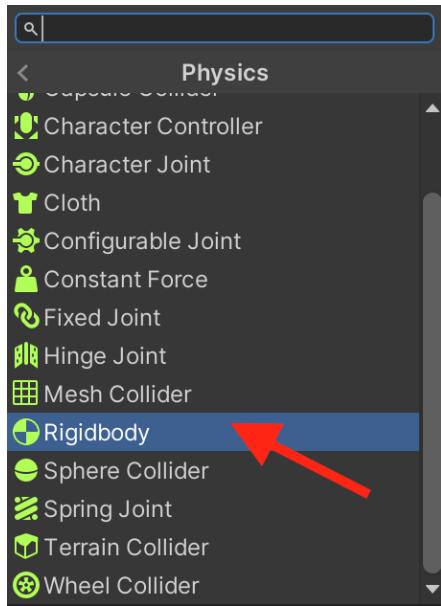
17 If you started the game now, not much would happen. Both game objects are just floating in space. If you want the bomb to drop, you'll need to have it respond to gravity. To do that, you'll need to add a new component to the object. With the bomb selected in the hierarchy, scroll to the bottom of the **Inspector** panel and click **Add Component**. This opens a menu of all sorts of new components and behaviors that can be added to the game objects. The component that you'll need is in **Physics**, so select that section.



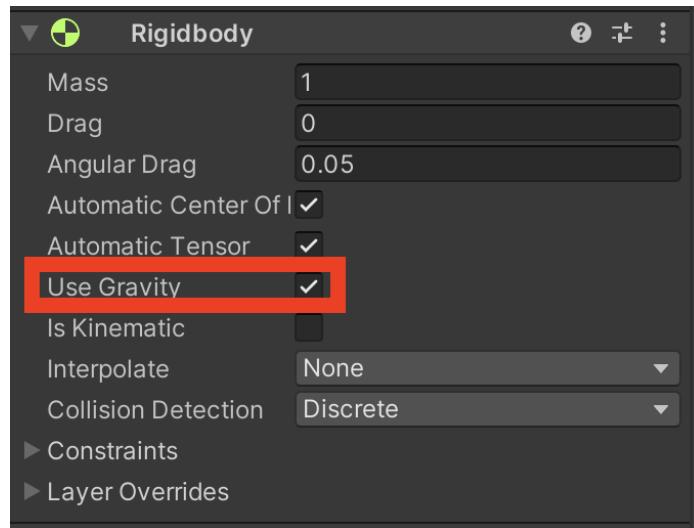
PRO TIP!

There are two choices for **Physics** in the **Add Component** menu. **Physics 2D** responds a bit differently than **Physics**. Therefore, make sure that you choose **Physics** for this activity!

18 A submenu of **Physics** related components should open. These components are useful for many different sorts of behaviors. The one we want is called **Rigidbody**. The **Rigidbody** component allows the **GameObject** to behave like any average object in the real world. Select it to add this component to the bomb object.



19 The **Rigidbody** component has many parameters that can be adjusted to get the **GameObject** to respond to forces inside the game. Right now, we just want to have it respond to gravity, so make sure that **Use Gravity** is checked.

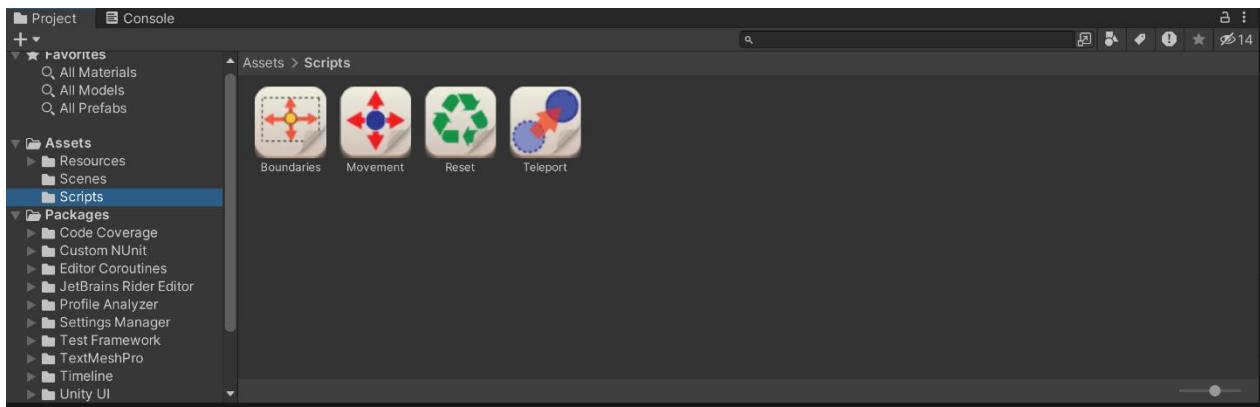


20 Test the game by clicking the Play arrow above the **Scene** panel. The Bomb should fall until it touches the cube and then it stops. When a 3D object like a cube or a sphere is added, a special component called a **Collider** is added with it. Without the Collider, the object wouldn't even know the other object is there and move right through it!

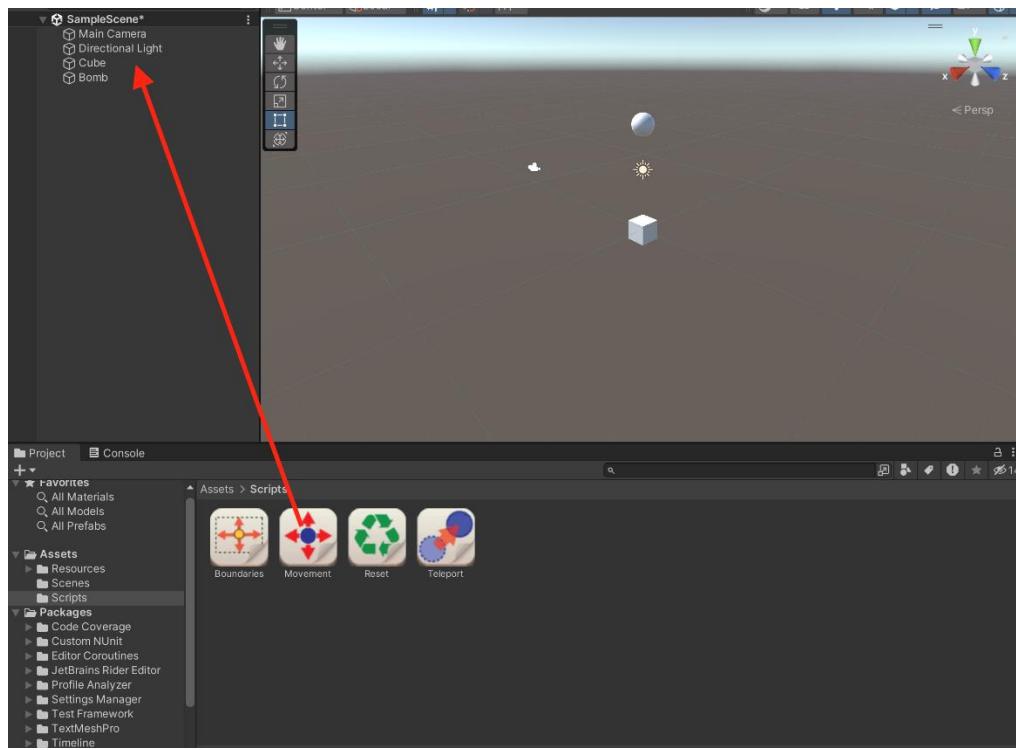


Stop the game by clicking the **Play** arrow a second time.

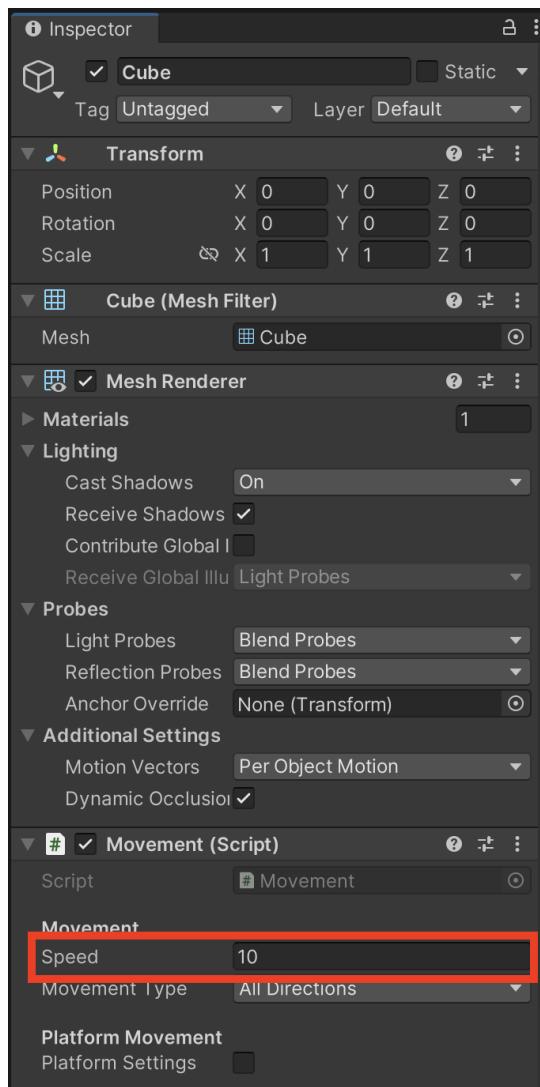
21 The objective of this game is to avoid falling bombs by moving the cube. To move the cube, we'll need a special set of instructions called a **script**. You loaded some scripts when you imported the unity package. To see them, click on the **Assets** folder in the **Project** panel then click **Scripts** to open the folder.



22 Attach the **Movement** script to the cube by simply dragging it into the Cube object in the **Hierarchy** panel.

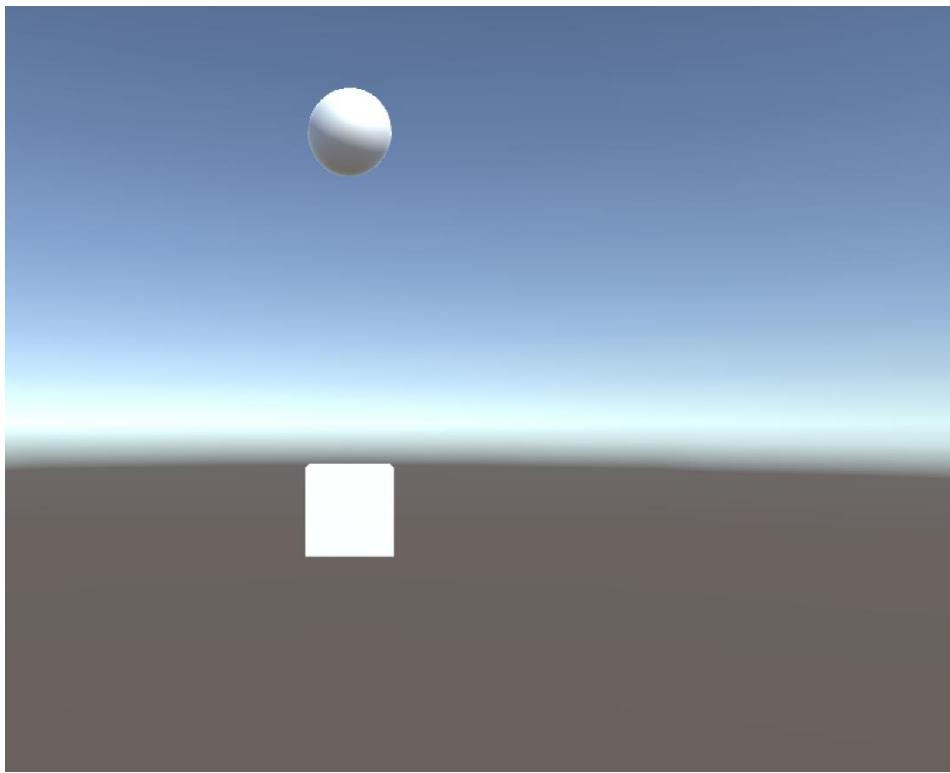


23 Make sure the cube is selected in the hierarchy panel. You will find the **Movement** script is now part of the components attached to the cube in the **Inspector** panel on the right. By default, the script has a movement speed of **0**, meaning the cube won't move at all. Let's try a movement speed of **10**. Leave the other parameters unchanged.



24

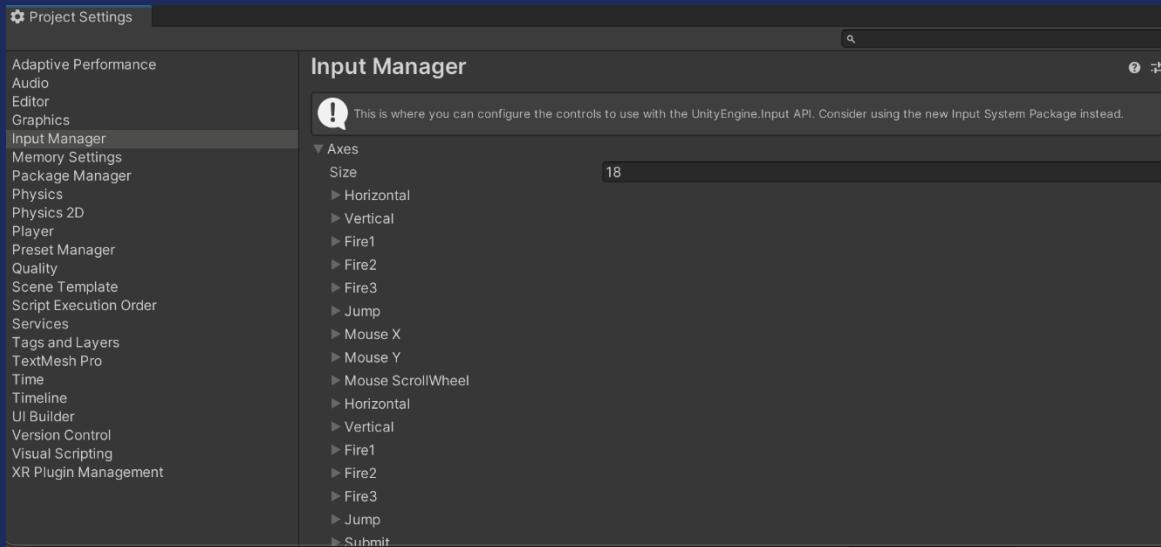
Play the game. Use the **arrow** keys or the **WASD** keys to move the cube around. Stop the game by clicking the **Play** arrow a second time.



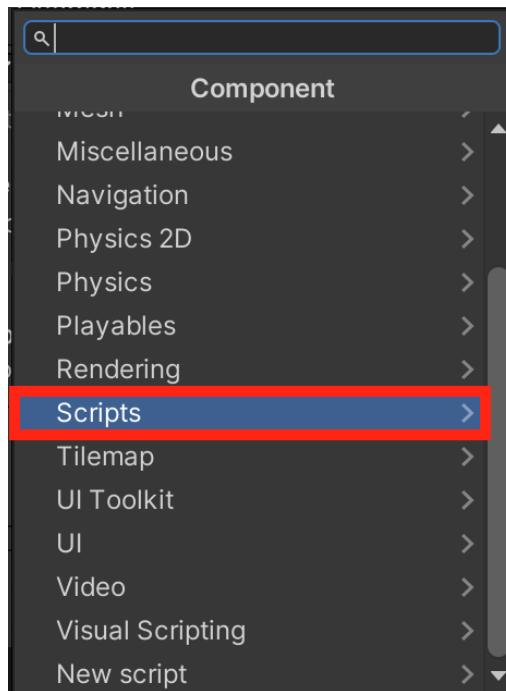
PROJECT SETTINGS

Every project in Unity has a variety of settings that can be configured for whatever situation that you need them for. One of these settings is Input, controlling how the user interacts with the project. Every new project starts out with default settings for input with room for more if needed.

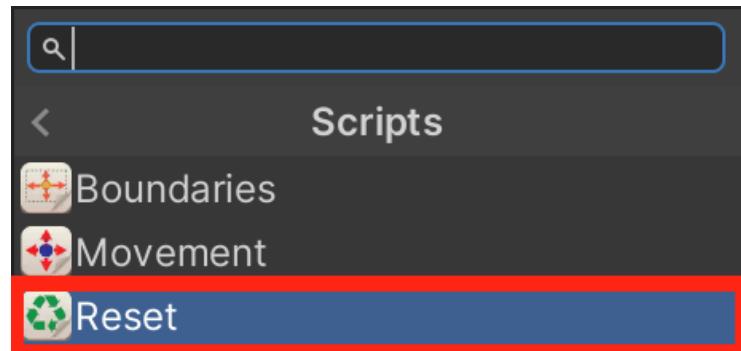
Our provided movement script takes advantage of those settings so that the game can accept input from the arrow keys as well as WASD keys and even a gamepad if one is available!



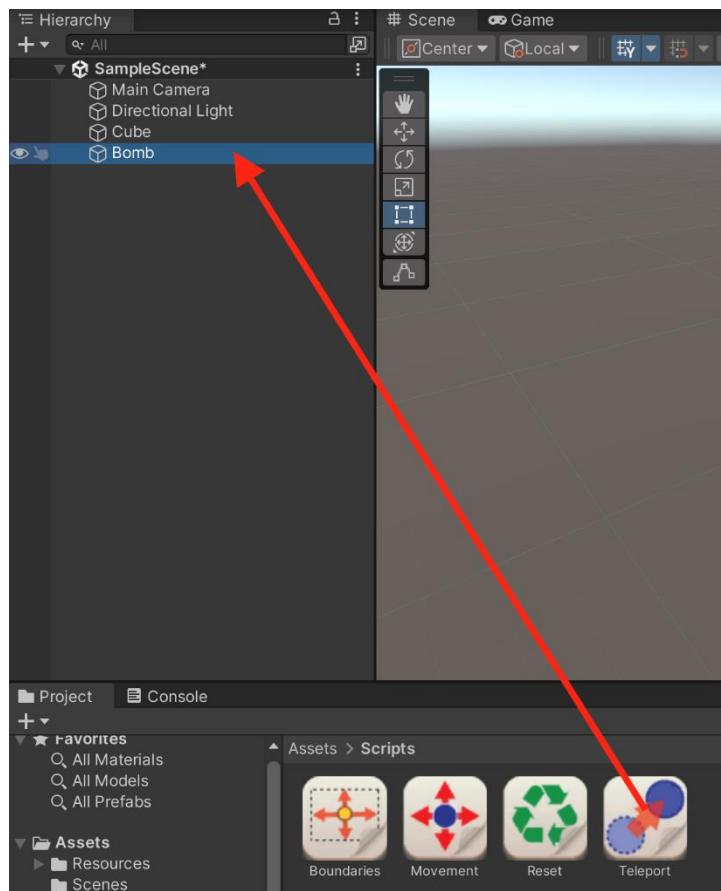
25 To make things more interesting, let's have something happen if the bomb touches the cube. We have a script called **Reset** that will start the game over again if anything collides with the **GameObject** the script is attached to. We want to attach this script to the bomb. Instead of dragging the script over like we did with the movement script, here's another way. Select the bomb in the **Hierarchy** and in the **Inspector** panel, click **Add Component** and select **Scripts**.



26 The menu will change to show the available scripts. Click **Reset** to add the reset script to the bomb.

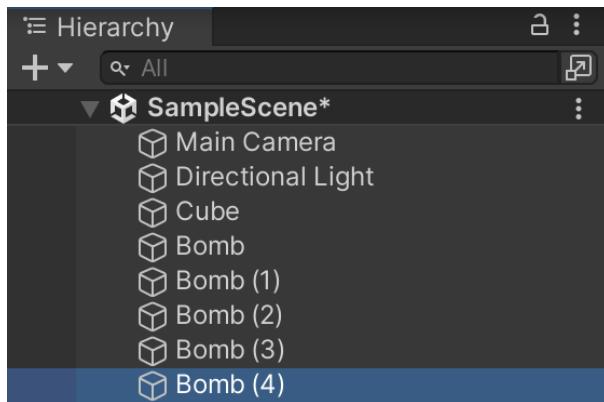


27 Of course, if the bomb doesn't touch the cube, it will just keep falling. We have a script to fix that, too. The **Teleport** script checks to see if the attached **GameObject** has gone below the bottom edge of the screen and if so, places that same object at the top of the screen to let it fall again from a new position. Add the **Teleport** script to the bomb either by dragging it over or by using **Add Component**.



28 To make things more interesting, let's add a few more bombs. Select the bomb in the **Hierarchy** panel and press **Ctrl+D (Windows)** to make a copy. Start with a few, test it, and then add some more until it feels right to you.

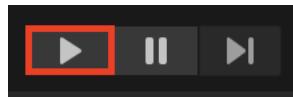
Once you have added some more bombs select your Cube in the **Hierarchy**. In the inspector panel select the **Tag** dropdown menu. In this menu select **Player**.



PRO TIP!

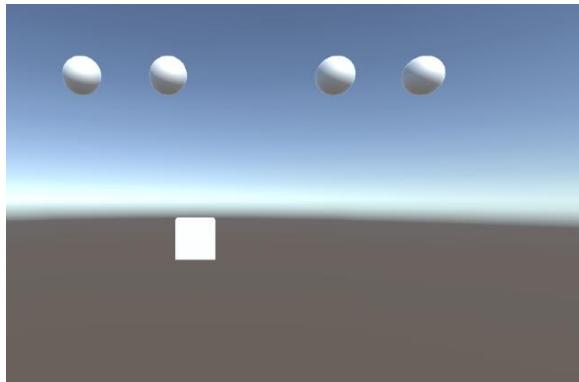
Remember, if you make changes to one of the bombs, you'll need to make changes to the rest of them or delete the other bombs and copy the new one. This isn't very efficient, is it? You'll learn a better way later.

29 Start the game. Use the **arrow** keys or the **WASD** keys to avoid the bombs. If the bomb hits the cube, the game resets to the original start settings. The **Teleport** script is now placing the bomb in random positions on the screen.

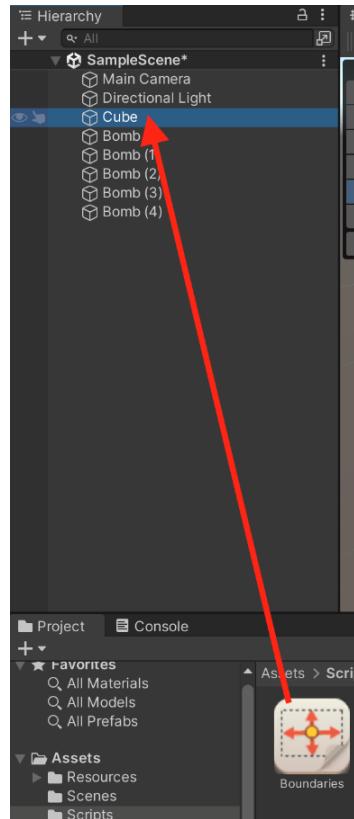


Stop the game by clicking on the **Play** arrow a second time.

30 Now when you start the game, many bombs come from different positions to attack your cube!



31 But there's one last thing to add to the game. At the moment, the cube can avoid all the bombs by leaving the screen! The **Boundaries** script keeps the cube inside the screen. Add the script to the cube to keep it in place using either option learned in this activity.

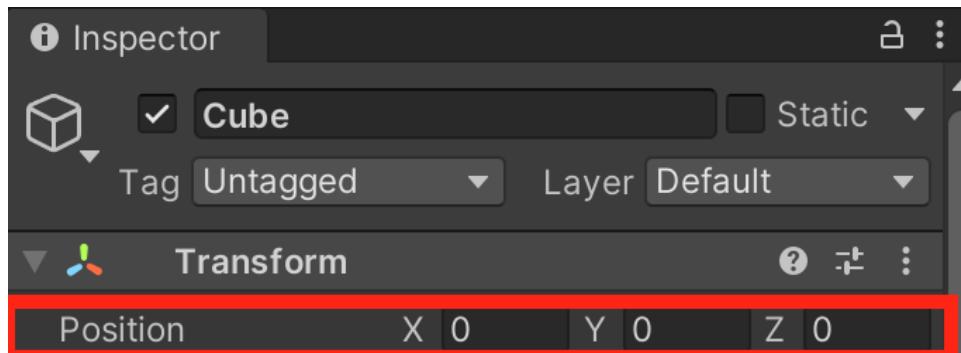


You just made your first game in Unity! Way to go! Later, you'll add some graphics and effects to make it look better, but the best games are fun even before all the graphics are added.

Adjusting the Cube

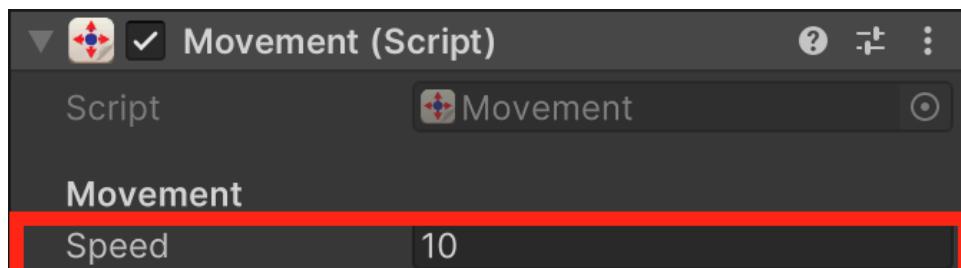
Starting Position

Do you have enough time to dodge the bombs when the game starts? You can move the cube lower in the screen by changing the Y value for the Transform position.



Movement Speed

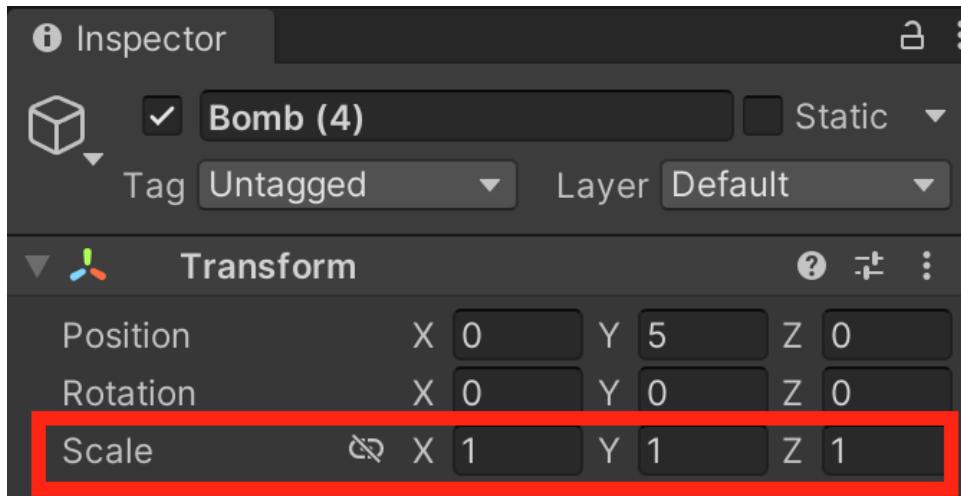
Is the cube not moving fast enough? Increase the number for the movement speed to make dodging easier. Just don't make it so fast that you can't control the cube!



Adjusting the Bomb

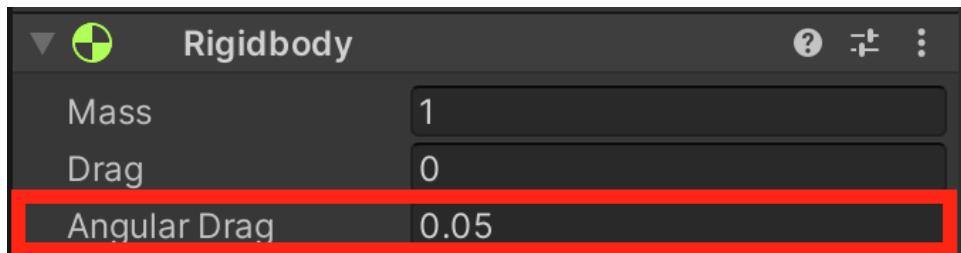
Size

Would a smaller target be easier to dodge? Change the scale in the X, Y and Z parameters to a fraction of 1 to make the bomb smaller.



Drag

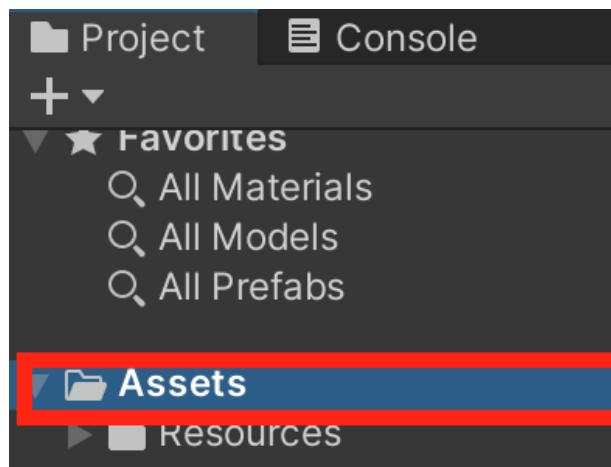
Change the Rigidbody settings of the sphere so that there's a bit of drag to slow it down when falling. Start with small numbers and work your way up.



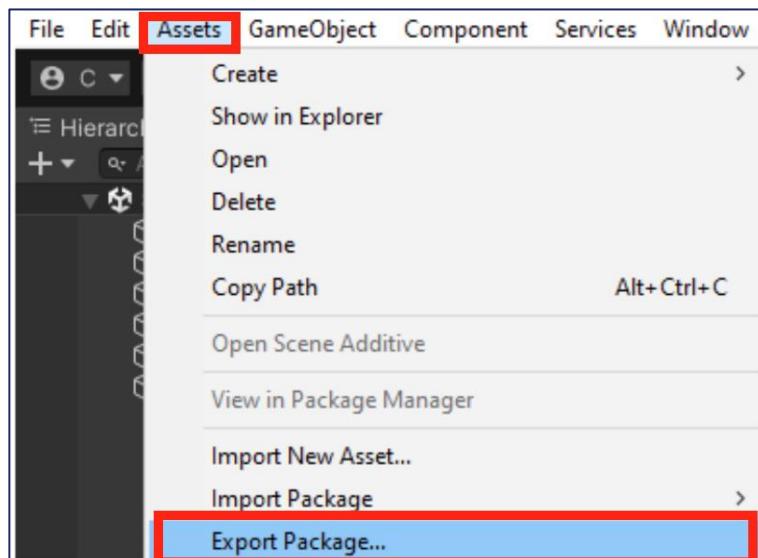
Don't Forget! When you change the parameters of one bomb it won't change any of the other! Either apply the same changes to all the bombs or delete the unchanged bombs and make copies of the changed one.

Saving and Submitting Your Games

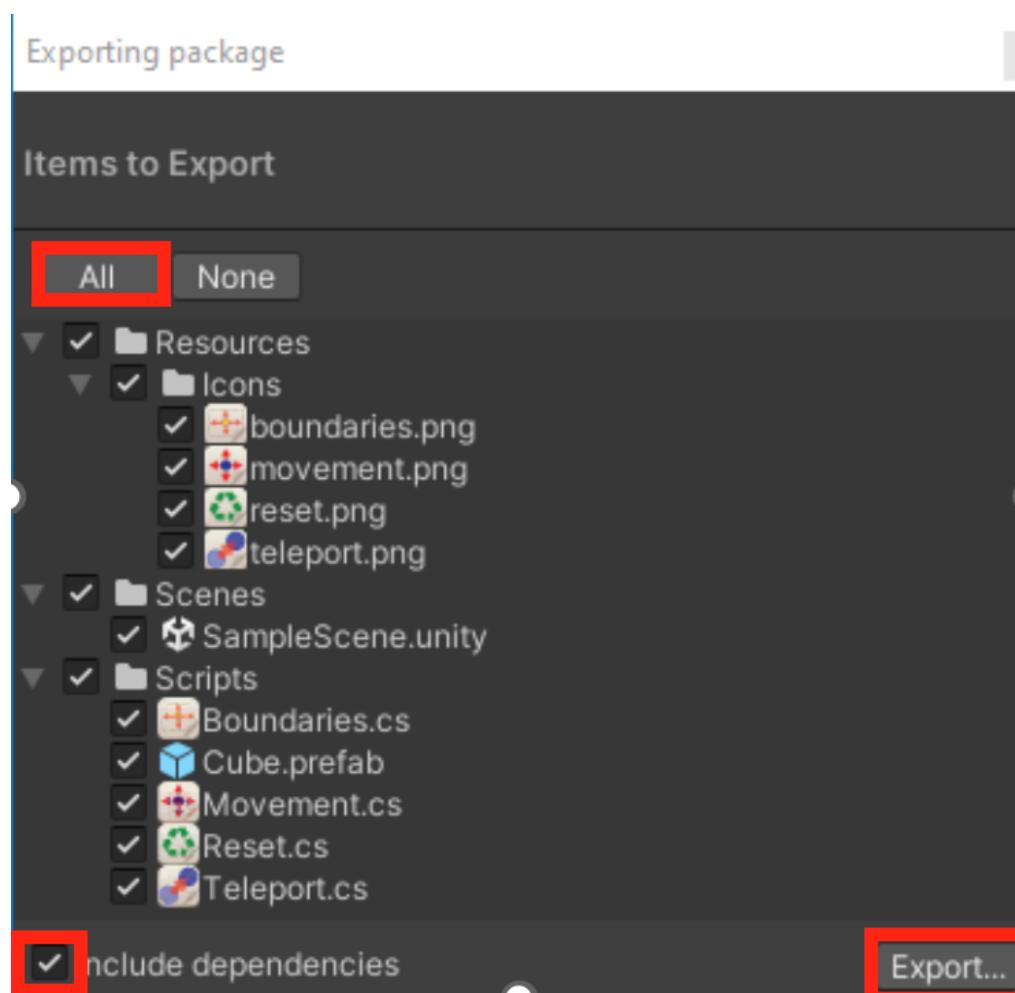
- 1 When you save a game in Unity, that project is saved on a folder in your computer for you to open with Unity and resume work on it. However, we need to get it to your Code Sensei so they can grade your game!
- 2 In the **Project** folder, click on the **Assets** folder. We do this so that Unity knows we are about to export all the assets and different components that make up your game.



- 3 We are going to export your game and it will create something called a **Unity Package**. First, click on the **Asset** button in the top left corner then navigate to **Export Package**.



- 4** Once **Export Package** is selected, the **Items to Export** Menu will appear. Ensure all your game components are checked and the **Include dependencies** checkbox is clicked before clicking on **Export** at the bottom-right.



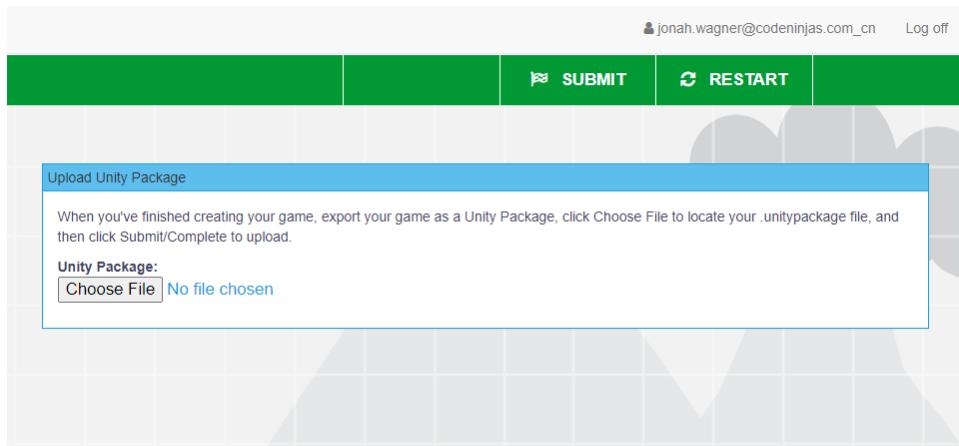
5 Once **Export** is selected, a window will appear asking you where you want to save your game and what you want to name it. This is called a **File Explorer window**. Use a file name that clearly describes your game, such as the Project's name, and put it into a folder that you can find easily.

In this example, we've put it into a folder named "**JWUnityGames**" and named the file "**JWDroppingBombs1**". Your names should include your initials to be a bit more specific!



It may take a minute for the export to finish so just wait until it is complete. Once it is complete you can move onto the next step!

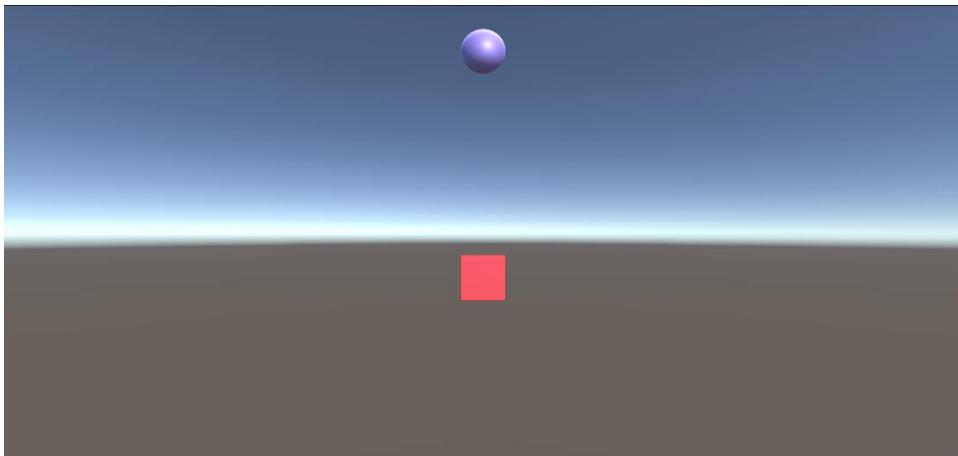
- 6** Select the activity you just finished in the paths page on the GDP. Open it and you should see an area to submit the .unitypackage file you just made!



Select the Choose File button, then find the .unitypackage file you just made. Double-click it to add it to the Dojo. Once you have added the file, click the Submit/Completed button next to the Save button and wait for your file to upload. You will see a message informing you that the file has been uploaded in the green bar at the top of your screen.

Once you do this your game is now submitted and ready for grading! You will submit each of your activities like this, so just navigate back to this page if you ever forget how.

Prove Yourself: Color Drop



For this first Prove Yourself, Color Drop, we will change the color of the sphere and the cube. This can be done entirely through the Unity interface, without ever going into any code! Open your Dropping Bombs Part 1 project. Can you figure out how to do the following:

- 1. Create a folder called "Materials" in the "Assets" folder under the "Projects" menu.**
 - *Think about where you would organize assets in Unity. Where can you create a new folder?*
- 2. In this folder, create 2 materials called SphereMaterial and CubeMaterial.**
 - *Consider where you'd create new assets like materials in Unity. How can you create new material? Where might you find this option?*
- 3. Give each of these materials their own color.**
 - *Once you've created materials, think about how you can customize their appearance. What properties do materials have that you can adjust? Where might you find these properties?*

4. Color the cube and sphere objects in your scene with these materials.

- *When it comes to changing the appearance of objects in your scene, think about what components or settings might be involved. How can you assign a material to an object? Where might you find this option?*

During this Prove Yourself take some time to look around the Unity interface, and don't be afraid to experiment!

Importing Assets

Unity is a project-based program. This means that while Unity is very good at helping you work with assets to make some great-looking games, it is less concerned about how those assets are created. In the end, this approach works out the best since there are plenty of places out there to make and find parts to use in our games, anything from 3D models right up to the fonts.

In this section, you will learn about some of the places to get these assets and how you can add them to Unity to use in a project!



Fantastic Assets (and where to find them)

In the previous activity, you imported a unity package into your project. This is a great way to transfer content made in one Unity project into another. It is a very common way of transferring content in Unity. It's also a great way for making backups of what you're currently working on. Since Unity is a project-based program, a lot of what you use in your games will come from outside of Unity. These include:

Graphics (Sprites, textures and other similar images.)



3D Models

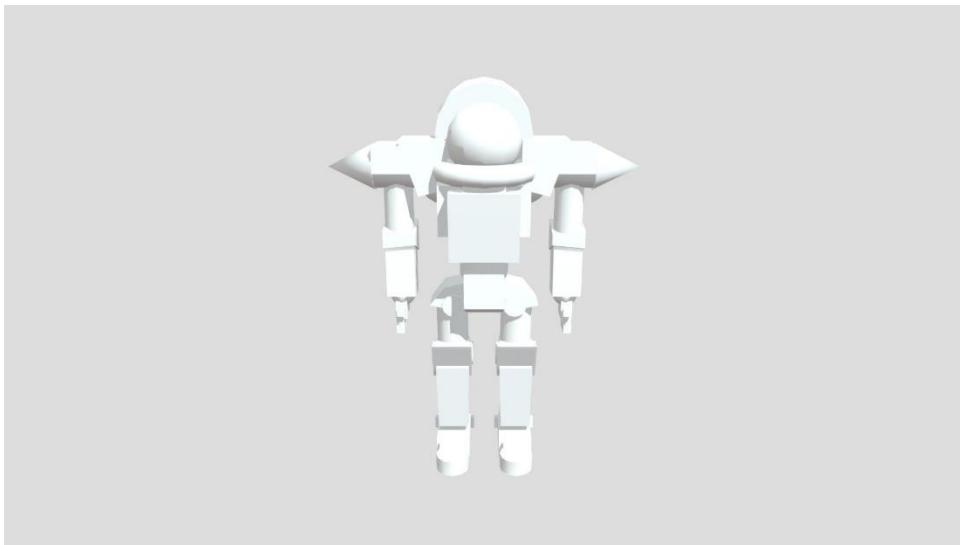
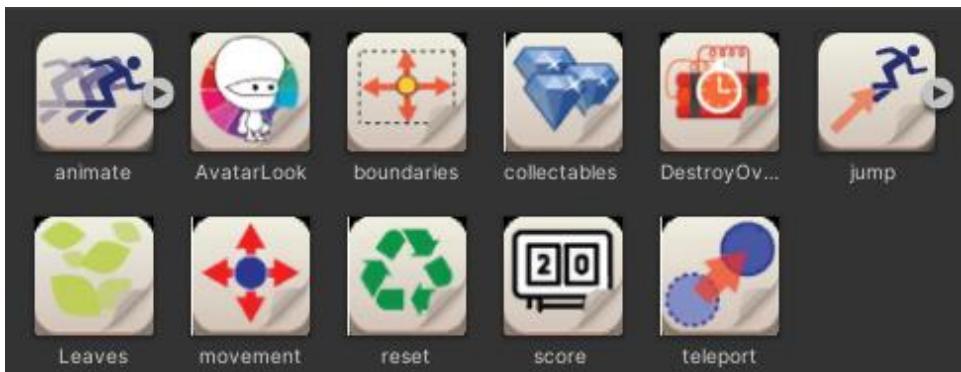


Image Source: <https://sketchfab.com/3d-models/space-marinebot-v1-without-texture-b8a744ebcd014c6aa532eab5b17b8ef4>

Scripts, fonts, audio files, and other files.

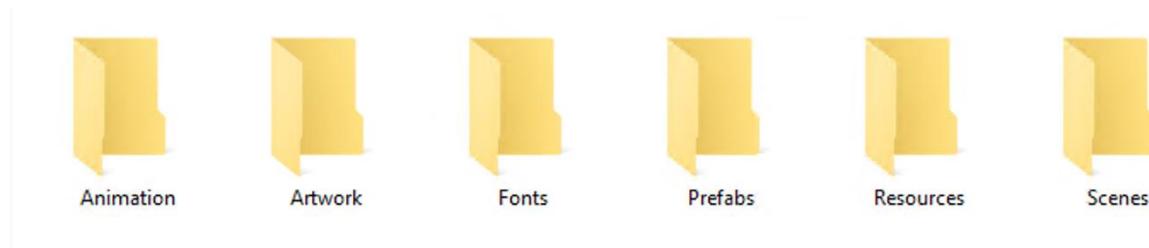


Importing Assets into Unity

There are several ways to import assets into Unity. We'll cover a few of the ways that you can import. Which way you use is entirely up to you.

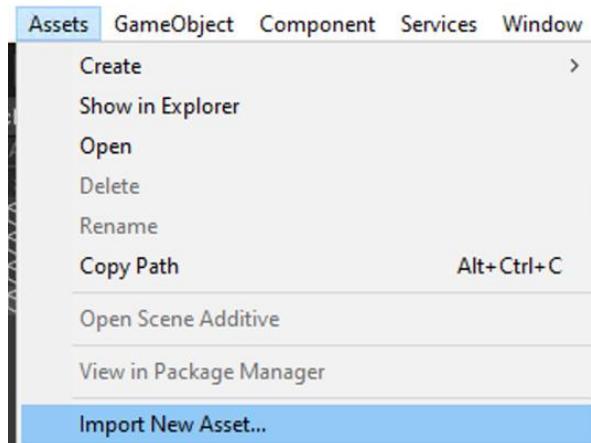
Windows File Explorer

With the Windows File Explorer, you can copy files and folders directly into the Assets folder of your project. Unity has technology built in that allows it to automatically recognize the files and sort them based on their type.



Unity Asset Importer

You can either click on the **Assets** tab at the top or right-click in the **Assets** panel to open the menu and select **Import New Asset** to select the files that you want to bring into your project.



Drag and Drop

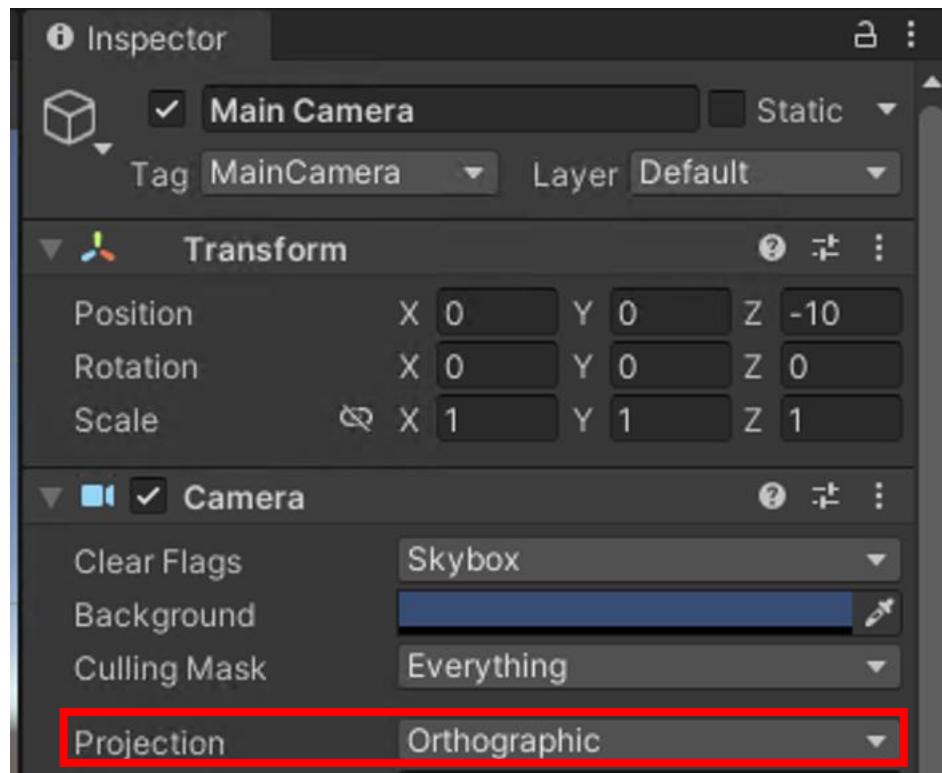
Another option for importing assets is opening a Windows File Explorer window with the files that you want to import and drag the files directly into the **Asset** panel in Unity.

Name	Date modified	Type	Size
caveback1.png	2/13/2024 9:34 AM	PNG File	12 KB
caveback2.png	2/13/2024 9:34 AM	PNG File	25 KB
clouds.png	2/13/2024 9:34 AM	PNG File	53 KB
DBTitle.png	2/13/2024 9:34 AM	PNG File	193 KB
debrisParticle.png	2/13/2024 9:34 AM	PNG File	2 KB
gamelicons.png	2/13/2024 9:34 AM	PNG File	80 KB
grungeHazard.png	2/13/2024 9:34 AM	PNG File	52 KB
JackSpriteSheetHead.png	2/13/2024 9:34 AM	PNG File	789 KB
JackSpriteSheetIdle.png	2/13/2024 9:34 AM	PNG File	519 KB
leaf.png	2/13/2024 9:34 AM	PNG File	2 KB
ScavengerHuntBackground.png	2/13/2024 9:34 AM	PNG File	155 KB
skull.png	2/13/2024 9:34 AM	PNG File	15 KB
SkyGradient.png	2/13/2024 9:34 AM	PNG File	6 KB
stalagmite_1.png	2/13/2024 9:34 AM	PNG File	3 KB

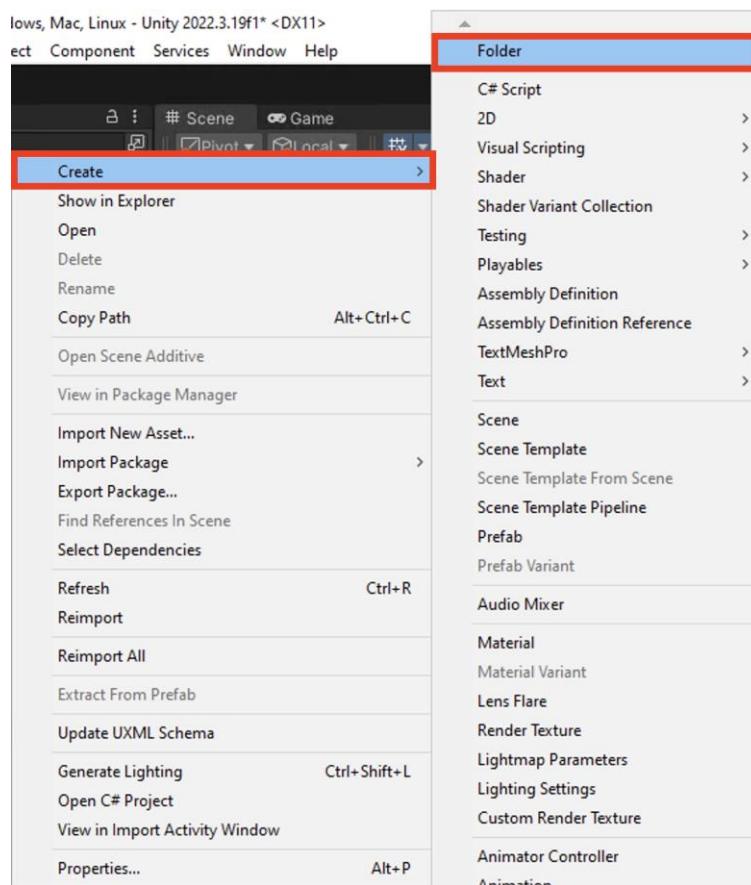
Activity 2: Scavenger Hunt

The previous activity was meant to serve as a quick introduction to working with Unity. This time, you'll build a simple platform game. To start, open Unity and create a **new** 2D project. Give it a name like **YourInitialsScavengerHunt**. For example, if your name was John Ninja, your project would be named **JNScavengerHunt**.

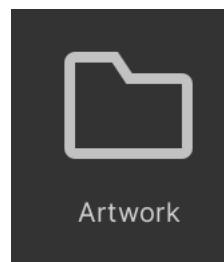
- 1 Depending on your layout settings, your display may not look exactly like this. You are free to work with the layout that you prefer. For this activity, we used the **Default** layout. Refer to the first two steps in the first activity if you are uncertain about how to change the layout.
- 2 Just like the previous activity, we will be using the **2D** camera to edit our scene. Select the **Main Camera GameObject** in the **Hierarchy** panel. If the **Main Camera's Projection** is not already set to **Orthographic** then change the **Projection** from **Perspective** to **Orthographic** as shown below.



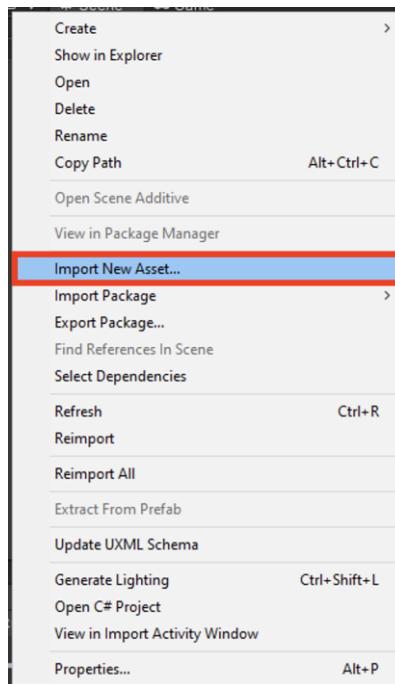
- 3** There will be some artwork provided for you to use in this game. Before adding it to the game, it's a good idea to have somewhere to put it. It's important to have different types of files in different folders in Unity so that you can find them quickly and stay organized. In the **Project** panel, right-click on the **Assets** folder window, click **Create** and select **Folder** to add a new folder to the **Assets** folder.



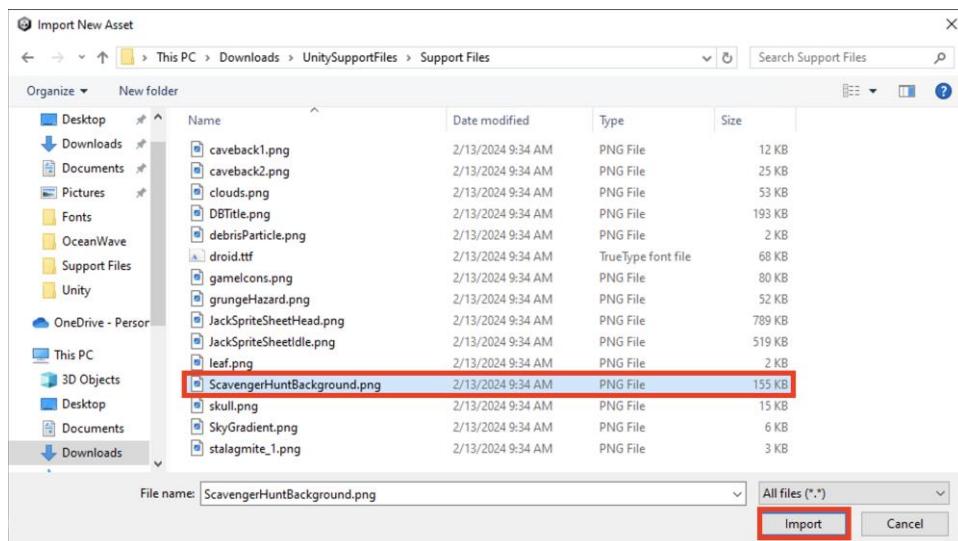
- 4** Rename the new folder to be "**Artwork**".



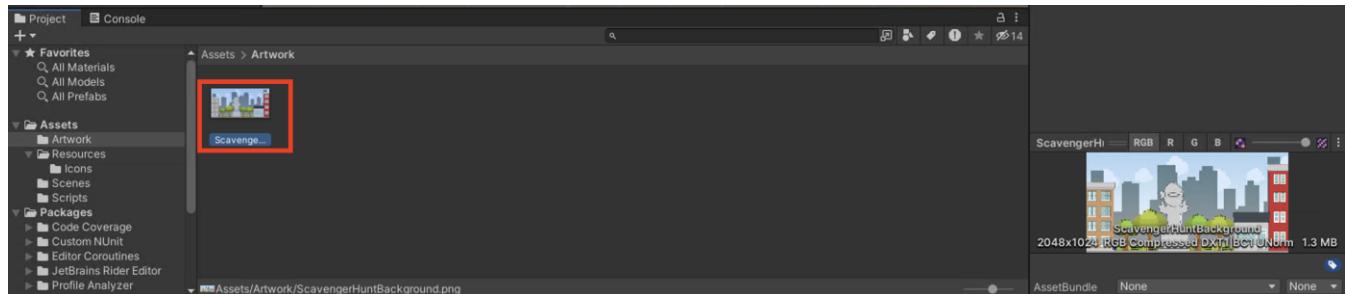
- 5** Open the **Artwork** folder. We are going to import an asset into this folder. There are many ways to import an asset. One way is to right-click in the folder that you want the asset and select **Import New Asset**.



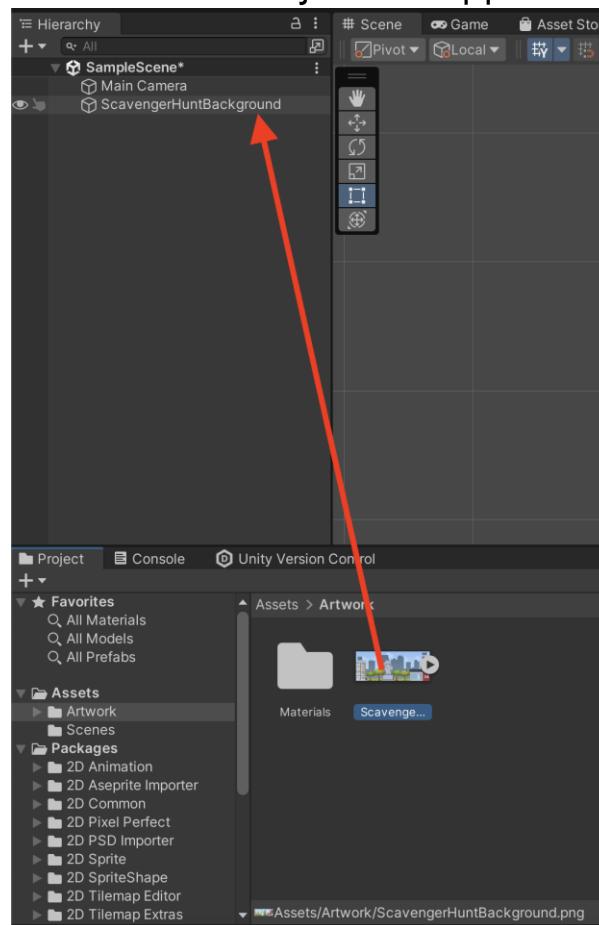
- 6** A new window will open with either File Explorer on Windows or Finder on Mac. Navigate to the folder where Purple Belt assets are located, then click into the folder and select **Activity 02 - ScavengerHuntBackground.png** to import your new artwork.

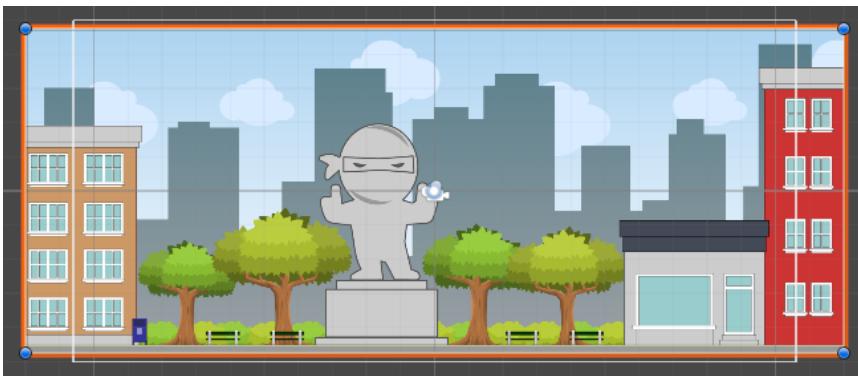


7 Once you select the import button, the new asset will now appear in the **Project** panel. Now that the asset is in the **Project** panel, select it to see all the information about it in the **Inspector** panel.



8 Now let's add the background to the hierarchy. Simply drag the background from the **Artwork** folder and a new object will appear as shown below.

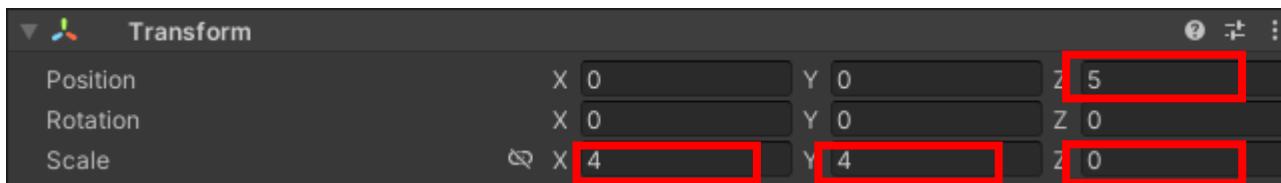




PRO TIP!

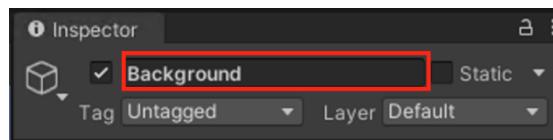
A **Quad** is short for quadrilateral. A **Quad** is like a plane in the fact that it only has four corners, but unlike a plane, it has depth. A **Quad** is ideal for holding simple graphic images as it only has two triangles in its mesh instead of the 200 found in the plane! Since you don't need all those extra triangles, why use them?

9 By default, the **Background** has a size of one unit. However, this is a little small for a background. Let's make it bigger by first selecting the background then increasing the scale by changing the x scale to **4** and the y scale to **4** in the Transform component as shown below. Change the z position to **5**.

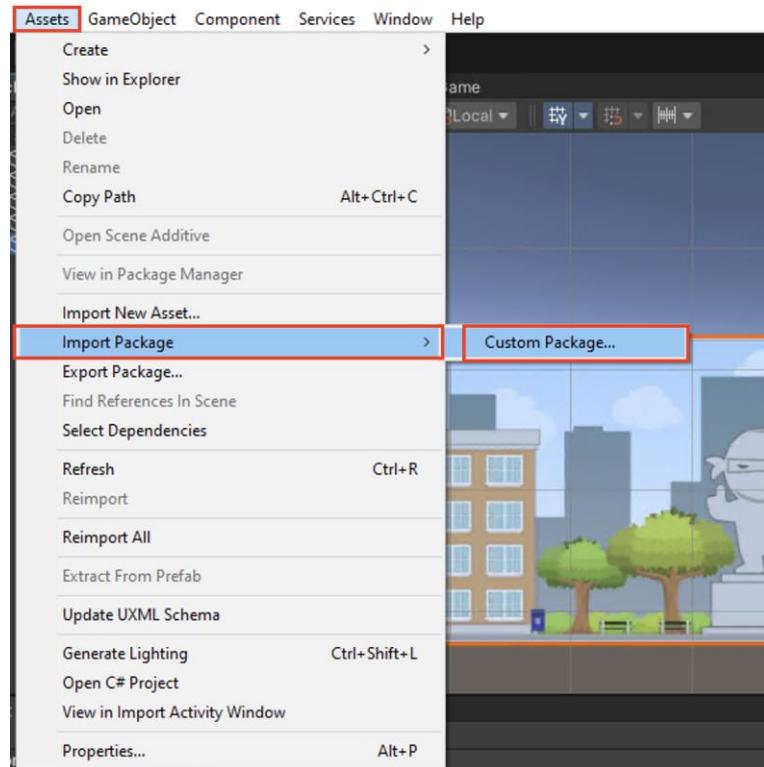


10 Ensure that in the inspector for **ScavengerHuntBackground** that the background is set as the sprite in sprite renderer.

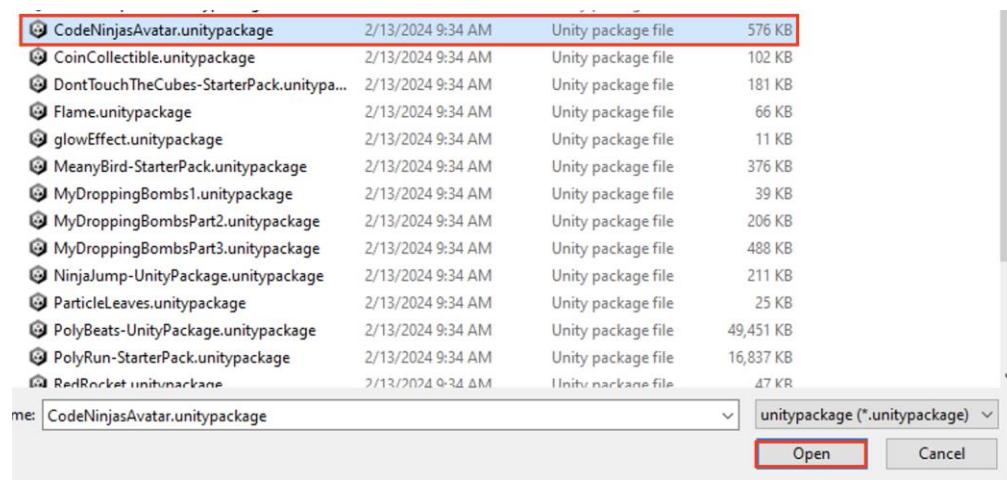
While the background is still selected, change the name from **ScavengerHuntBackground** to **Background**.



- 11** Now that there is a background image, you are ready to add a player character. Code Ninjas has a custom package put together for that. Click the **Assets** tab, select **Import Package** then **Custom Package...**.

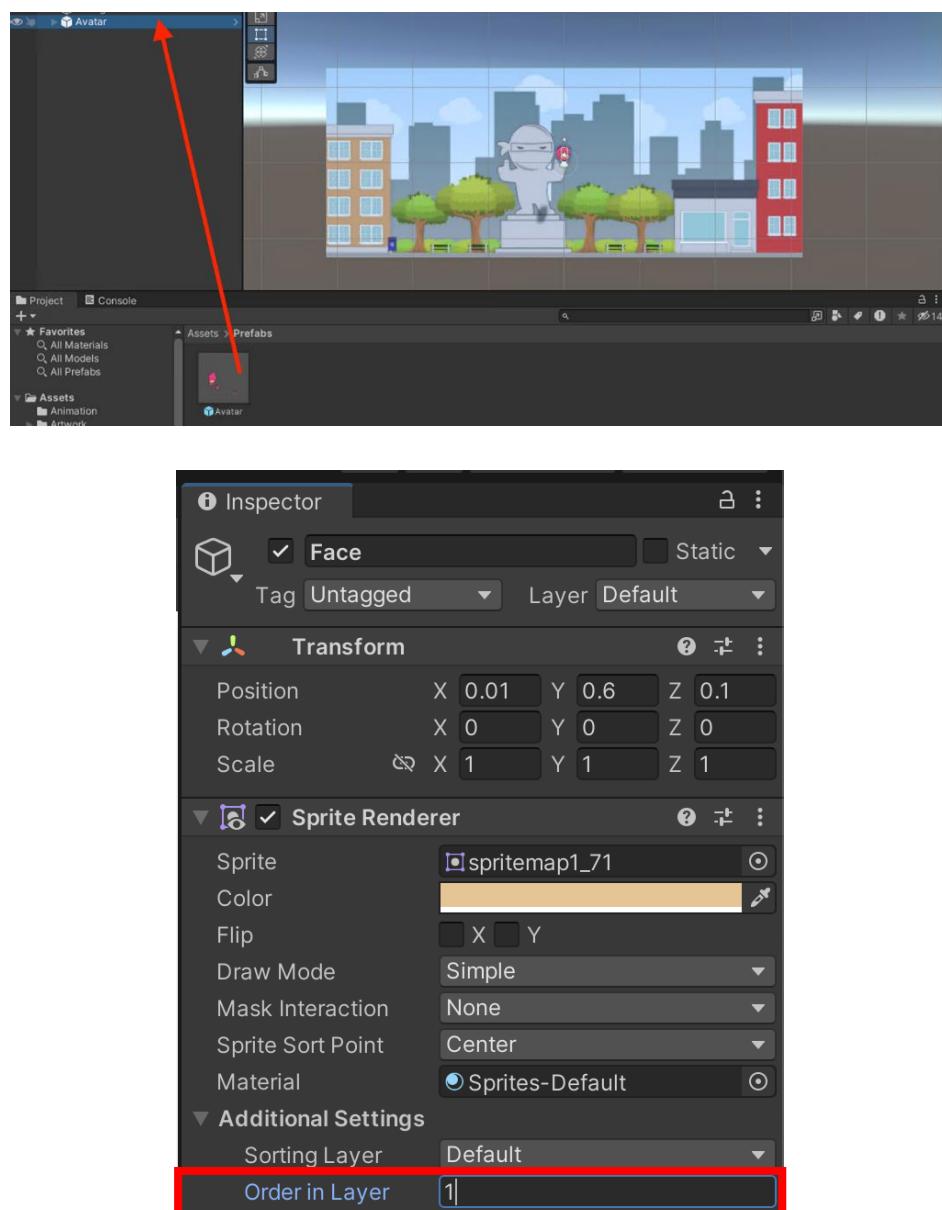


- 12** Navigate to the folder where your Purple Belt assets are located and select **Activity 02 - CodeNinjasAvatar.unitypackage** to import the new player assets.

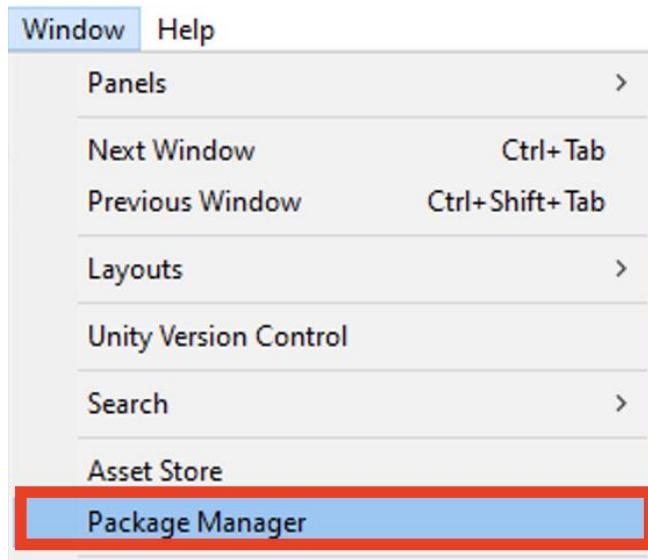


13

The package you just imported includes a new folder called **Prefabs**. Open this folder to find your **Avatar**. To add the avatar to your scene, drag it from the **Projects** panel into the **Hierarchy** panel. If the avatar appears to be missing a body part, select the drop-down arrow to the left of the **Avatar** in the **Hierarchy**. Then, select the missing part and change the **Order In Layer** to a number equal to or greater than 1. In the screenshot below you can see an example of this done for the face since it was not properly appearing. **Prefabs** will be covered in a later activity so for now, know that it is a special **GameObject** with everything it needs.



- 14** In addition to the custom packages, there is another type of Unity package that you can import. These can be found in the **Package Manager**. Open the **Package Manager** by clicking the **Window** tab and selecting **Package Manager**.

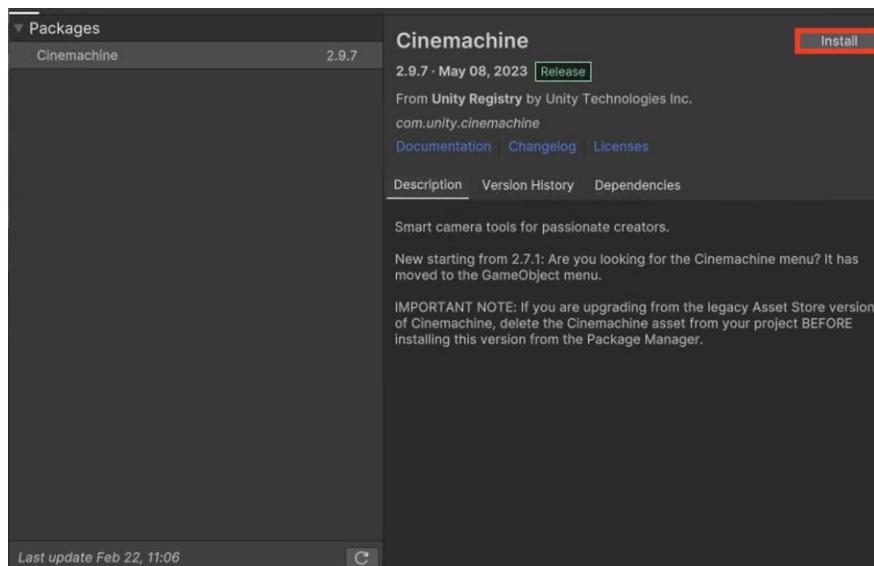


PRO TIP!

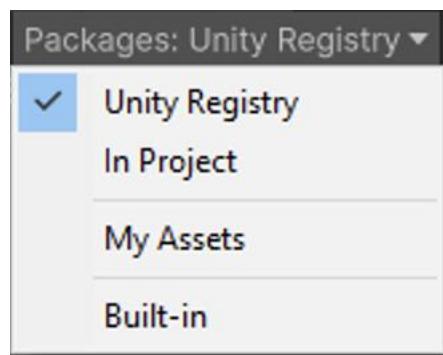
If the package that you are looking for doesn't appear immediately, it may be because Unity is still collecting all the information on available packages. Just give Unity a moment to load all this information.

15

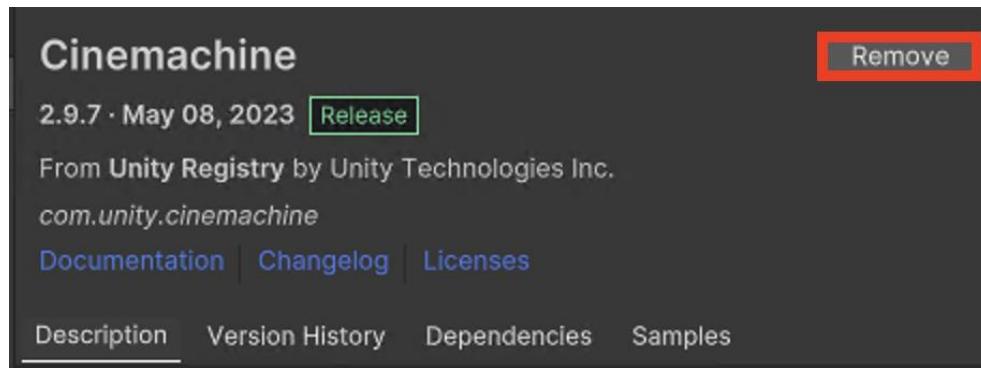
The package that you'll want to add is **Cinemachine**, a set of virtual camera tools that are ideal for the type of platforming game that you will be making. You may need to change the menu appearance from Packages: In Project to Packages: Unity Registry. In the search bar type **Cinemachine** then from the menu select it and click **Install**.



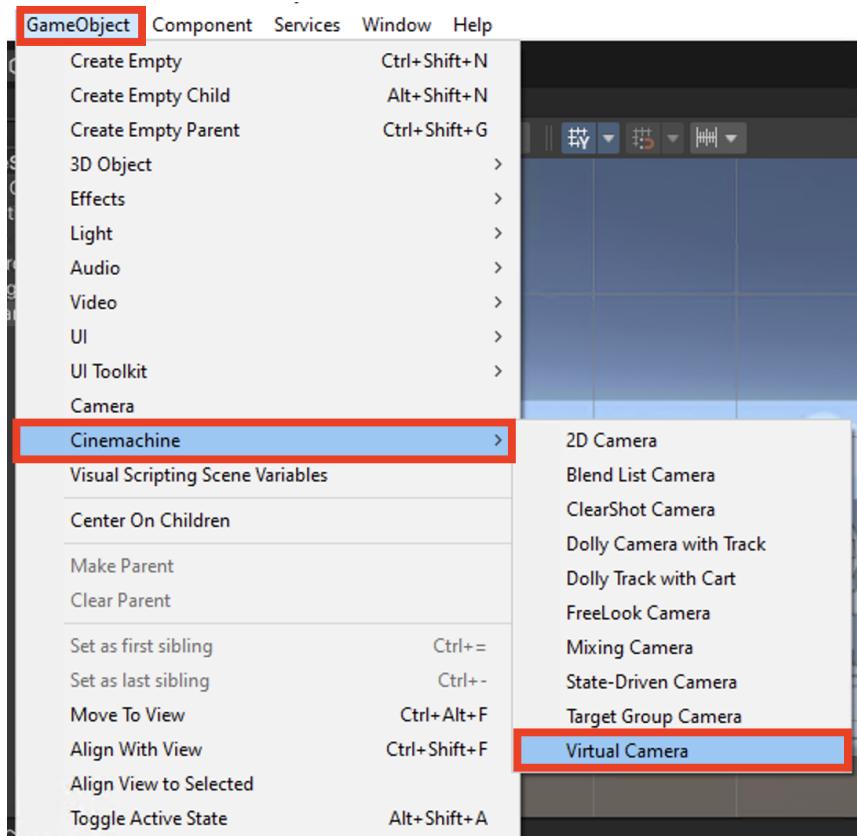
If you cannot see **Cinemachine**, make sure at the top it says **Packages: Unity Registry**, not **Packages: In Project**.



- 16** Wait until the button changes to "Remove" before continuing.



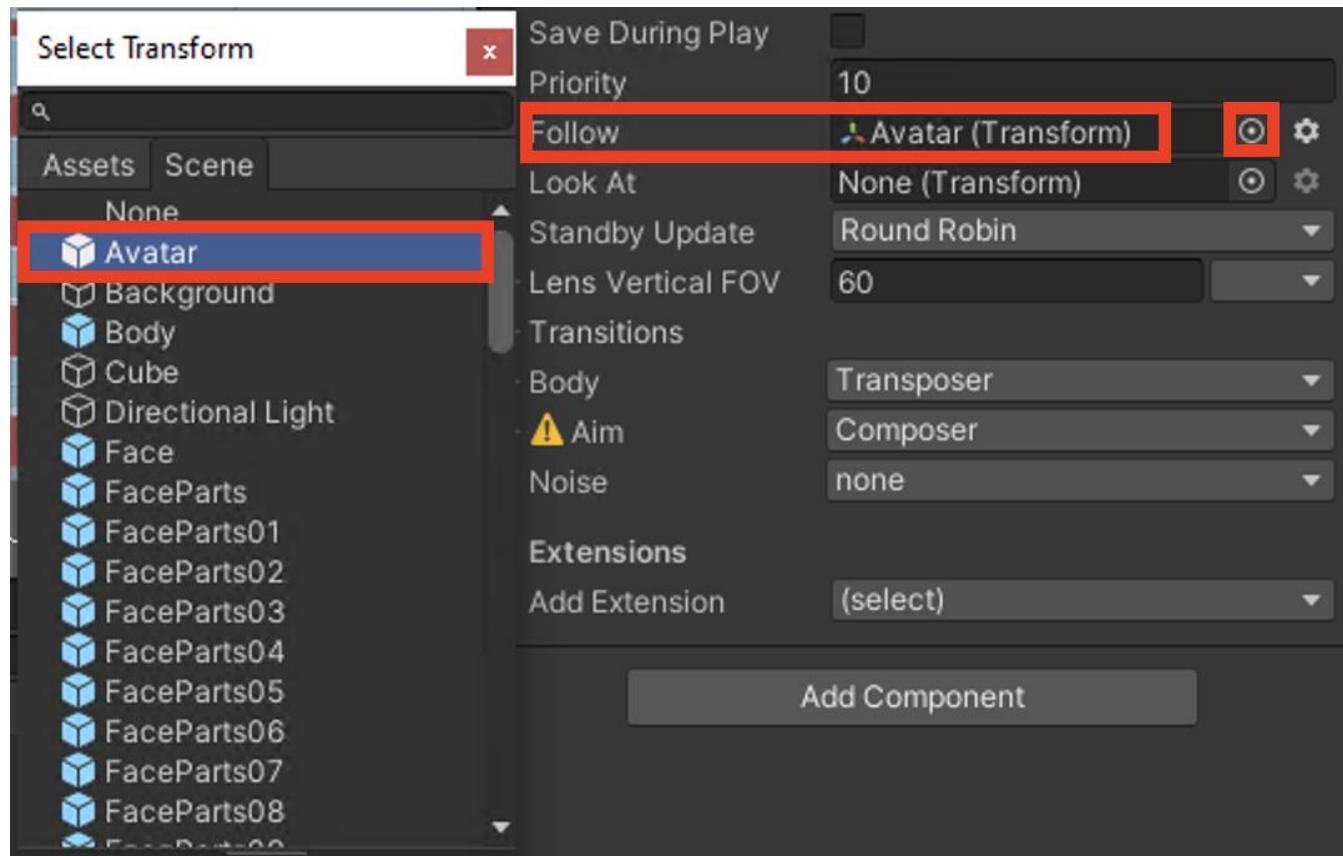
- 17** The package has added a new option in the **GameObject** and **Component** menus called **Cinemachine**. To find it, click on the **GameObject** menu and locate **Cinemachine**. Click on it to see many options for cameras in this menu. You'll want the option that says, "**Virtual Camera**". Click that option to add the **Cinemachine Virtual Camera** to your **Hierarchy**.



18

Click the **Virtual Camera GameObject** in the **Hierarchy** to set it up. In this game, we want the camera to follow the player wherever they go in the scene.

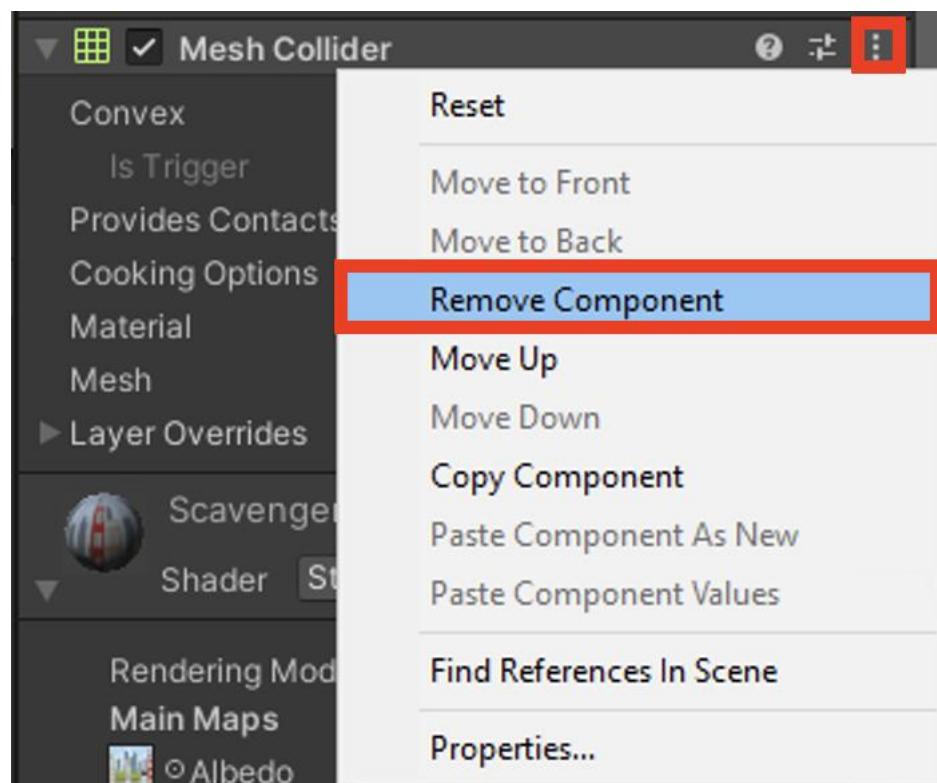
In the **Inspector**, find the **Follow** property and click on the small circle next to **None** to select a target for the camera to follow. This opens a menu of possible **GameObjects**. Click the **Avatar** to select it.



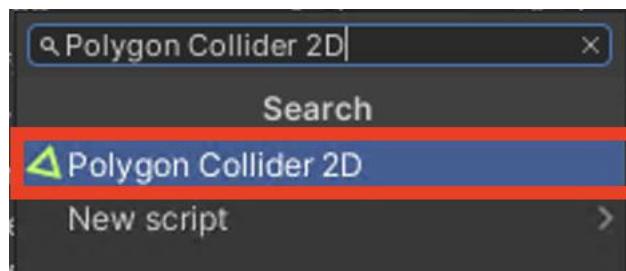
19

The **virtual camera** will now follow the player throughout the scene and beyond. However, the camera shouldn't go past the edges of the background. There needs to be some way to tell the **virtual camera** that the edges of the background are as far as it should go. To do this, you'll need to add a special collider to the background. But before that, we need to remove the current collider.

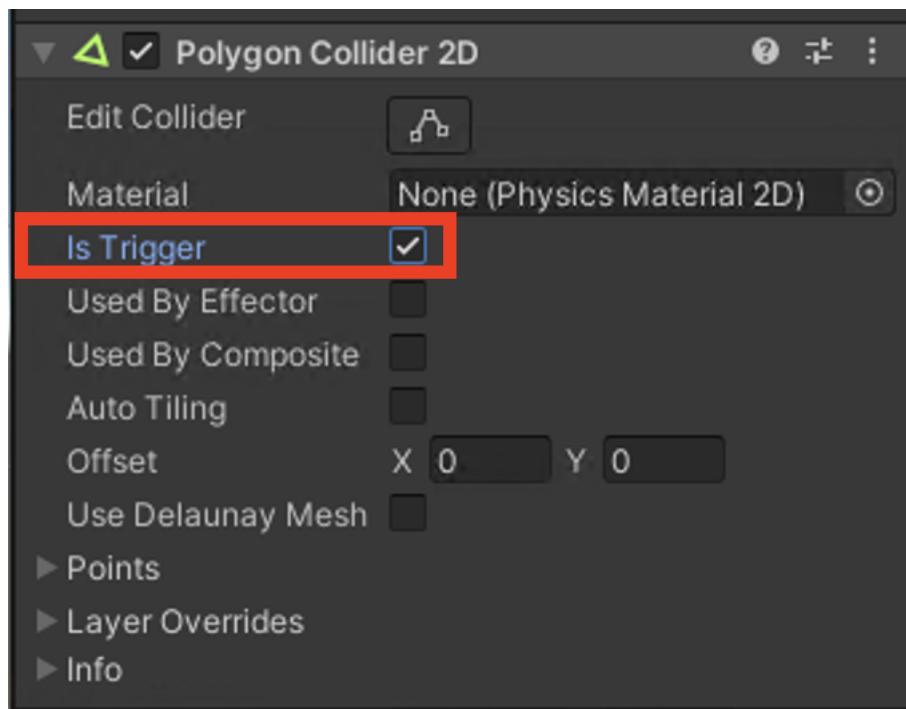
In the **Inspector**, select the **Background** object and find the **Mesh Collider** component. Click the three dots drop down menu in the right corner of the component to open the menu. Click **Remove Component** to remove it from the **GameObject**.



20 To do this we need to add a colider, click **Add Component** at the bottom of the **Inspector**. In the search bar type **Polygon Collider 2D** and then select it.



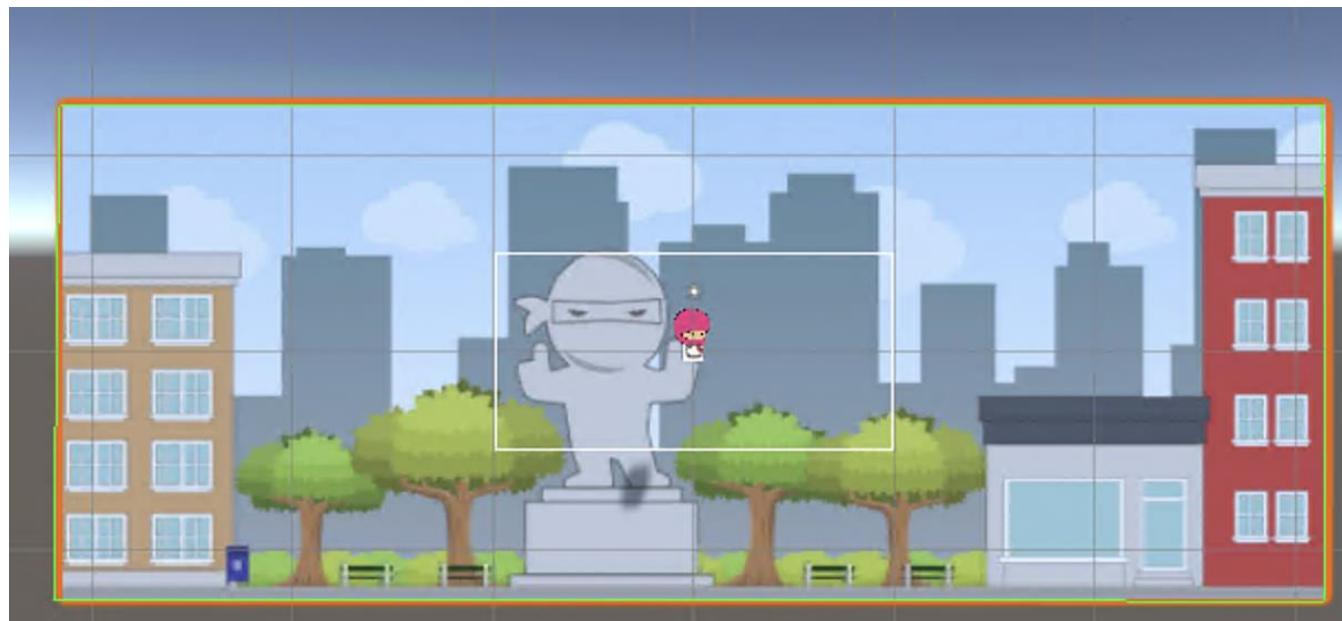
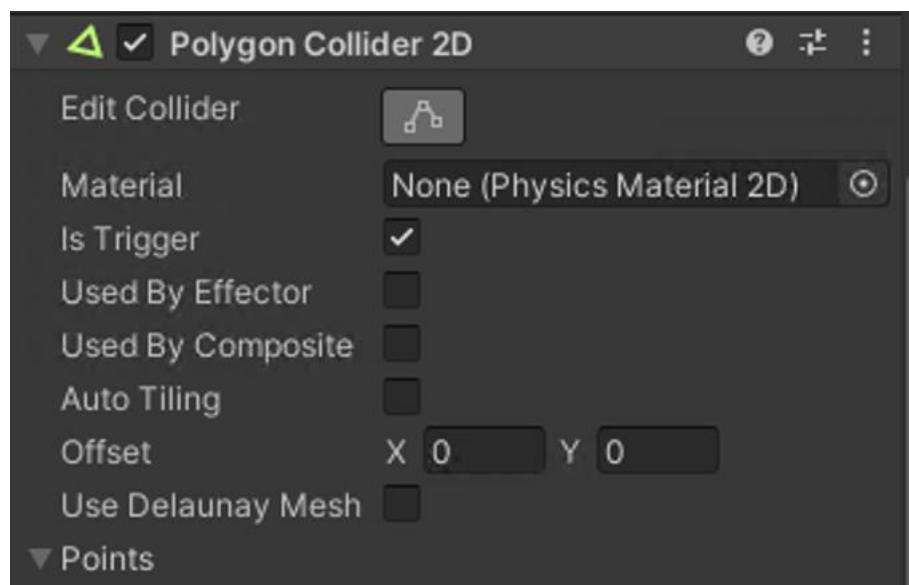
21 You should now see the **Polygon Collider 2D** component in the inspector window. Make sure that **Is Trigger** is checked.



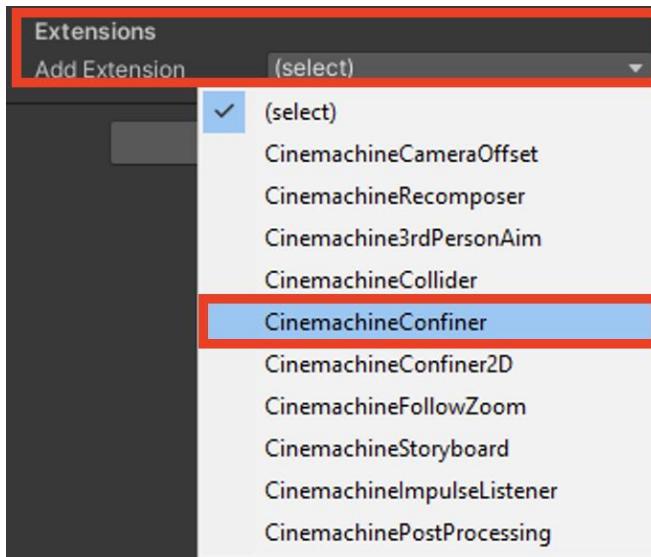
IMPORTANT: If you do not check **Is Trigger**, Unity will treat the entire background as a solid object and move the player outside of it!

22

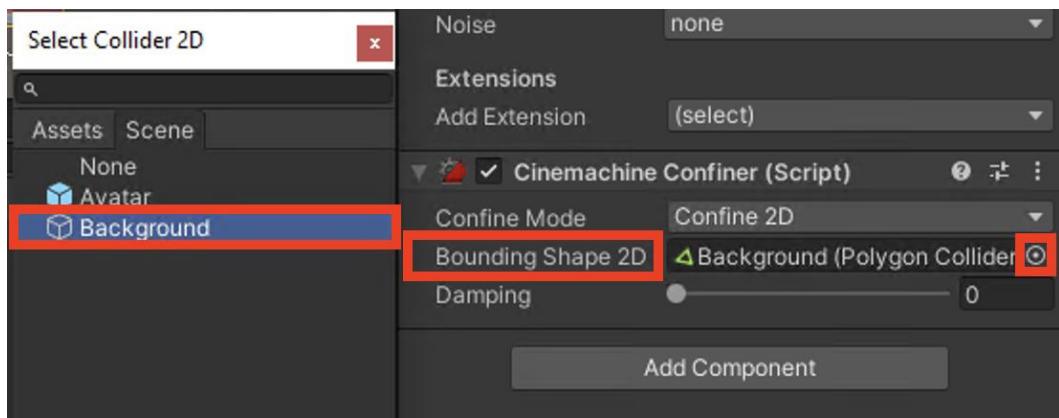
The **Polygon Collider** needs to be adjusted to match the background. Click **Edit Collider** and see if the collider is already at the corners of the background. If not then drag the corners of the collider to the corners of the background. Any extra corner points can be removed by opening “**Points**” then “**Paths**” and then “**Elements**” in the component. Then, right click the point that you want to delete and remove it.



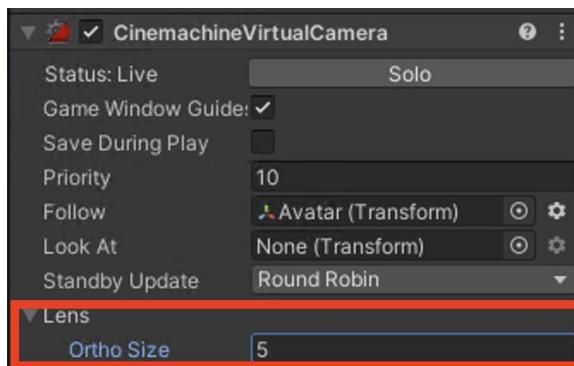
23 The final step of configuring the virtual camera involves specifying which **GameObject** contains the boundaries it should follow. To do this, locate and select the "Virtual Camera" **GameObject** in the **Hierarchy**. Then, in the **Inspector** panel, scroll down to the bottom and click "Add Extension." From the dropdown menu, choose "**CinemachineConfiner**."



24 **Cinemachine Confiner**, a new component, has been added to the Inspector. Where it says **Bounding Shape 2D**, we need to add the **Background GameObject**. This can be done either by clicking on the small circle and selecting **Background** from the menu that appears, or by dragging the **Background GameObject** from the Hierarchy panel into the slot for **Bounding Shape**.

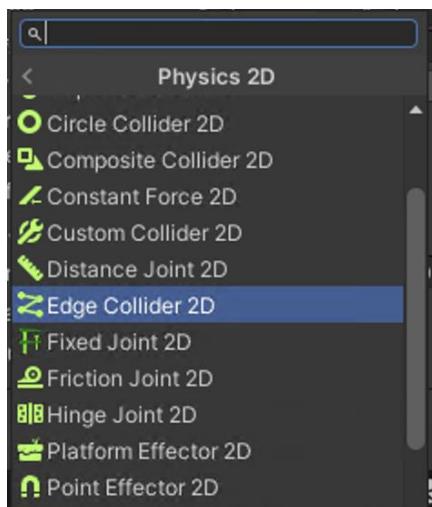


25 Now, let's adjust the **virtual camera** to bring it closer to the **Avatar**. In the **Inspector** for **Virtual Camera**, find the setting for **Lens**. If necessary, you can expand it by clicking the triangle to the left. Change **Ortho Size** to 5 to make the camera view closer to the **Avatar**.



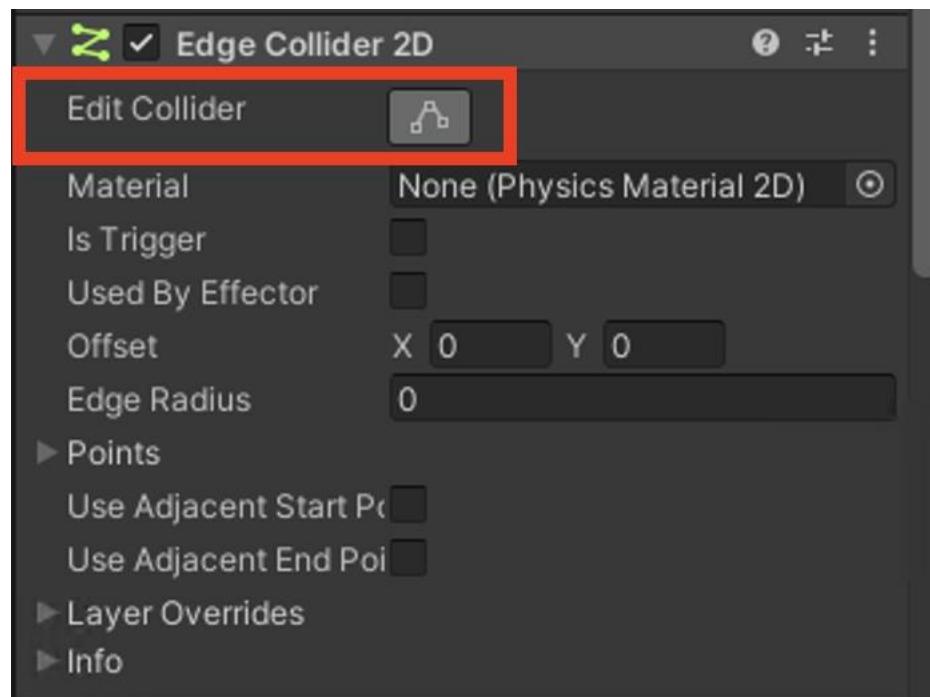
26 Test the camera by clicking the Play arrow above the scene. You'll notice that the virtual camera will follow the player until it falls off the edge of the screen. Since we've confined the camera, it won't move past the edge of the background. Unfortunately, there's nothing stopping the Avatar. Let's fix that. Click the Play button again to stop the game.

27 To keep the player within the scene, we can use a collider. If you added an ordinary **Box Collider**, Unity would try to put the player outside of the box. Fortunately, there's a collider for situations like this called the **Edge Collider**. In the **Inspector** panel for the **Background**, click **Add Component**, select the **Physics 2D** menu then select **Edge Collider 2D**.



28

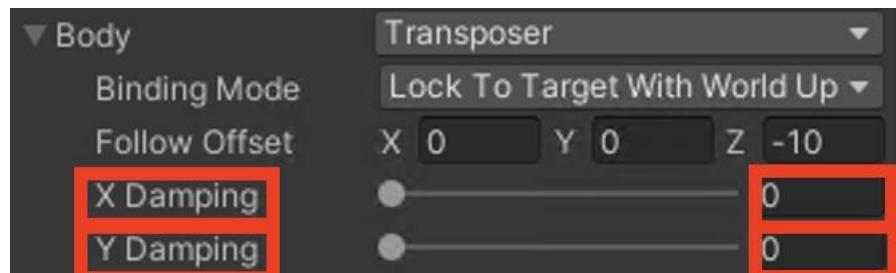
The **Edge Collider** might be hard to see at first since it's nothing more than a straight line. Click **Edit Collider** to adjust it to match the background.



29

Editing the **Edge Collider** follows the same process as editing the **Polygon Collider** which you have now done. You can simply click and drag one end of the collider over to where you want it to be. You can add more points and make corners by holding the cursor over the middle of the line. A drippable point will appear there. Stop once you have made a closed box around the edge of the background.

In the **Hierarchy** select the **Virtual Camera GameObject**. In the **Inspector** panel open the "**Body**" section using the triangle. Adjust the "**X Damping**" and the "**Y Damping**" to 0. This ensures the camera does not get jittery.

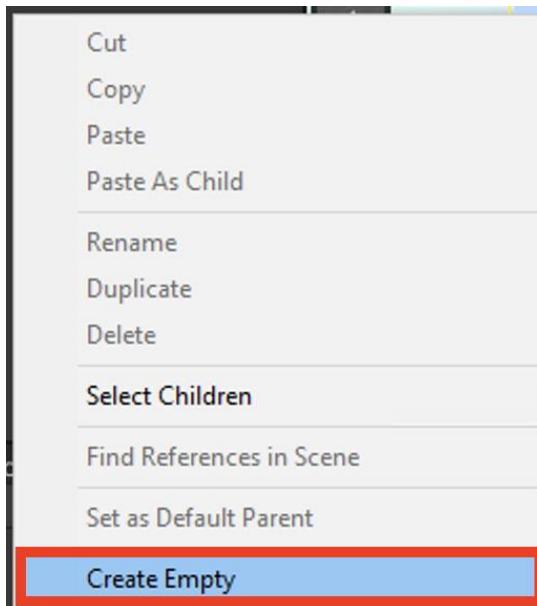




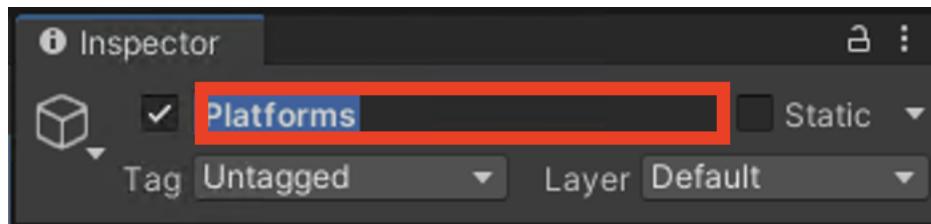
PRO TIP!

The edge collider is exactly where the Avatar stops moving, so feel free to place it above the sidewalk and inside the edges of the background to make the Avatar have a more natural look and feel when it moves. You can test this by playing the scene until everything looks good to you.

- 30** The next step is to create more colliders for the **Avatar** to walk on. To keep things organized, we'll create a place in the scene to hold all the platforms. In the **Hierarchy** panel, right click and select **Create Empty** to create an empty **GameObject**.



- 31** Change the name of the new **GameObject** to **Platforms**.





PRO TIP!

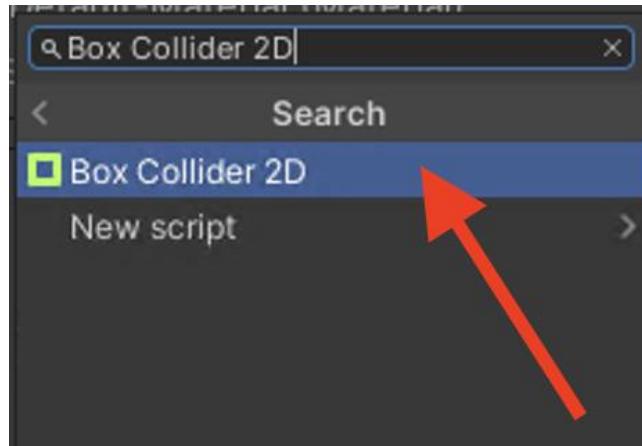
In the **Hierarchy** an empty **GameObject** can serve the same purpose as folders in the **Project** panel. In addition, since they are **GameObjects**, you can add **components** to them so that they can perform other functions in your game!

- 32** Before we begin creating Platforms select your **Avatar** in the **Hierarchy** and find your **Rigidbody2D**. Set the **Collision Detection** for your Rigidbody2D to **Continuous**. In addition check to ensure that the Z box is checked for Freeze Rotation.



- 33** To begin adding platforms, right-click on the **Platform GameObject** in the **Hierarchy** and select **2D Object** then **Sprites** and finally **Square** to add a Square to the scene inside the **Platform** object.

- 34** In the **Inspector** panel, select **Add Component**, then add **Box Collider 2D** to the square. Adjust the size of the square so that it matches the shape of the window ledge.

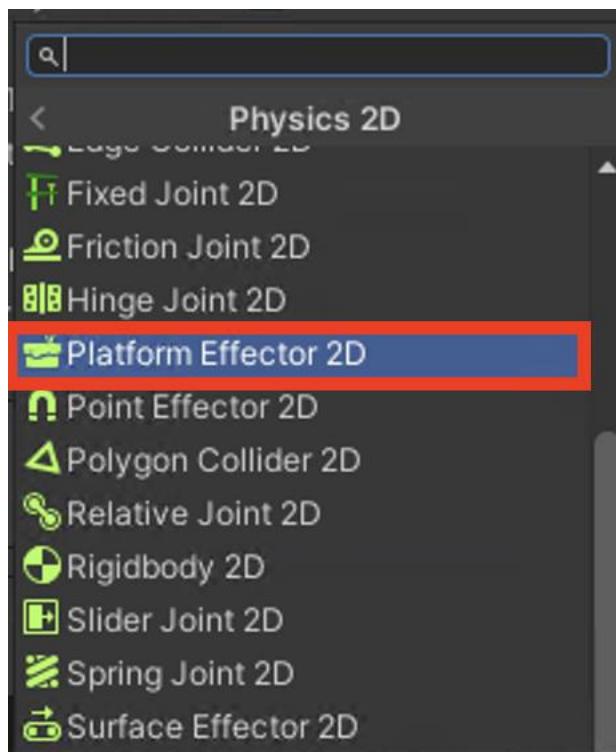


PRO TIP!

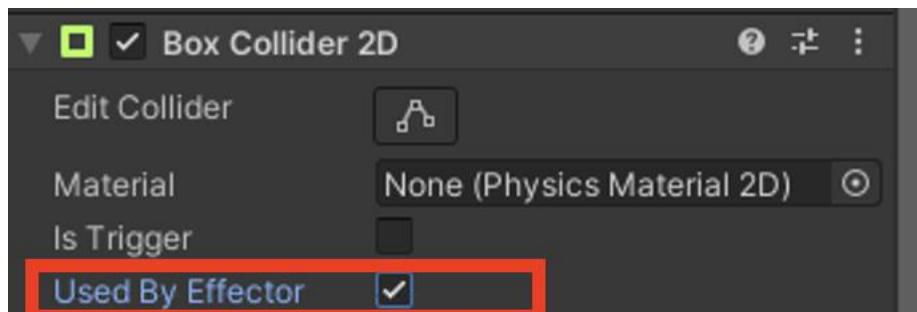
Don't forget, you can easily move around in the scene panel by using the mouse. Use the mouse wheel to zoom in or out of the scene and hold down the right mouse button and drag to get to the part of the scene that you need to work with.

35

Currently, the **Square** will block the player from all directions. However, what we truly want is a way for the collider to stop the player only when they land on the collider from above, and to let the player pass through it from all other directions. To achieve this, you'll need to add another component to modify the **Box Collider** that was just added. Click **Add Component** again and select **Platform Effector 2D**.



36 Next, we need to configure the two components to work together. With the **Square** still selected, go down to the **Box Collider 2D** component and make sure that “**Used By Effector**” is checked. This means the **Platform Effector** settings will modify the **Box Collider 2D** component. In the **Platform Effector** component, keep the default settings as they are.



37 Let’s test our platform. Begin the game by clicking the Play button. Use the arrow keys or WASD to move the avatar over to the ledge and then use the space key to jump onto the ledge. Now that there is a working platform object, you can simply copy it over to where it’s needed.

Stop the game by clicking the **Play** button again.

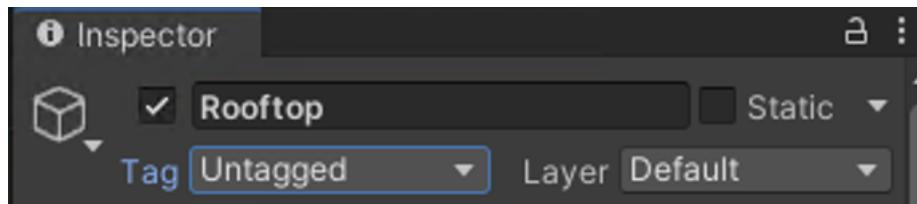
PRO TIP!

Don’t worry if the platform object is still visible in front of the background.
That will be fixed in the next step.

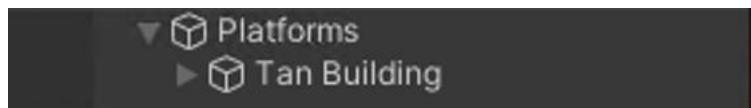
38 Before proceeding, adjust the position of the **Square** so that it is positioned behind the background by changing the **Order In Layer** value to **-1**.

With the **Square** still selected, rename it to **WindowLedge**. Then, use **Ctrl+D** to duplicate **WindowLedge** and move the duplicate over to one of the other windows. Select both the original and the copy and press **Ctrl+D** again to copy both objects and move the copies over to new windows. Select all four platforms and press **Ctrl+D** again to create enough window ledges for the entire building.

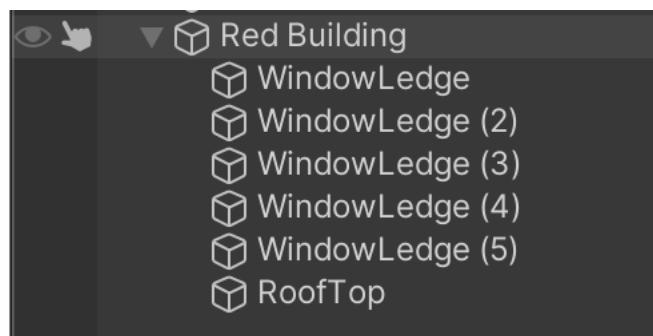
To complete the building, make a single copy of a ledge and move it to the roof. Rename the platform **Rooftop** and adjust its size to cover the length of the roof.



39 To keep things organized, right click on the **Platform GameObject** in the **Hierarchy** panel and create an **Empty Object**. Rename this object **Tan Building** and move all the objects you just made so that they are now part of the **Tan Building GameObject**.



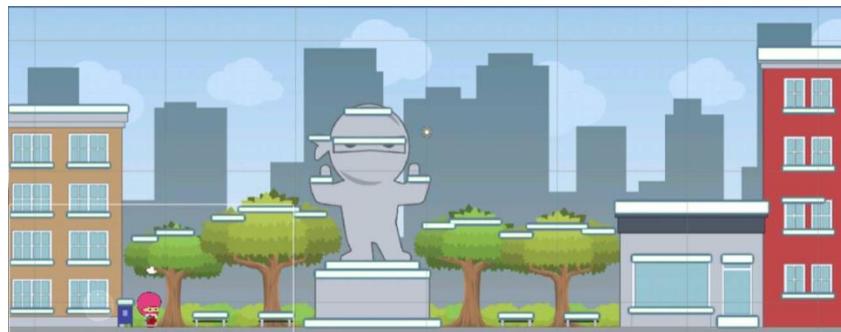
- 40** While the **Tan Building GameObject** is selected, press **Ctrl+D** to copy the object along with all the platforms inside it. Change the new **GameObject** name from **Tan Building** to **Red Building** and use the **Multiple Objects Tool** to move these new platforms to the red building on the right side.



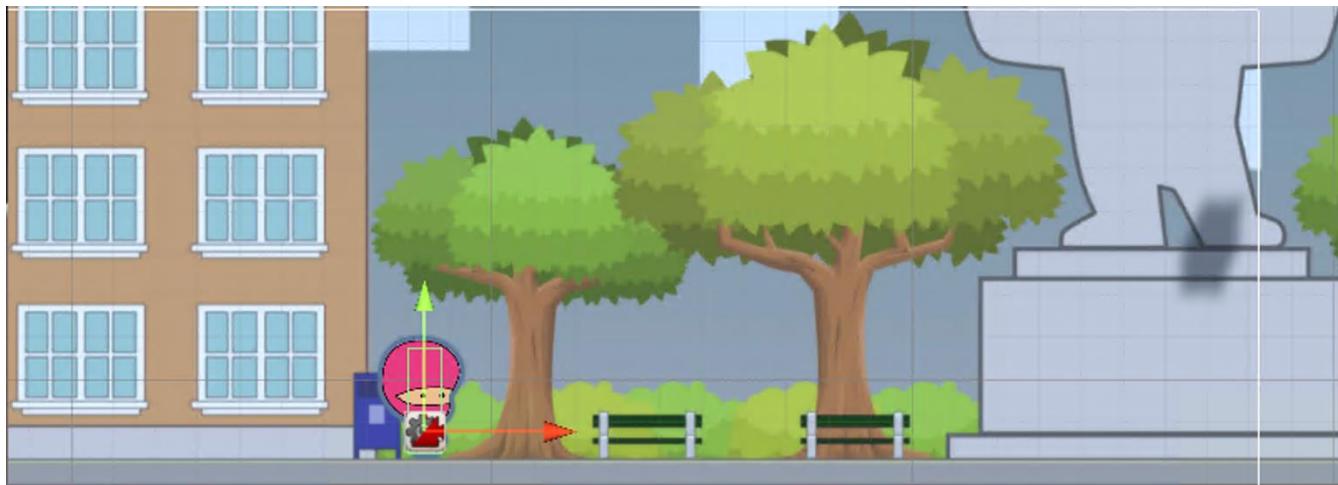
- 41** Similarly to how you handled the Tan Building, move the platforms for the window ledges and roof top to their proper places for the Red Building. Delete any platforms you don't need.



- 42** Continue with duplicating and repositioning the groups of platforms. Use this guide as a suggestion for grouping and placing the platforms.



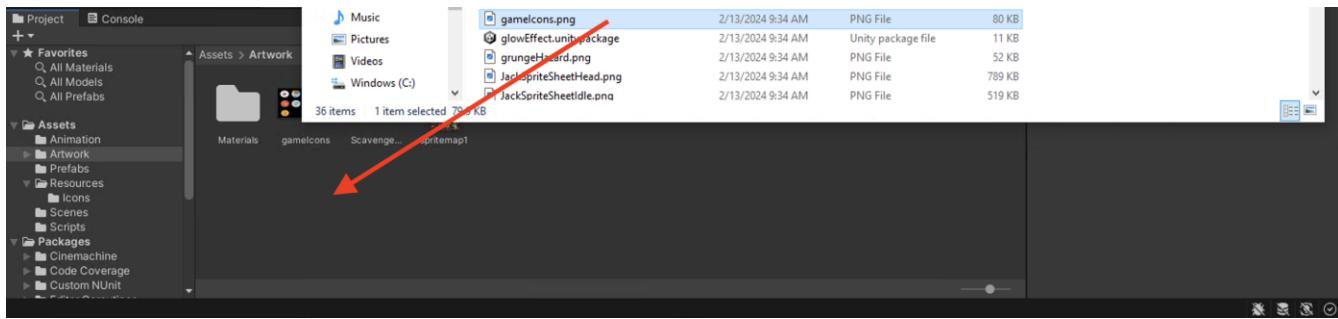
43 If it is not done already, position the Avatar on the ground. Here, we have placed it next to the mailbox.



44 Test the game by clicking the Play arrow. Make sure that your avatar can reach all parts of the scene that have platforms.

Stop the game by clicking the Play arrow again.

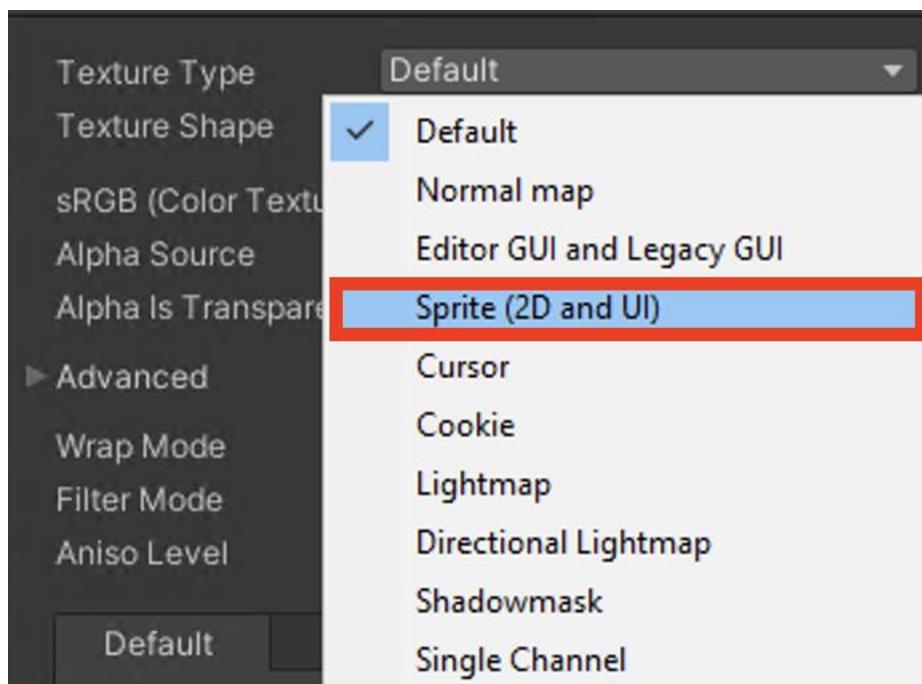
45 Let's provide our player with something to collect. In the **Projects** panel, make sure that the **Artwork** folder is open. Open File Explorer on your computer and navigate to where the Purple Belt files are located. Find the file called **Activity 02 - gamelcons.png**. Click and drag it into the **Artwork** folder.



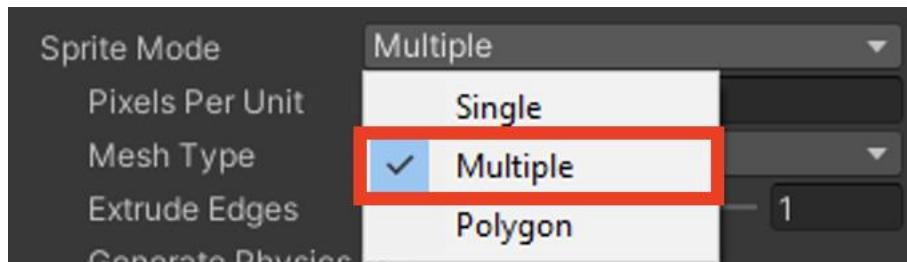
46

With the gamelcons selected in the **Projects** panel, navigate to the **Inspector**, and make two changes as follows:

Ensure the **Texture Type** is set to **Sprite (2D and UI)**.



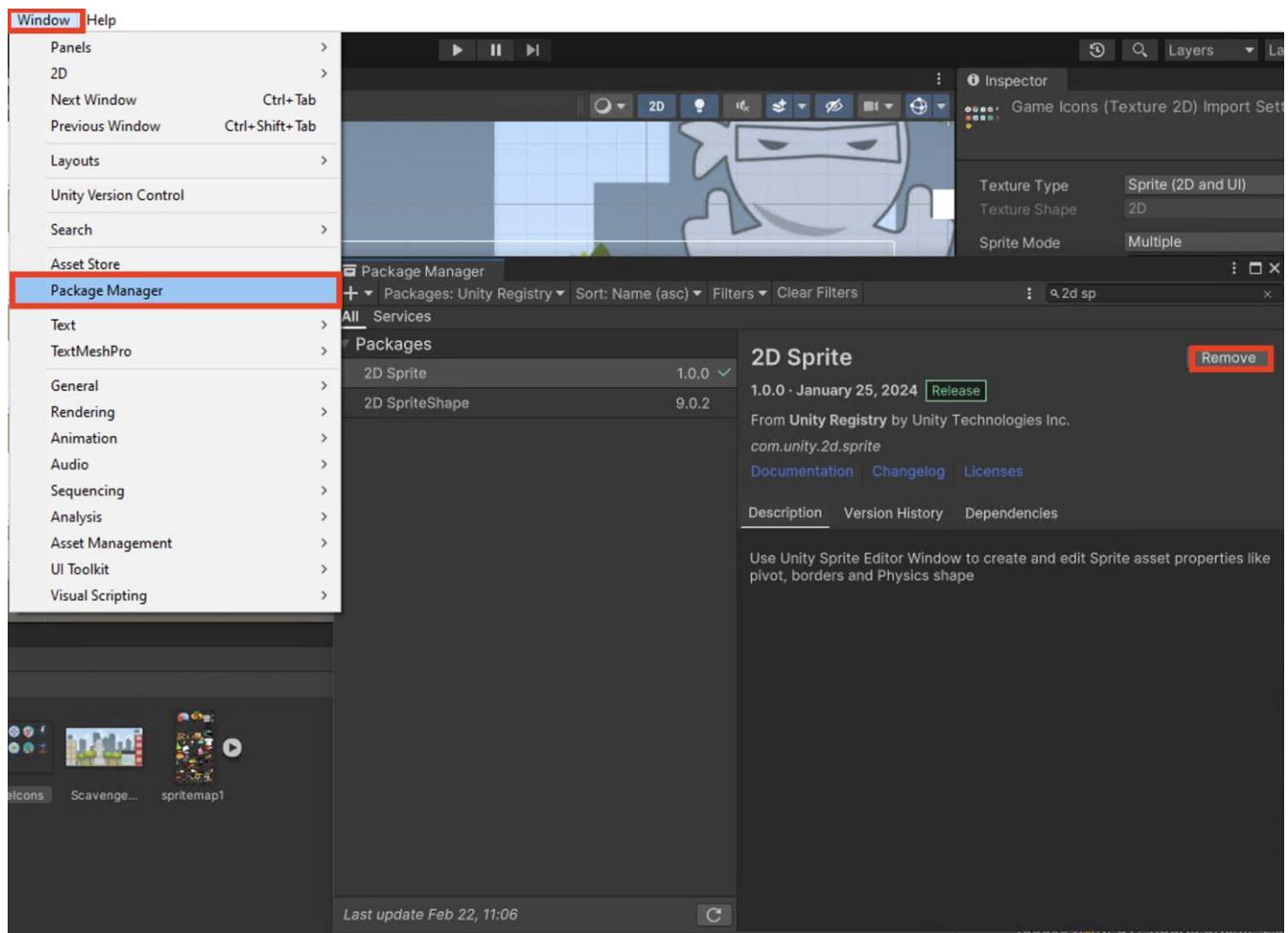
Set the Sprite Mode to **Multiple**.



To save these changes click the apply button near the bottom right corner.

47

While you should have the **2D Sprite** package installed. Let's check by clicking the "**Windows**" tab in Unity and accessing the **Package Manager**. Locate and select the "**2D Sprite**" package. Click "**Install**" if prompted and wait for the installation to be completed before closing the Package Manager and proceeding. If it says "**Remove**" then you can exit Package Manager.



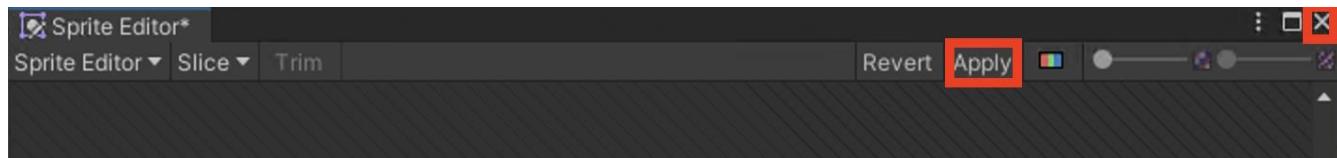
48

While still in the **Inspector**, click on the **Sprite Editor** button. By changing the mode to **Multiple**, we can now get many sprites from this single image. To do so, click on the **Slice** tab at the top of the **Sprite Editor**. For now, we'll keep the settings as is. Click on **Slice**.



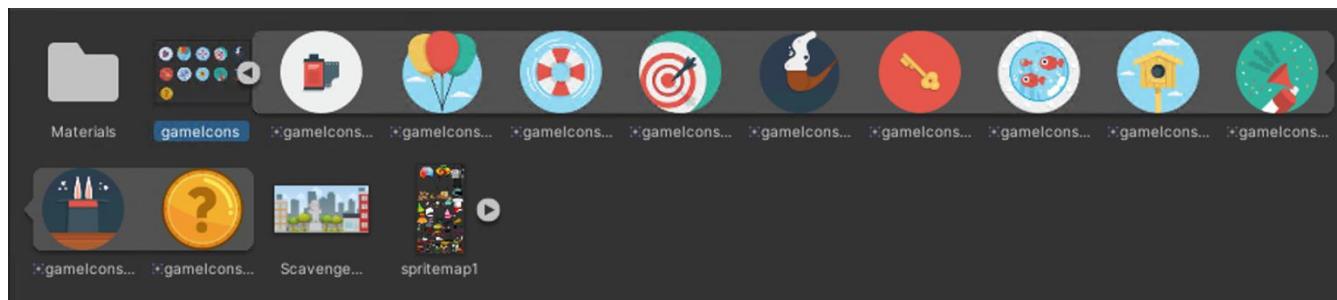
49

Now, all the icons on the sheet are defined as individual sprites. Click on any of them to see more details. Once you're done, click **Apply** and then close the **Sprite Editor**.



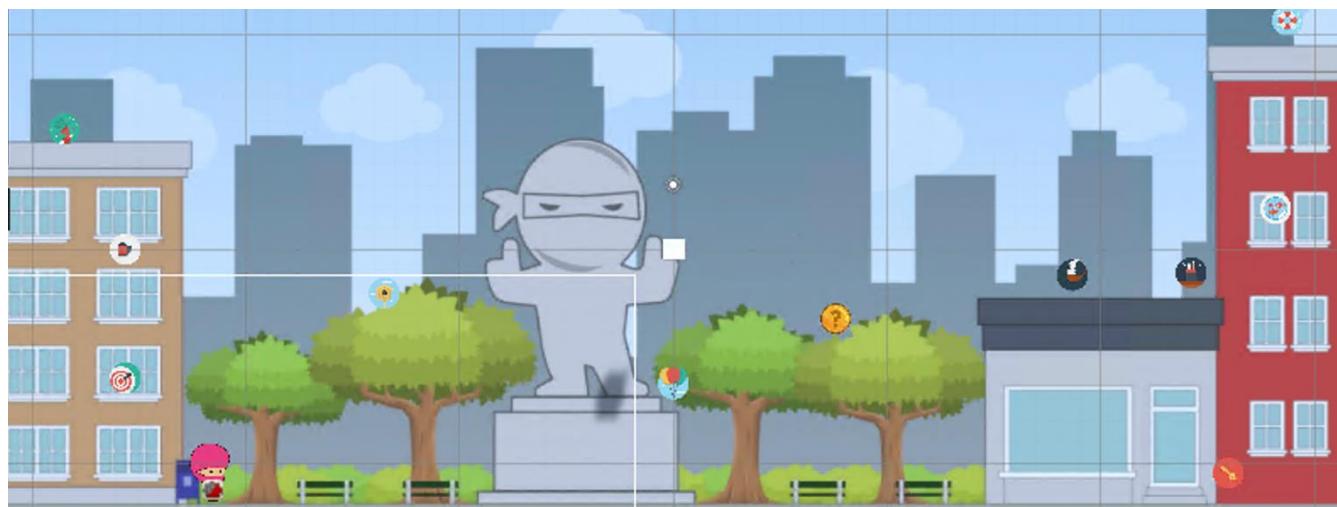
50

In the **Project** panel, the **gamelcon** asset now has an arrow on the right side. Click on it to see all the sprites.



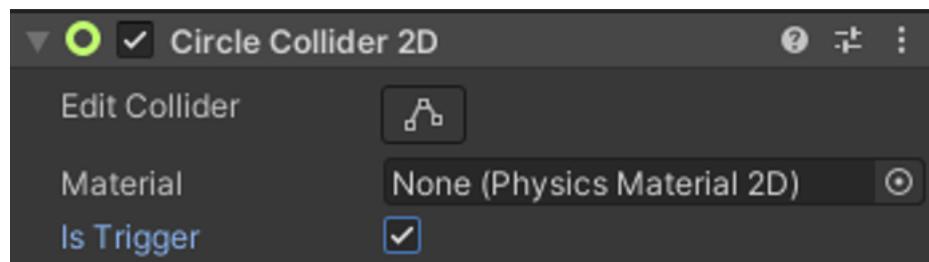
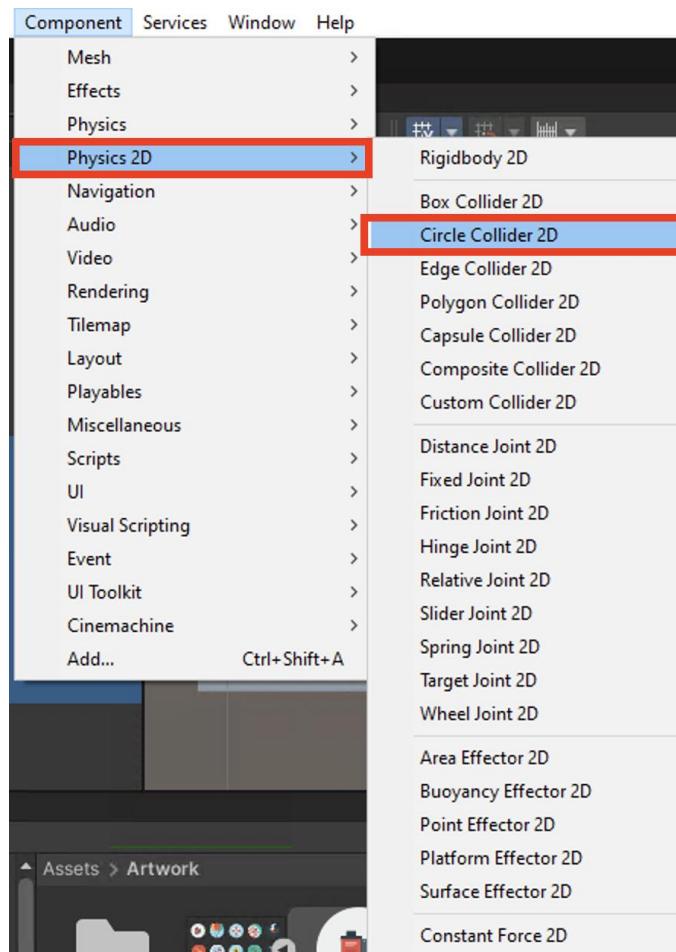
51

In the **Hierarchy** panel, create a new empty object and name it **Collectibles**. Drag each of the new sprites into the **Collectibles** object within the **Hierarchy**, then position them throughout your scene in places where they might be challenging to reach.



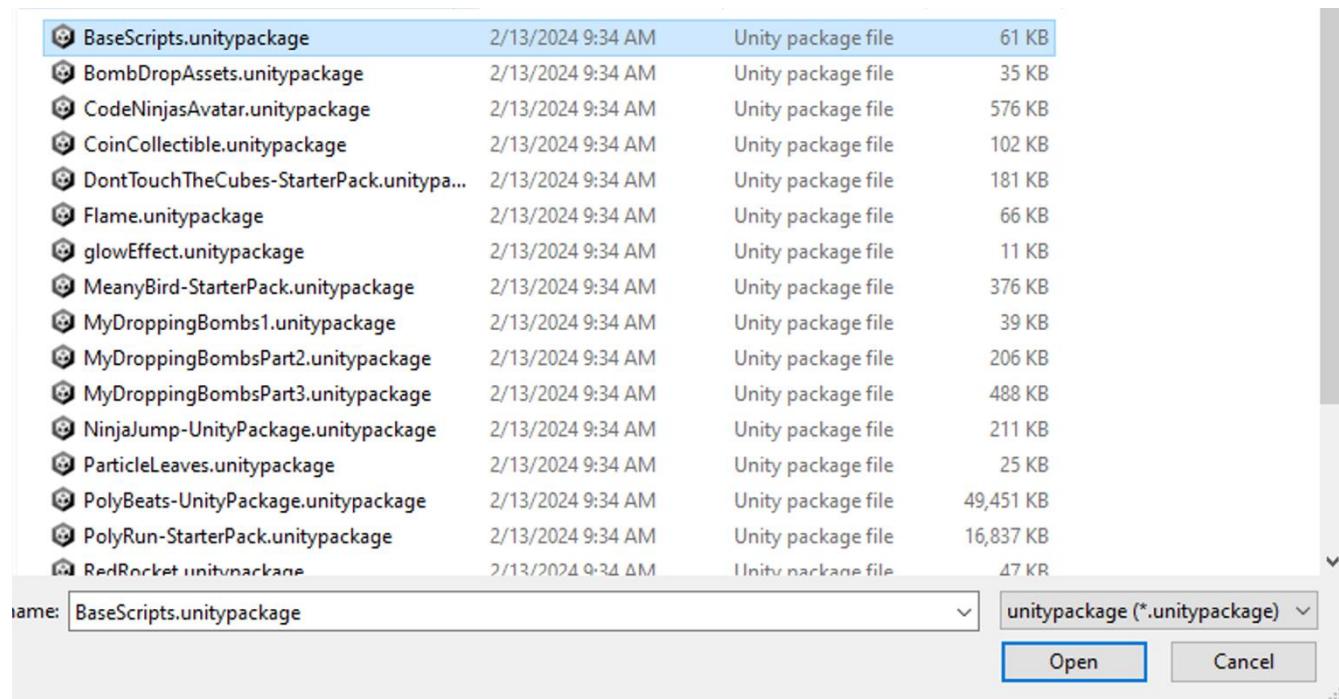
52

Currently, the sprites won't detect when the player touches them because they have no collider. Select all sprites in the **Collectibles** group and click the **Component** menu to add a **Circle Collider 2D** to them as shown. While all sprites are selected, ensure that **IsTrigger** is checked in the **Circle Collider** component.



53

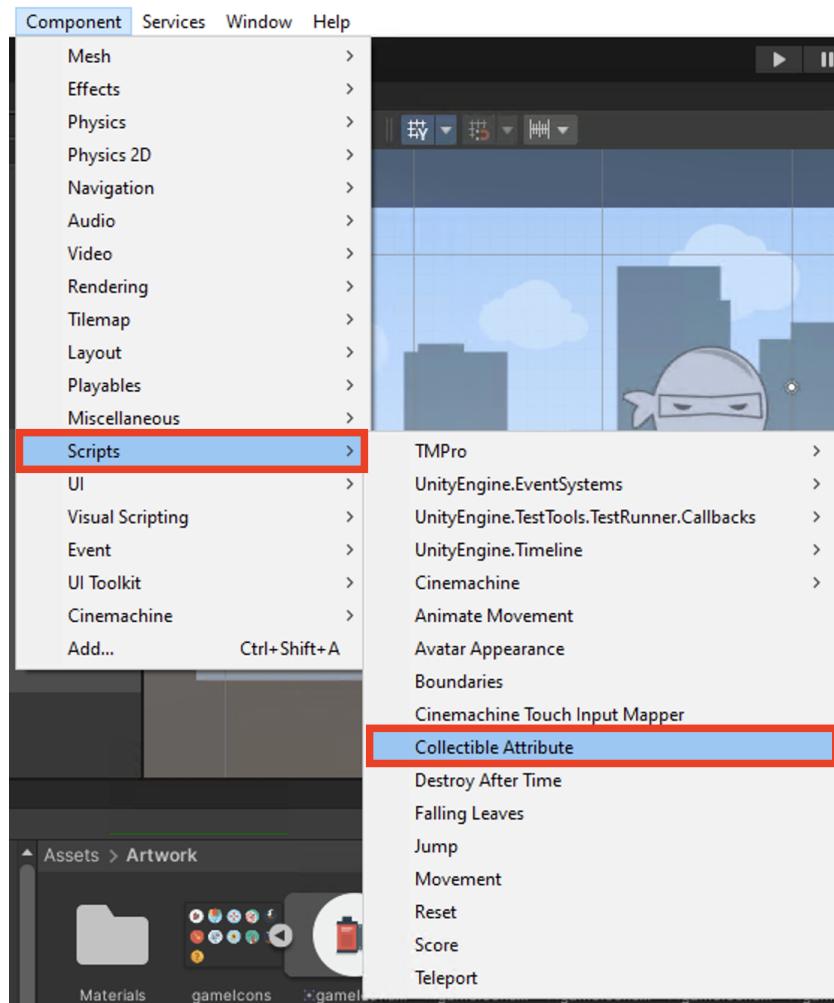
We also need a script to tell the sprites what to do. Click the **Assets** tab, then select **Import Package** followed by **Custom Package**. Navigate to your Purple Belt files, then select **Activity 02 - BaseScripts.unitypackage** and import it.



54

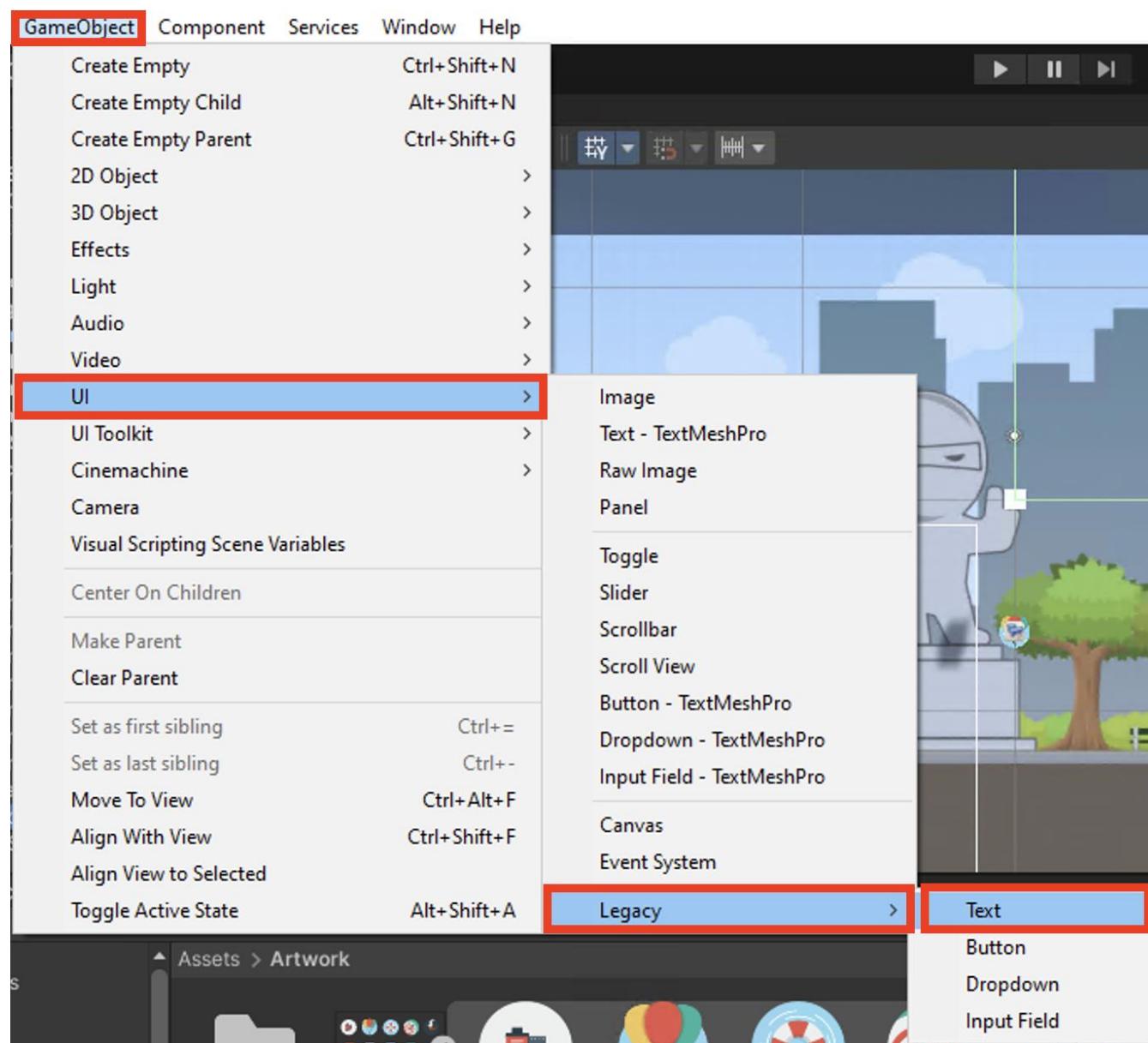
You should now have some new scripts for the game. Highlight the sprites in the **Collectibles** object again, then click on the **Components** tab.

Navigate to **Scripts**, then select the **Collectible Attribute** script to add it to all sprites. The script will have the sprites add points to your score when you touch them. But wait, you don't have a score just yet!



55

To display the score, you'll need a new **GameObject** called a **User Interface (UI)**. Click the **GameObject** menu, select **UI**, then **Legacy**, then **Text**.



56

The **UI GameObject** automatically generates an object named **Canvas** to hold all **UI** elements. However, you probably won't see anything right away.

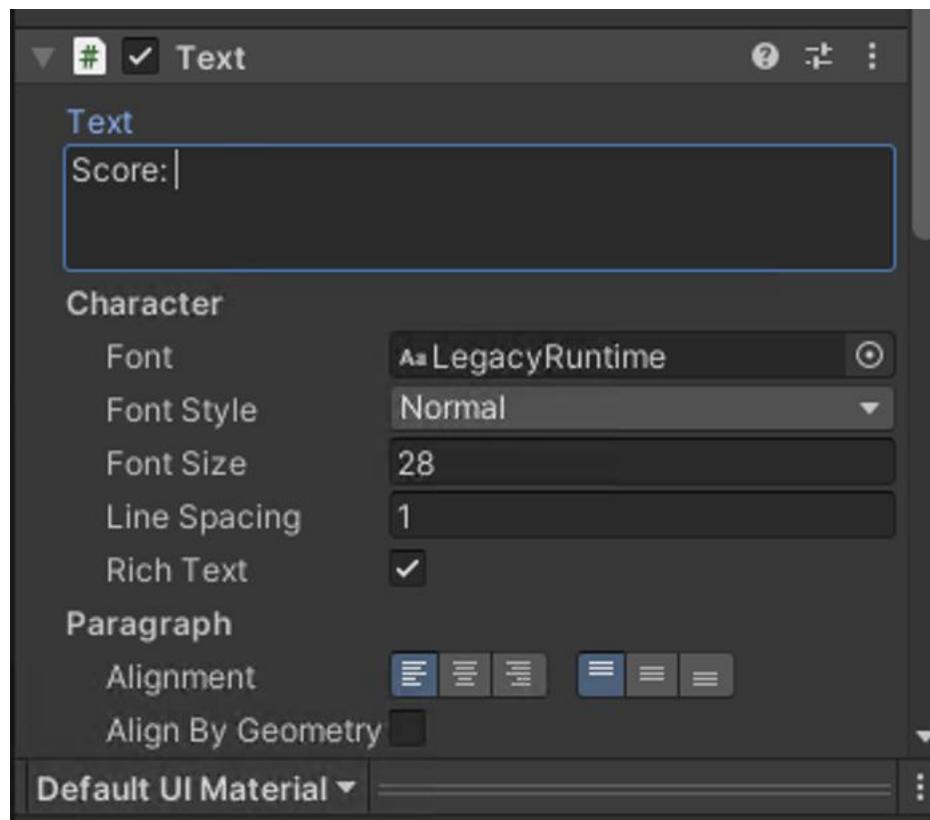
Next to the **Scene** tab, there is a **Game** tab that allows you to preview the game screen. Select the **Text** object in the **Hierarchy** and click on the **Rect Transform** square in the Inspector to expand it. This provides options for adjusting where the text appears. While holding down the **Alt** key, click on the upper left square in the grid to move the text to that corner. Now the text should be visible, but it may appear small.



57

Click back to the **Scene** tab. With **Text** still selected, press the **F** key or double-click to focus on the **Text** object. If pressing **F** doesn't work, try **Shift + F** to focus.

Now, let's fix the size of the **canvas**. Unity ensures that the **UI** appears correctly in complex 3D environments by defaulting to a large canvas size. In the **Inspector**, scroll down to the **Text** component. Change the **Text** to "**Score:**" (don't forget the space after the colon!) and increase the font size to **28**. Even though the font is larger, the Text object stays the same size. Use the **Rect Tool** to make the object large enough for both your font and your score. Also, move the object so that it isn't touching the edges of the **Canvas**.



58 If you want to, you can change the font. There are plenty of fonts available for download at fonts.google.com. If you find one you like, select it and download it to your computer. Google fonts are in a .zip file that needs to be unpacked, but then you can add the fonts to your game just like any other asset.

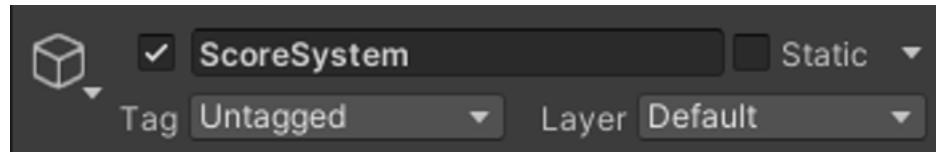
1 font family selected



How to use

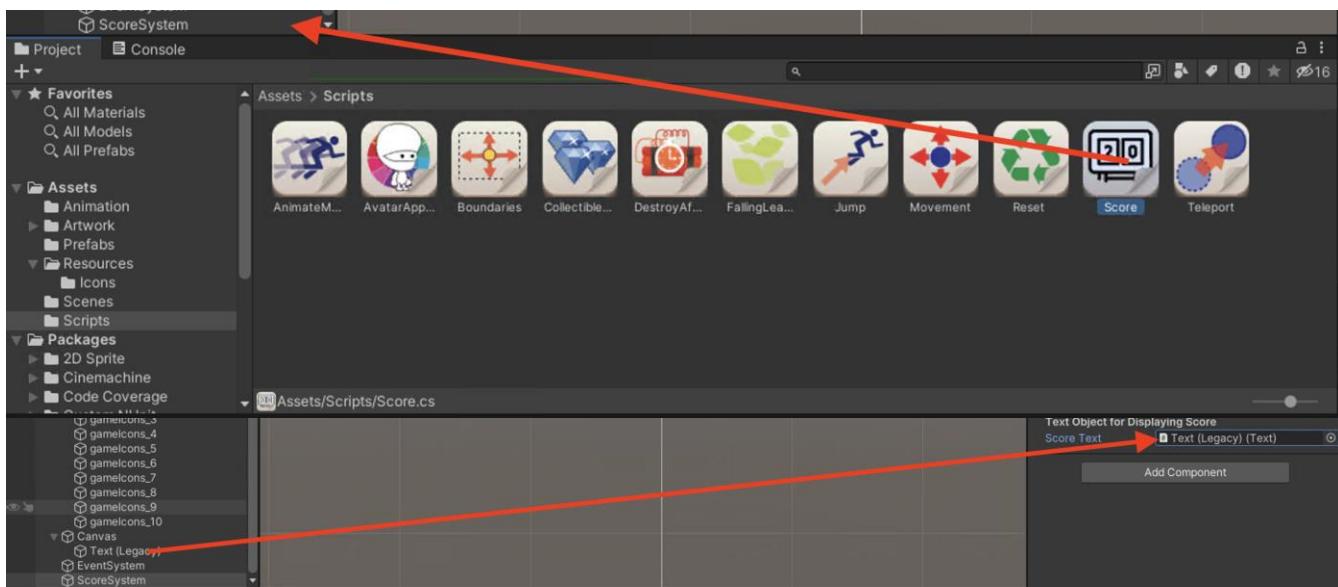
All Design Develop Google products

59 Let's return to our scoring system. In the **Hierarchy** panel, create a new **Empty Object** and name it **ScoreSystem**.



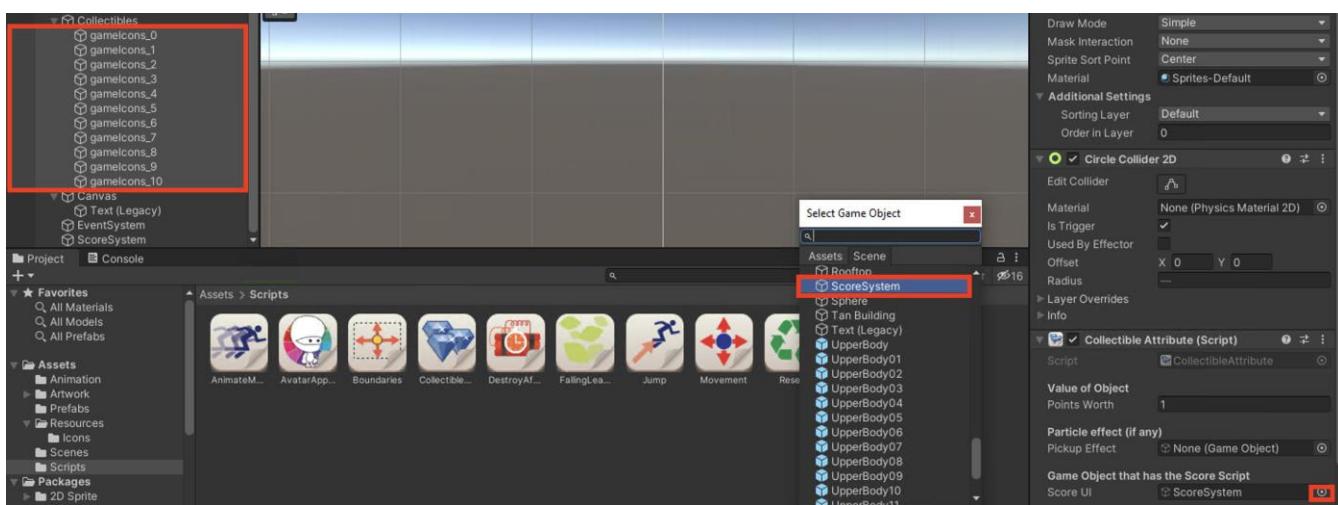
60

Open the **Scripts** folder and drag the **Score** script into the **ScoreSystem** object. In the script component, there's a property for the **Text** object. Drag the **Text** object from the **Hierarchy** panel into the **Score Text** field in the script component.



61

To connect the **Collectibles** to the **ScoreSystem**, select all the sprites within the **Collectibles** object. In the **Inspector** panel, find the **Collectible Attribute** script. There is a slot for the **Game Object with the Score Script**. Click on the circle to the right of the box to open the menu and select **ScoreSystem** to add that parameter to all the collectibles.

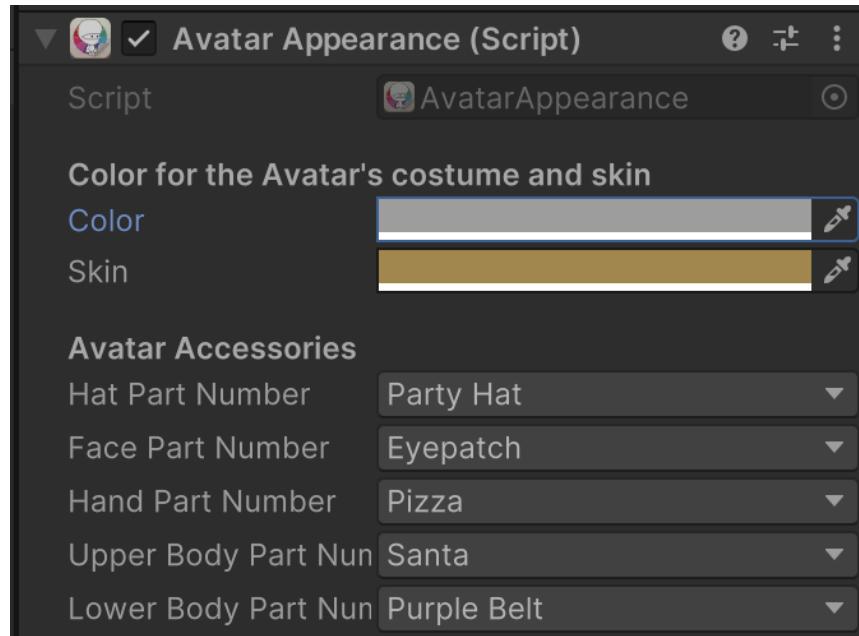


62

Congratulations - your Scavenger Hunt game is ready to play! Go ahead and give it a try!

Customizing the Player Avatar

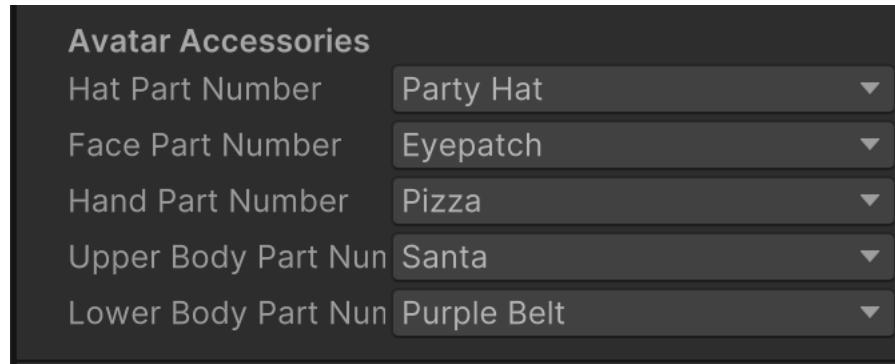
In the Hierarchy panel, select the Player Avatar. Then, in the Inspector panel, scroll down until you find the Avatar Appearance script component.



To change the player's color or skin, click on the appropriate property to open the color selector window. Close the window when you have a color that you like.



To change the accessories, click on any of the menus under the Avatar Accessories heading. Pick the object you want by name (or choose None to leave that section empty).



Save your project and the next time you open your game, it will have the settings that you've selected.

Prove Yourself: Particle Hunt



Unity has a simple way to add particle effects to objects. You can make things look like they're on fire, use smoke effects, or any of the dozens of effects Unity has built-in! For this Prove Yourself, you're going to use the glow effect. In the folder where you get your unity files, you'll see one called "**Activity 02 Prove Yourself - glowEffect.unitypackage**"; import it. If you don't have these files, ask a Code Sensei for help. Under the prefabs you will now see a gem asset. Drag this onto a **gamelcon** in the hierarchy.

Select that **gamelcon**, and in inspect under the sprite renderer, change the **gamelcons** order in layer to 1. Now, in the hierarchy, find the glow object and select it. Scroll down until you find "start color" and change the color to match the current **gamelcon** you have selected.

Now repeat these steps until all the **gamelcons** in the scene have an attached particle effect.

Hint: Make sure to reset the transform of the particle effect **GameObject** to **(0,0,0)** to ensure it is properly positioned!



PRO TIP!

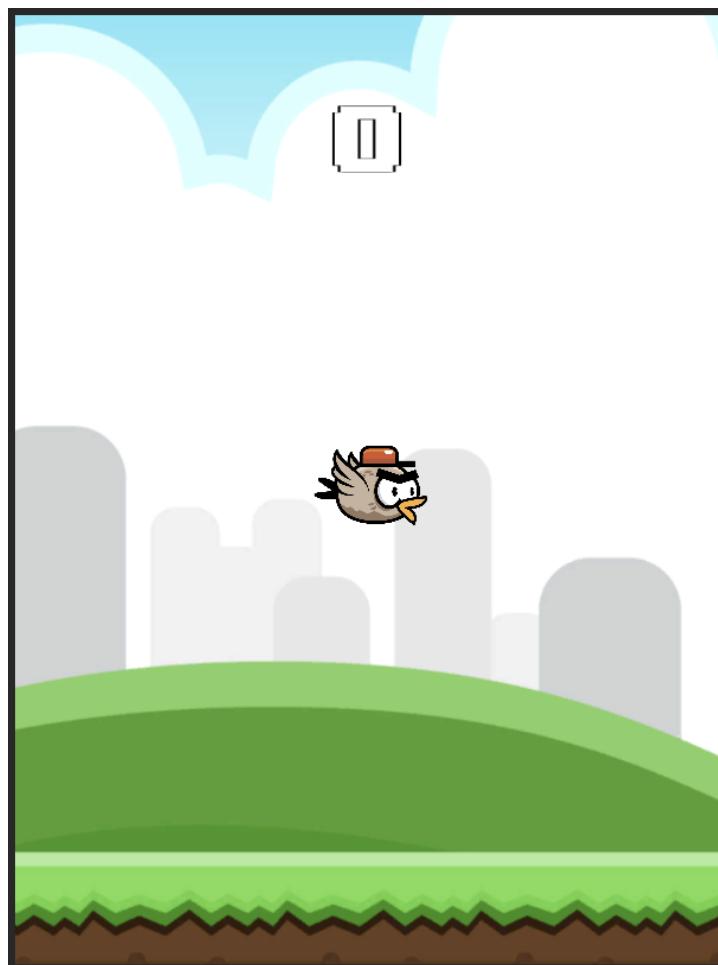
You MUST reset the transform of the gem game object after dragging and dropping the gem prefab onto the game icon object. The glow effect won't show up directly beneath the game object if you don't.

The "Order in Layer" property refers to the rendering order of 2D objects within the same sorting layer. Objects with a higher "Order in Layer" value will be rendered on top of objects with a lower value within the same sorting layer.

Graphics and Animation

In the previous section, we covered how to bring in assets to create a simple platforming game. You also learned how to apply imported images to objects in Unity.

Now, it's time to give the images a little more life using motion and animation, by starting with this simple, yet challenging game.



Making Things Look Good

If a game is fun, why should it look good? The reason has to do with a concept called **User Experience**. The goal of user experience is to make sure that whoever uses the product has a good time. Having something that does what it's supposed to do is only one part of the game experience. Adding in the right graphics can help with "setting the stage" and letting the user know what to expect.

Graphics in Unity

Graphics are displayed in several ways in Unity. The first way graphics are displayed is through rendering, by applying a material to an object and applying light and other parameters to give it a specific look. 2D objects such as sprites are also rendered, using a different method than 3D objects. Another way graphics are displayed in Unity is through visual effects, such as particle systems.



Animation

Games tend to work better with motion, and Unity provides many tools to keep things active and moving. Unity uses physics to simulate realistic events such as objects falling or bumping into each other. Through scripting, code can be used to direct motion within a game. Unity also has animation tools that allow for the creation of visuals like having a character running or causing objects to spin and fly.



Image Source: <https://youtu.be/XQIFokCzU6M>

Bells and Whistles

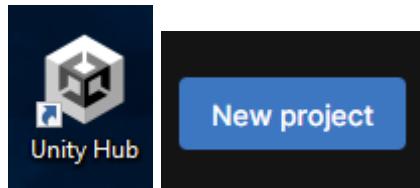
In the video game realm, developers have a term for effects that improve the user experience but aren't absolutely necessary. They often refer to these enhancements as "bells and whistles" (it is believed that the term originated in advertising in the 1950's).

This term encompasses more than just flashy graphics and sound effects. It's a combination of sounds and images that assist the user's imagination in taking them to wherever you want them to go when they play your game. The skills required to create stunning visuals take quite some time to develop, but they all begin with some simple questions: "Does it look right?", "What can be done to make it better?"

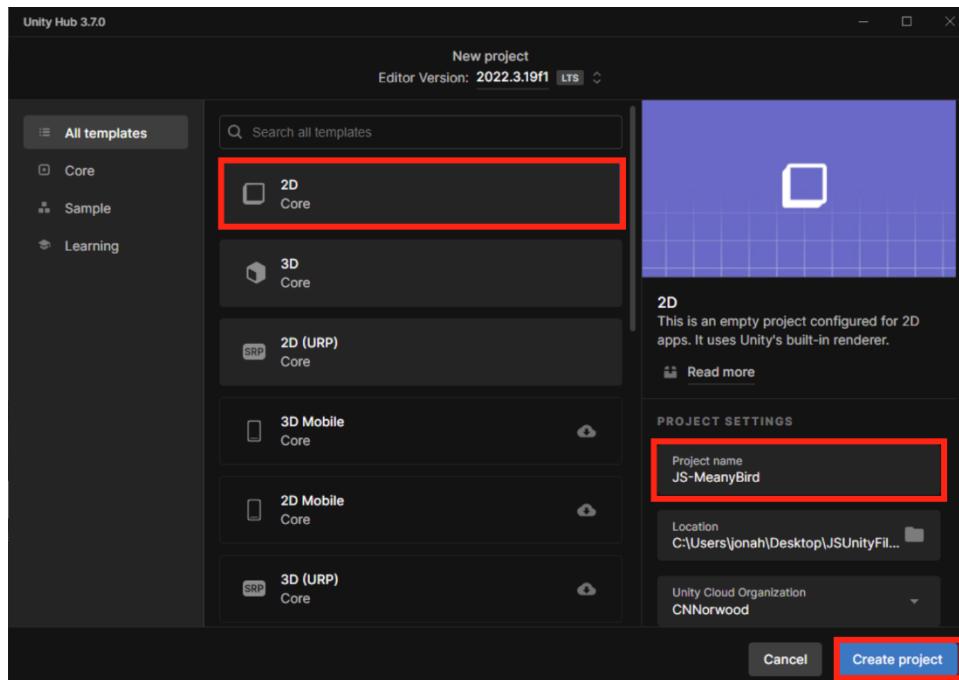
Activity 3: Meany Bird

In this lesson, you'll be learning how to create your own animations using the bird sprite we have provided for you.

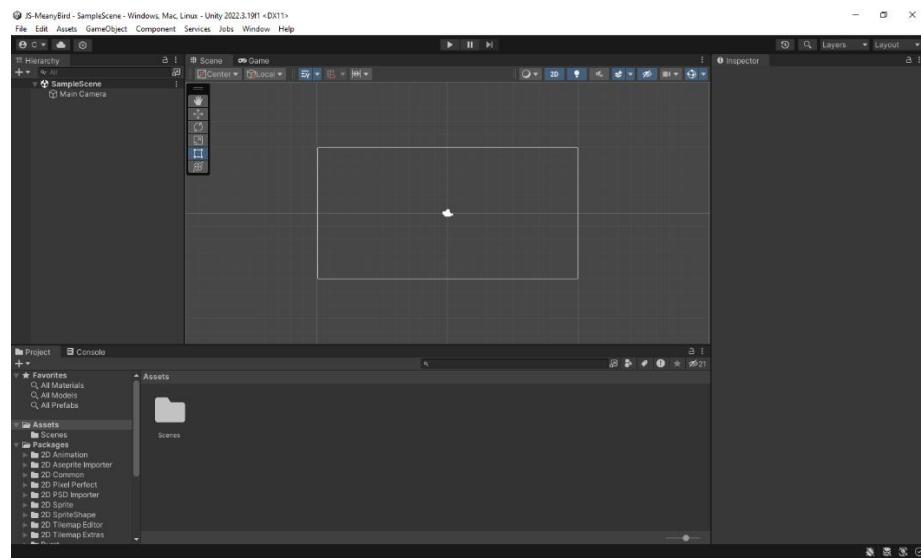
- 1 Open the Unity Hub application on your computer. Select the **New Project** button in the upper right-hand corner.



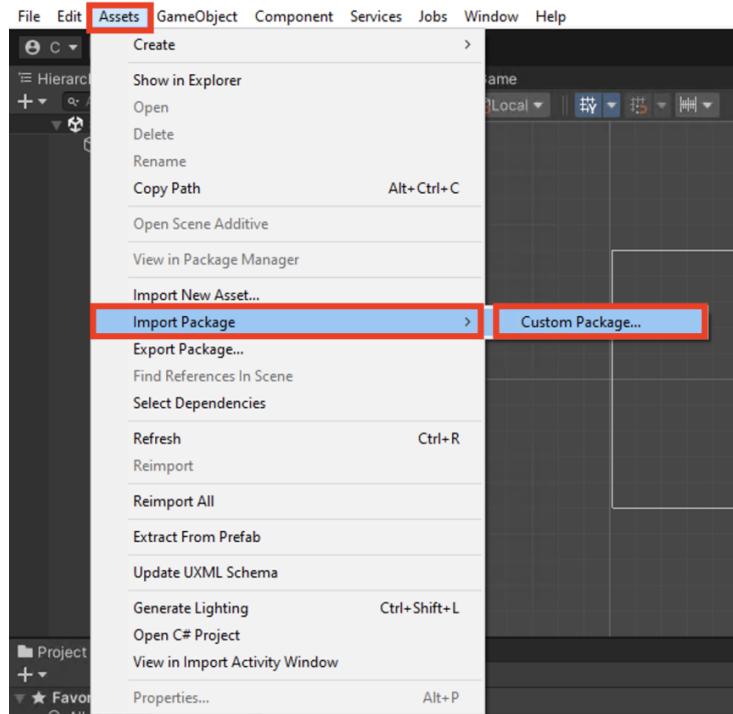
- 2 Ensure that the top shows that the **2022.3 LTS** version is being used. Select **2D** in the templates column. Next, type in your first name and last name initials and the name of the project. For example, John Smith would save their project as JS-MeanyBird. Select the folder location where you want to save your project. Click the **Create** button.



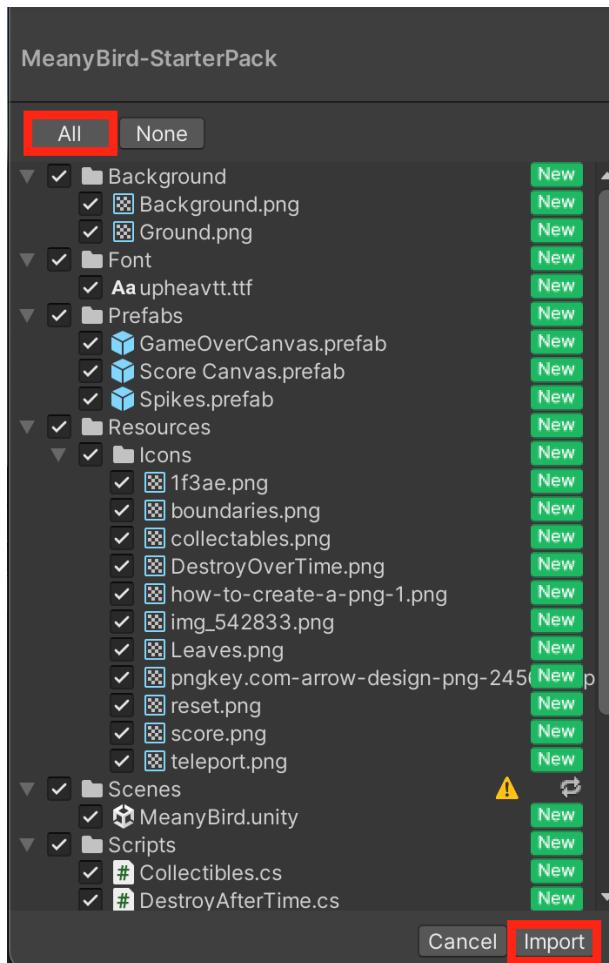
3 This is what you will see when Unity has loaded the 3 necessary assets into the program. Go to the **Assets** menu.



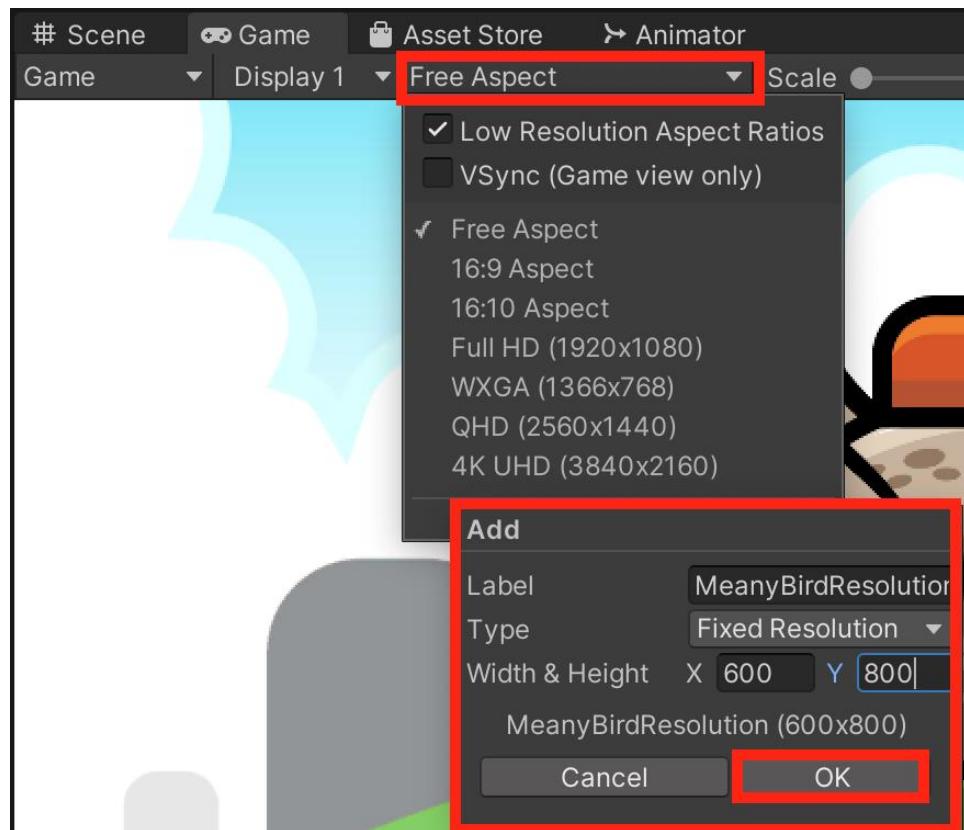
4 Select **Import Package**, then click on **Custom Package**. This allows us to import any package that has been compressed in Unity. Select the **Activity 03 - MeanyBird-StarterPack.unitypackage** file. Once imported, Unity will show a list of everything that is inside the package.



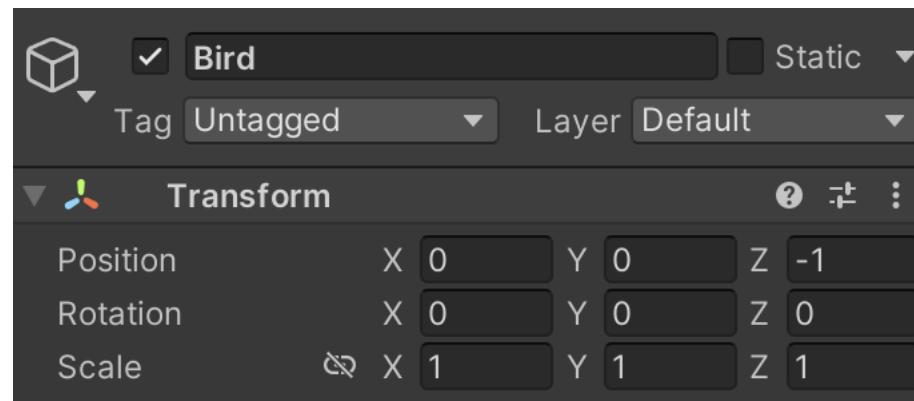
- 5** All the materials inside the Unity package should be selected. If they aren't, click the **All** button on the bottom-left. Click **Import** (bottom-right) and wait until all the material is imported into the project.



- 6** Navigate to the **Scenes** folder under the Assets directory (accessible via the Projects tab) and open the MeanyBird scene. In the Game window, click on Free Aspect. With this tab still open next, click on the "+" symbol at the bottom to adjust the resolution size. Give the resolution a label like MeanyBirdResolution. Then, set the Width and Height to 600x800.

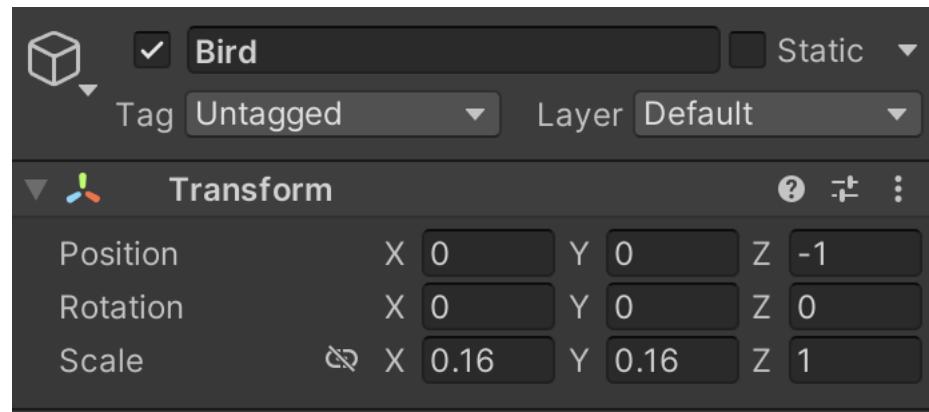


- 7** Now, let's add our bird sprite. Go to the **Sprites** folder and select the bird sprite. Drag it into the hierarchy and the bird should be shown in the game view. If not, change the Z values in the transform window as shown below.



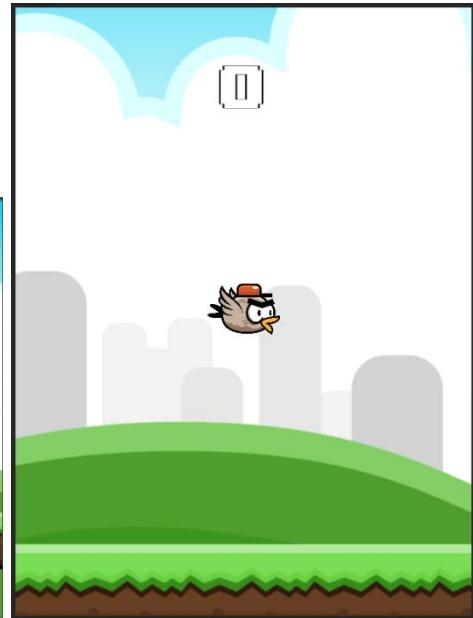
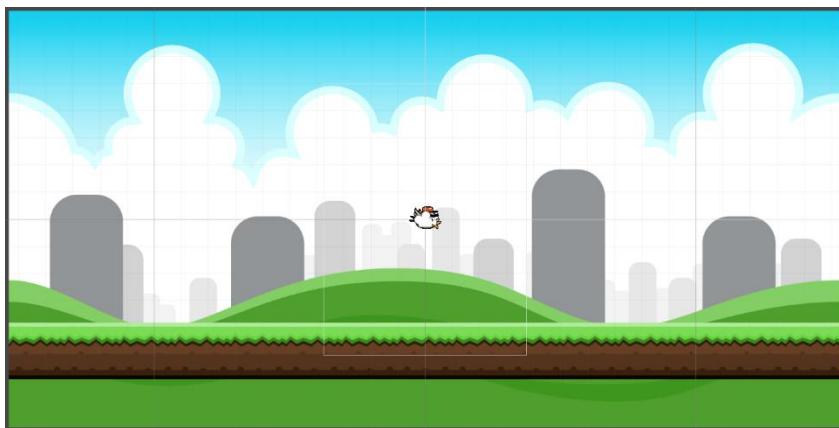
Position values: X: 0, Y: 0, Z: -1

- 8** Now the bird is too big for the game view, so let's change its size. With the bird still selected, navigate to the **Inspector** window. Change the size of the bird using the following scale values:

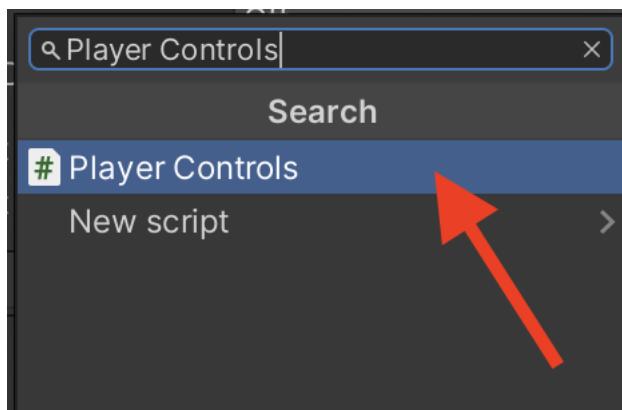


Scale values: X: 0.16, Y: 0.16, Z: 1

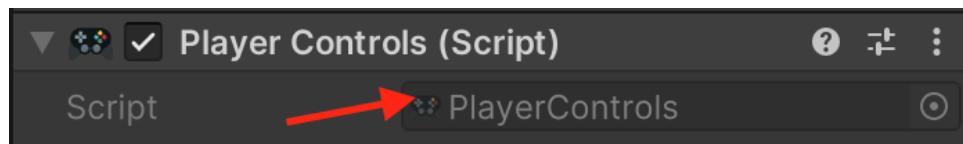
Your bird should look like this in both the scene and game windows:



- 9** With the bird still selected, click the **Add Component** button in the Inspector window. Type **Player** and select **Player Controls**.



- 10** Now, double-click on the script to open Visual Studio. Note: If Visual Studio opens and there is nothing showing, double-click the script again.



This is what your script should look like when Visual Studio is opened:

A screenshot of a code editor window titled "PlayerControls.cs". The code is as follows:

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class PlayerControls : MonoBehaviour
6  {
7      // Start is called before the first frame update
8      void Start()
9      {
10
11
12
13      // Update is called once per frame
14      void Update()
15      {
16
17
18
19 }
```



PRO TIP - Scripting

The "using" section at the top of the script imports additional tools and resources, allowing the program to access predefined functionalities.

A **class** is a blueprint or template for creating objects in programming. It defines the properties (attributes) and behaviors (methods) of objects.

A **public class** is accessible from other parts of the program, meaning its properties and methods can be accessed and modified from anywhere.

A **private class** restricts access to its properties and methods only within the class itself, keeping them hidden from other parts of the program.

A **header** in programming typically refers to the introductory section of a file or script, containing essential information like the name of the file, author details, creation date, and a brief description of its purpose.

A **public variable** is a data container accessible and modified from any part of the program. It is like a public resource that anyone can use.

A **private variable**, on the other hand, can only be accessed and modified within the class it is declared in. It is like a secret resource that is only available to certain parts of the program.

A **float** is a data type used in programming to represent numbers with decimal points. It is used for storing values such as distances, measurements, or percentages.

Void can be used as the return type of a method (or a local function) to specify that the method doesn't return a value.

11

Now, let's start coding our bird. The code needs to get placed under **public class playerControls: monobehaviour**. This code defines variables. Add this code:

```
public class PlayerControls : MonoBehaviour
{
    //Game manager object
    [Header("Game Controller Object for controlling the game")]
    public GameController gameController;
    [Header("Velocity")]
    public float velocity = 1; // Corrected the header to "Velocity"
    //Physics for the bird
    private Rigidbody2D rb;
    //Height of the bird object on the y axis
    private float objectHeight;
```

Next, type inside the **void Start** function:

```
// Start is called before the first frame update
void Start()
{
    //Game Controller component
    gameController = GetComponent<GameController>();
    //Speed for the game is at a playing state
    Time.timeScale = 1;
    rb = GetComponent<Rigidbody2D>();
    //Object Height equals the size of the height of the sprite
    objectHeight = transform.GetComponent<SpriteRenderer>().bounds.size.y / 2;
}
```

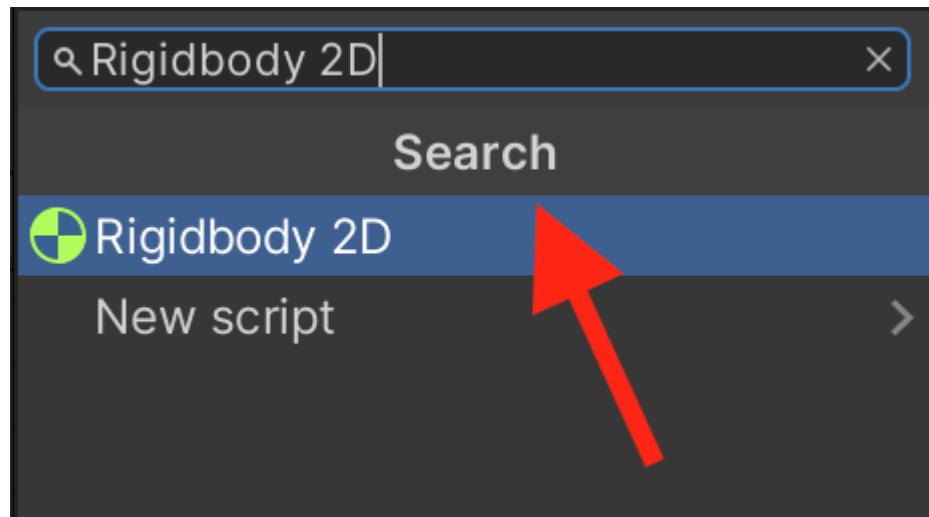
Now, type inside the **void Update** function:

```
// Update is called once per frame
void Update()
{
    //If the left mouse button is clicked
    if (Input.GetMouseButtonDown(0))
    {
        //The bird will float up on the Y axis
        //and float back down on Y axis
        rb.velocity = Vector2.up * velocity;
    }
}
```

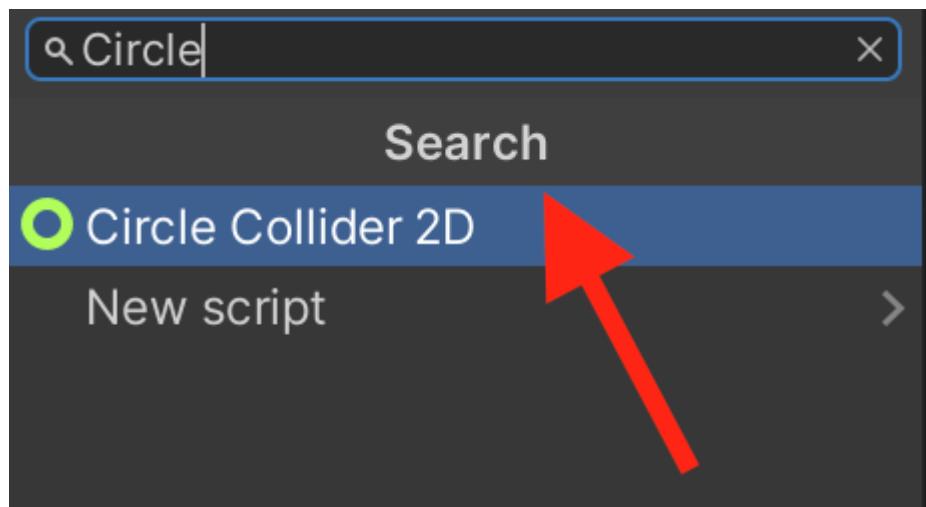
Now, save the script by pressing **Ctrl + S** and return to Unity!

12

We are almost ready to playtest the game, but we must do a few more things first! Select the **Bird** GameObject in the **Hierarchy** panel, then click **add component** in the **Inspector** menu. In the input field, type **Rigidbody 2D** and then select **Rigidbody 2D**.



- 13** Click **Add Component** once more. Then type **Circle Collider 2D** in the input field. Select **Circle Collider 2D**.



- 14** Finally, change the velocity under player controls to 5 in the inspector so the bird can float around. Make sure to change from **Play Focused** or **Play Unfocused** to **Play Maximized** before you click the Play button.

```
public class PlayerControls : MonoBehaviour
{
    //Game manager object
    [Header("Game Controller Object for controlling the game")]
    public GameController gameController;
    [Header("Default Score")]
    public float velocity = 5;
    //Physics for the bird
    private Rigidbody2D rb;
    //height of the bird object on the y axis
    private float objectHeight;

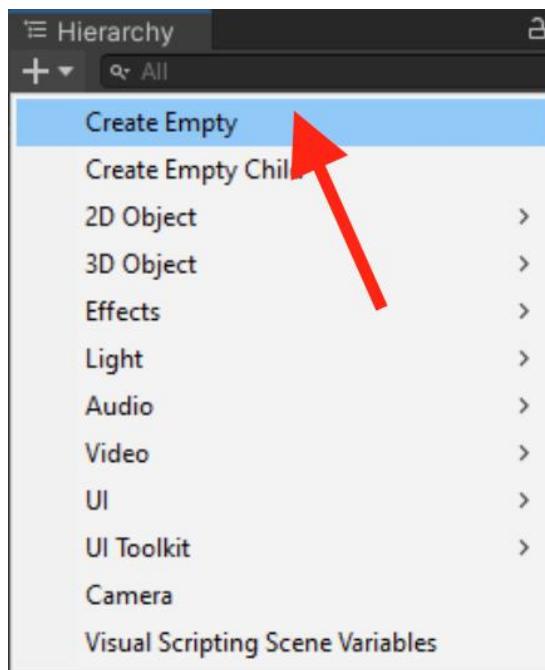
    // Start is called before the first frame update
    void Start()
    {
    }
}
```

Now, let's playtest your game! Continue clicking on the game screen to make the bird fly.

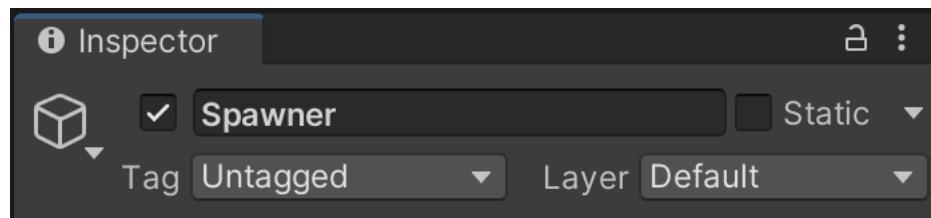
- 15** Stop the game by selecting the play button again.

16

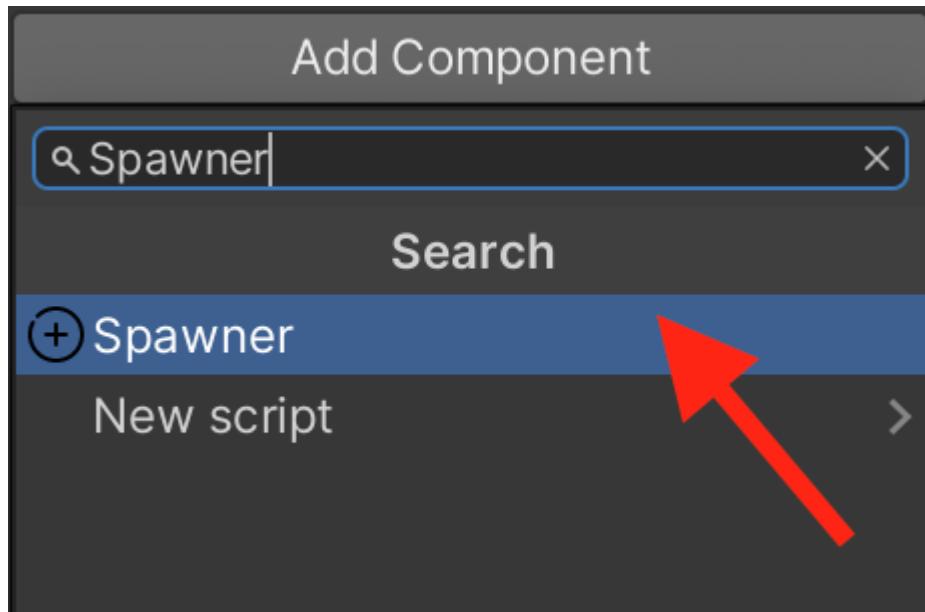
With our bird set up, let's add spikes into the game. In the **Hierarchy** panel, click the **+** button, then select **Create Empty**.



In the Inspector menu, rename the empty game object, "Spawner".



- 17** In the **Inspector** window for the Spawner, select **Add Component**. In the search box, type **Spawner** and select the **Spawner** script component.



- 18** Open the Spawner script. Add the following to the script inside the **class** before the void start function:

```
public class Spawner : MonoBehaviour
{
    //Object that we will attach to the script for spawning object
    [Header("Spikes Object for controlling the game")]
    public GameObject spikes;
    //Height position of the spikes
    [Header("Default Height")]
    public float height;
    // Start is called before the first frame update
    void Start()
    {
    }
```

Next, type inside the **void Start** function:

```
// Start is called before the first frame update
void Start()
{
    //Start function repeating every 4 seconds
    InvokeRepeating("InstantiateObjects", 1f, 4f);
}
```

This will help us set up the timer to decide when the next object will appear.

Next, type inside the **void Update** function:

```
void Update()
{
    //Position for the gameobjects
    spikes.transform.position = new Vector3(5, Random.Range(-height, height), 0);
}
```

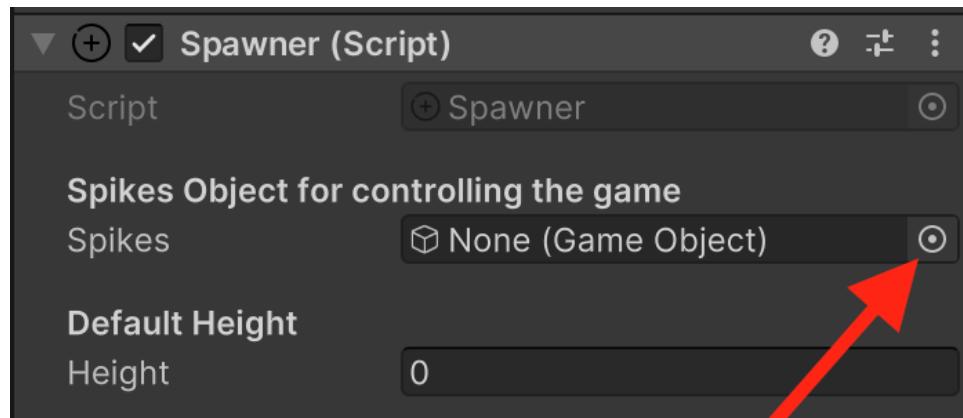
Create a new void function called **InstantiateObjects** inside the class and just after the **void Update** function. Add this code inside:

```
//InstantiateObjects Function
void InstantiateObjects()
{
    //Spawn object by position and rotation
    Instantiate(spikes, spikes.transform.position, spikes.transform.rotation);
}
```

Save the script (**Ctrl + S**) and return to Unity!

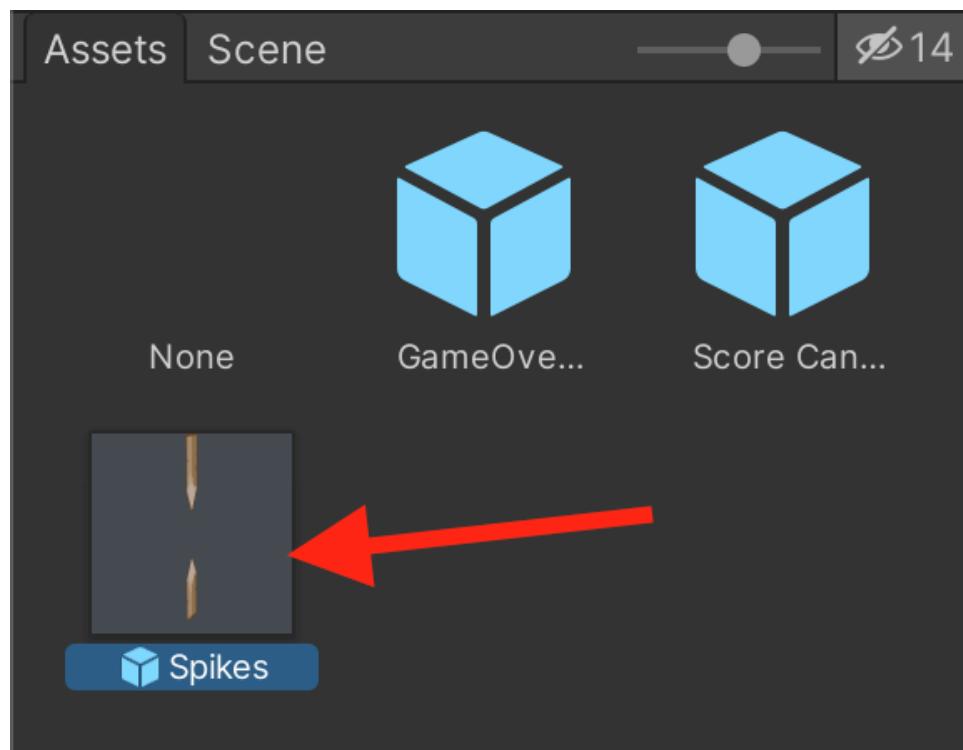
19

Using our updated script, look in the **Inspector** window. The Spawner has a **GameObject** input field. Click the properties button next to the input field as shown below.

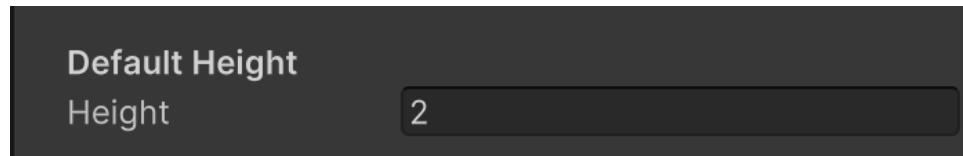


20

Select **Assets** on the top left of the window that appeared. You should see a menu like the one below. Then select the **Spikes** object.



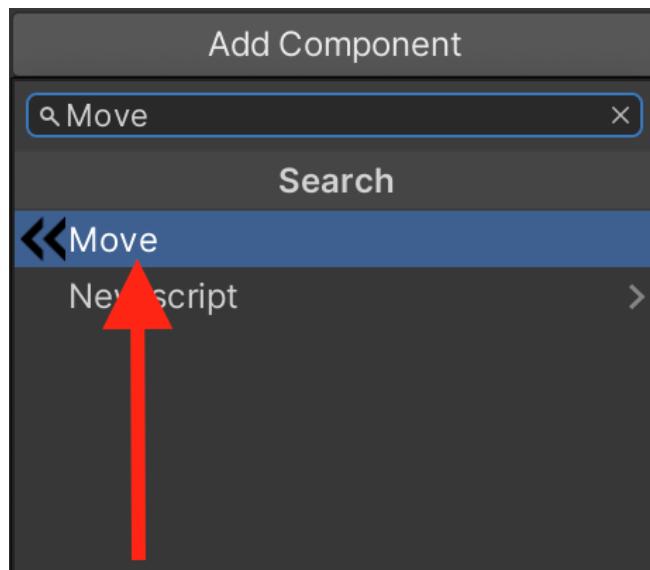
- 21** In the **Inspector**, navigate to the **Spawner** component and adjust the height value from **0** to **2**.



- 22** Now, let's go ahead and play the game. While playing, look at the Hierarchy. You should notice that the objects are being generated but aren't moving across the screen.

- 23** Stop the game.

- 24** Switch back to the scene tab. Go into the **Prefabs** folder and double-click the **Spikes** prefab. Make sure the **Spikes** parent object is selected and shown in the **inspector** menu. Then, click **Add Component** in this **inspector** menu and type **Move** in the search box. Select the **Move** script component.



Open the Move script and add the following:

```
public class Move : MonoBehaviour
{
    [Header("Default Speed")]
    //Speed for the movement
    public float speed;
    // Start is called before the first frame update
    void Start()
    {

    }
```

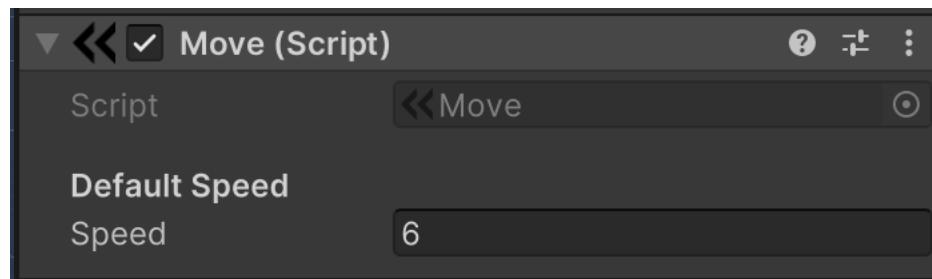
Next, delete the **void Start** function and type inside the **void Update** function:

```
public class Move : MonoBehaviour
{
    [Header("Default Speed")]
    //Speed for the movement
    public float speed;

    // Update is called once per frame
    void Update()
    {
        //Transform the object to move left
        //according to the axis and speed
        transform.position += Vector3.left * speed * Time.deltaTime;
    }
}
```

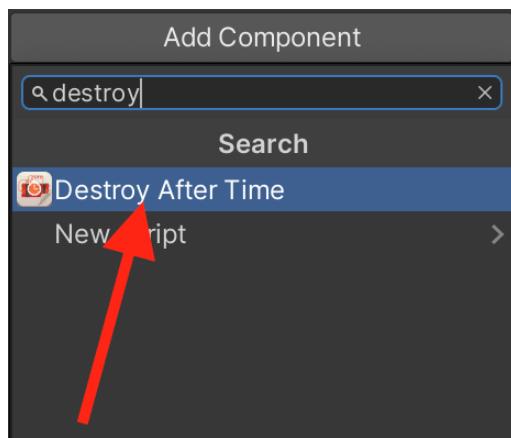
Save the script (**Ctrl + S**) and return to Unity!

25 With the **Spikes** parent object still selected in the inspector, change the speed value from **0** to **6**.



26

Now, with the **Spikes** parent object still selected, click **Add Component**. In the search box, type **Destroy** and select the **Destroy After Time** script.



Open the **Destroy After Time** script and add the following:

```
public class DestroyAfterTime : MonoBehaviour
{
    //After this time, the object will be destroyed
    [Header("Default Destruction Time")]
    public float timeToDestruction;
    // Start is called before the first frame update
    void Start()
    {
```

Next, type inside the **void Start** function:

```
void Start()
{
    //Execute DestroyObject function based on timeToDestruction
    Invoke("DestroyObject", timeToDestruction);
}
```

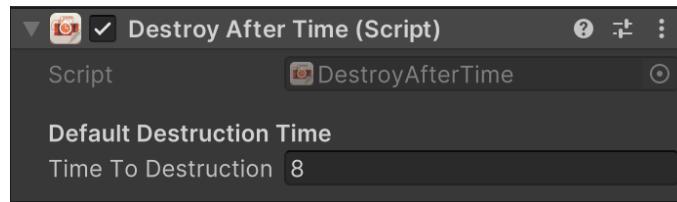
Invoke allows you to specify the time when a function is run.

Create a new void function called **DestroyObject**, then add this:

```
//This function will destroy this object
void DestroyObject()
{
    //Destroy Object
    Destroy(gameObject);
}
```

You can delete the **void Update** function as this script does not require it in this context. Once finished, save the script and return to Unity!

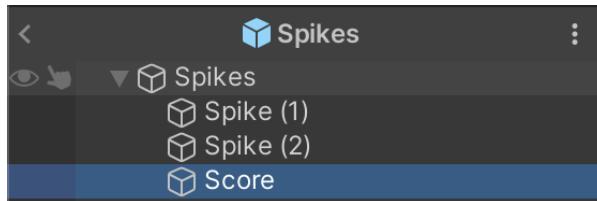
27 Now, in the **Destroy After Time** component, change the value for Time to Destruction from **0** to **8**.



28 Now, play the game!

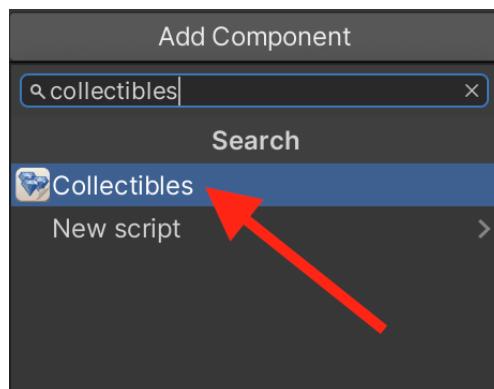
29 Stop the game.

30 Double-click the **Spikes** prefab and this time select the **Score** child object.



31

Click **Add Component**. Type **Collectibles** in the search box and select the **Collectibles** script.

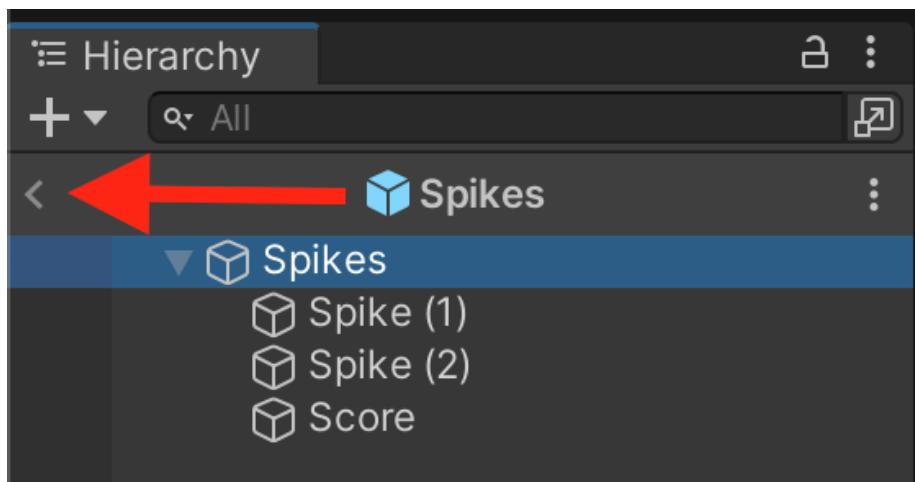


Open the **Collectibles** script. Delete both **void Start** and **Update** functions. Inside the class, create a new void function called **OnTriggerEnter2D**, add this:

```
public class Collectibles : MonoBehaviour
{
    // Start is called before the first frame update
    //If an object collides with a trigger
    private void OnTriggerEnter2D(Collider2D collision)
    {
        //Add score
        Score.score++;
    }
}
```

Save the script and return to Unity!

- 32** In the Hierarchy panel, click the arrow shown blown to return to Scene mode.



- 33** Now, play the game. Try to aim between the spikes to add points to the score.

- 34** Stop the game.

35 Before moving on, let's add a function for if the player touches the two spikes or the ground, then the game is over. Reopen the **Player Controls** script.

Create a new void function inside the class below the void Update function called **void OnCollisionEnter2D**, afterwards add this:

```
// Update is called once per frame
void Update()
{
    if (Input.GetMouseButtonDown (0))
    {
        rb.velocity = Vector2.up * velocity;
    }
}

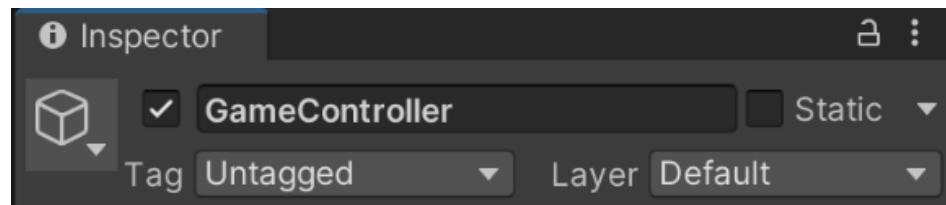
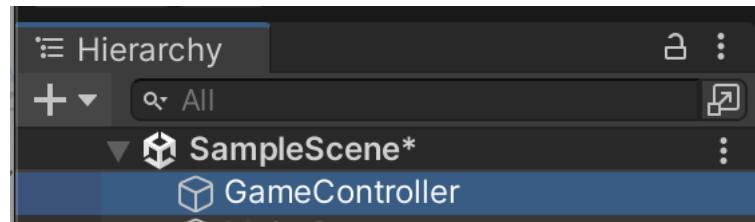
//Function where the player collides with an object
private void OnCollisionEnter2D(Collision2D collision)
{
    if(collision.gameObject.tag == "HighSpike" || collision.gameObject.tag == "LowSpike" || collision.gameObject.tag == "Ground")
    {
        //Game is at a stopping state
        Time.timeScale = 0;
    }
}
```

Save the script and return to Unity!

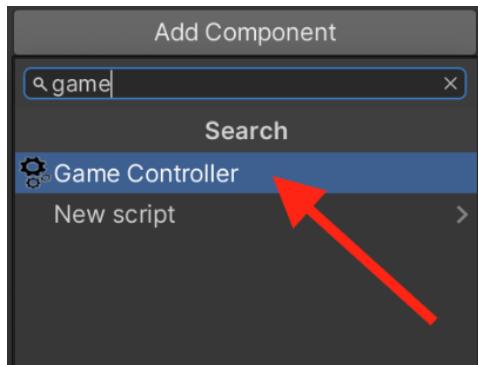
36 Now, try playing the game. You will notice that if the bird touches the ground or spikes, then the game will be stopped.

37 Stop the game.

38 Let's add a new game object. Go to **Hierarchy** and click the **+** button. Select "**Create Empty**". Then, rename the new **GameObject** to **GameController**.



- 39** With the **GameController** object selected, click **Add Component**. Type **Game** in the search box and select the **Game Controller** script component.



Open the **Game Controller** script. Add the following to the script inside the class before the void Start function:

```
public class GameController : MonoBehaviour
{
    [Header("References")]
    //Game Over Canvas that is used for the game
    public GameObject gameOverCanvas;

    //Score Canvas that is used for the game
    public GameObject scoreCanvas;

    //Spawner object that is used for the game
    public GameObject spawner;

    // Start is called before the first frame update
    void Start()
    {
    }
}
```

Next, type inside the **void Start** function:

```
void Start()
{
    //Speed for the game is at a playing state
    Time.timeScale = 1;
    //Score is visible
    scoreCanvas.SetActive(true);
    //Game Over UI is invisible
    gameOverCanvas.SetActive(false);
    //The spawner is shown in the game
    spawner.SetActive(true);
}
```

Create a new void function called **void GameOver** inside the class and after the void Start function. Afterwards add this:

```
void Start()
{
    //Speed for the game is at a playing state
    Time.timeScale = 1;
    //Score is visible
    scoreCanvas.SetActive(true);
    //Game Over UI is invisible
    gameOverCanvas.SetActive(false);
    //The spawner is shown in the game
    spawner.SetActive(true);
}
public void GameOver()
{
    //Game Over UI is visible
    gameOverCanvas.SetActive(true);
    //The spawner is now invisible in the game
    spawner.SetActive(false);
    //The speed for the game is now at a stopping state
    Time.timeScale = 0;
}
```

The **void Update** function can be deleted as it is not needed in this context. Once you do that save your script and return to Unity!

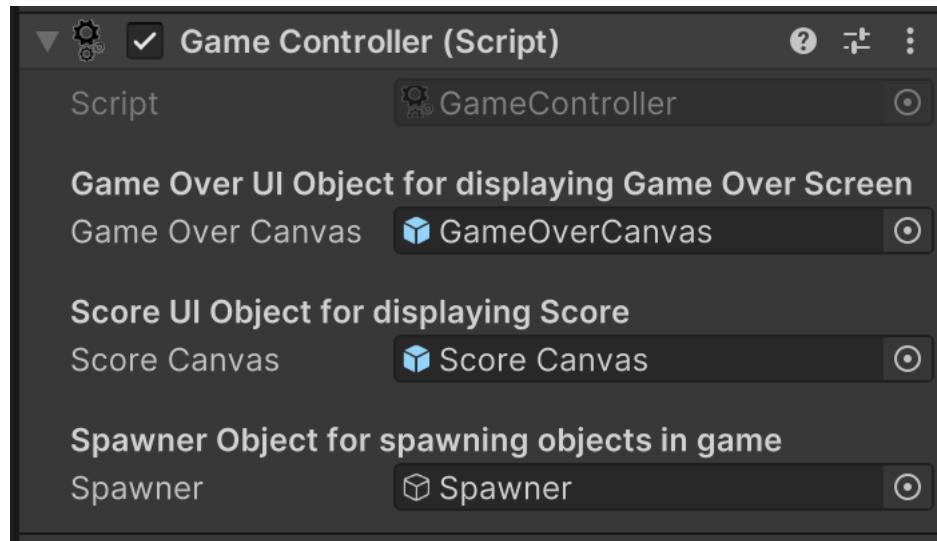
40 Before moving on, let's update the code in the player controls for the **Game Over Canvas** to appear. Select the Bird game object in the hierarchy panel then open the Bird's **Player Controls** Script.

Inside the **void OnCollisionEnter2D** method, replace the code inside the if statement with the code below:

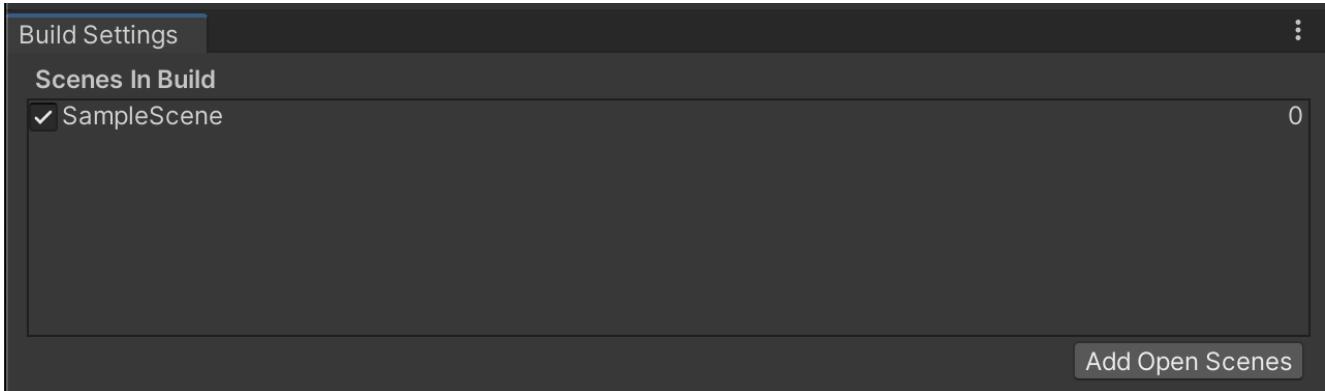
```
//Game Over function is called from the game manager  
GameObject.Find("GameController").GetComponent<GameController>().GameOver();
```

Save the script and return to Unity!

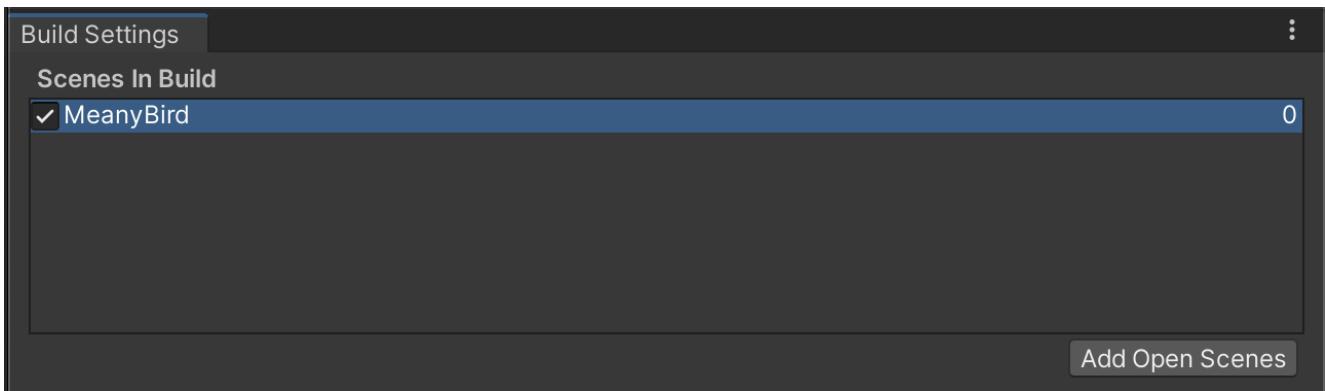
41 For the Game Controller script component, let's select our objects. Make sure the checkbox next to the Game Controller (Script) is activated, then within the Scene menu, select the **GameOverCanvas**, **Score Canvas**, and the **Spawner**.



- 42** To get the game over button to work we will have to adjust the build settings. Click the File tab then Build Settings. Select the Scenes/SampleScene and delete it.



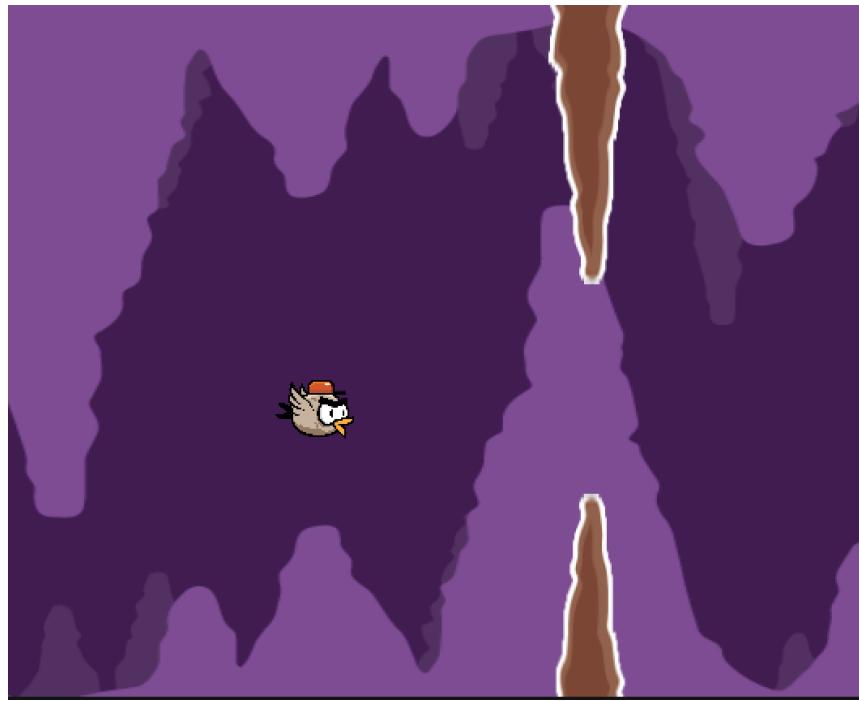
- 43** Once deleted, you want to click the **Add Open Scenes** button. This will allow the game to restart.



- 44** Play the game! Make sure that it looks like everything's working.

- 45** Stop the game. Save your game and export it. Submit your game.

Prove Yourself: Meaner Bird



Swapping out assets in Unity is a lot easier than dodging stalactites and stalagmites while flying through a cave. (*Stalactites are the rocks on the ceiling, while stalagmites are the ones sticking out of the ground.*)

Go into the folder where the Unity assets are provided (from the Download Unity Game Assets folder in SharePoint). The assets you'll need from this folder are:

"Activity 03 Prove Yourself - caveback1.png", "Activity 03 Prove Yourself - caveback2.png", and "Activity 03 Prove Yourself - stalagmite_1.png".

In this Prove Yourself we are going to accomplish:

- Use "**Activity 03 Prove Yourself - caveback1 .png**" and "**Activity 03 Prove Yourself - caveback2. png**" as shown above as the background.
- Apply the "**Activity 03 Prove Yourself - stalagmite_1.png**" image to the obstacles in your existing Meany Bird game.

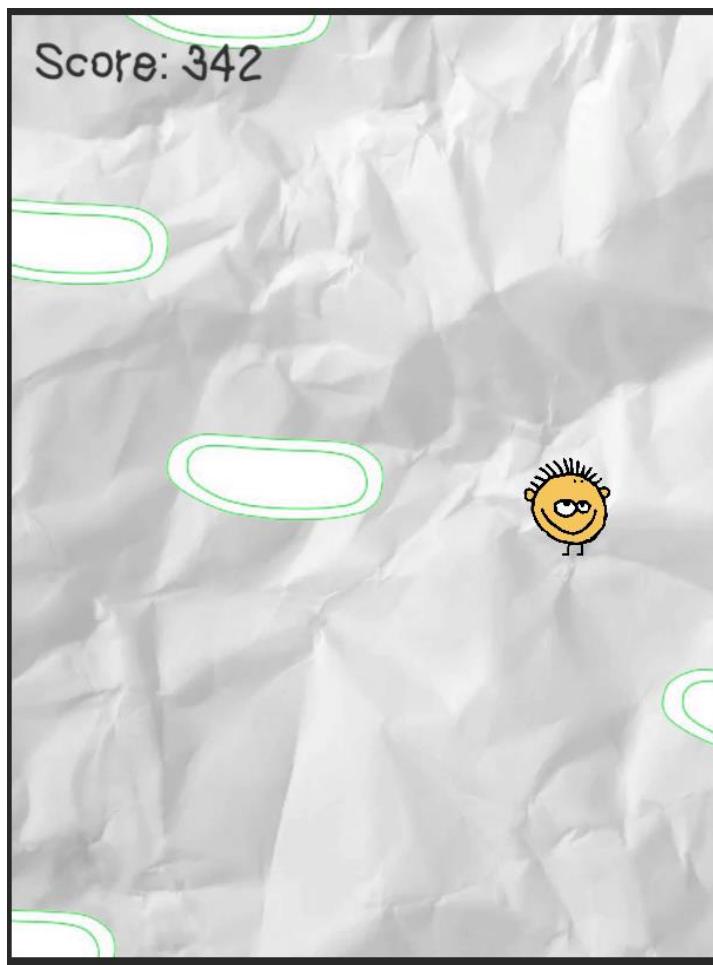
To get started follow the steps below:

- 1. Add the "caveback1", "caveback2", and "stalagmite1" images to your sprites folder!**
- 2. Select the background object in the hierarchy and change its sprite to caveback1**
- 3. Make the background image fit correctly by adjusting the scale property in the transform component to 1.7 for both X and Y.**
- 4. Duplicate your background object and change the sprite of this duplicated background object to caveback2. Then set the z position to -1.**
- 5. Update the z position of your bird object to -2 in order to get it to appear above both backgrounds.**
- 6. Use Unity's move tool to move your "Ground" gameobject down. It should be barely on screen. The Y position should be somewhere around 0.67-0.70.**
- 7. Duplicate your ground object and rename it Ceiling. Change the Y position of this new GameObject to around 12.55.**
- 8. In both the "Ground" and "Ceiling" objects disable the sprite renderer components to hide the objects. They are still there even though the camera cannot seem them!**
- 9. Open the spikes prefab. Select "Spike (1)" in the hierarchy panel and change the property of its sprite to "stalagmite_1" Set the Y scale to around 5.32.**
- 10. Click the Edit Collider button for Box Collider 2D and then utilize your mouse to drag the dots. Repeat the same steps (12 and 13) for Spike 2.**

Physics

When we throw a ball, there are a lot of forces at work that determine what happens to that ball and where it goes. Of course, the energy that you use to throw the ball is present, but there is also gravity pulling it down to the ground. In addition, there is the air that the ball is moving through, slowing it down. Also, there is rotation which can affect whether the ball moves in the direction you intended it to go.

Learning about forces gives you an idea of where the ball is going to go. Unity has a large set of tools to simulate physics and knowing about them can help you make better games.



What Goes Up...

Centuries ago, a famous mathematician and physicist by the name of Sir Isaac Newton discovered the laws of motion and showed the world how it drove everything from rocks to the planets. What does that have to do with Unity? Newton showed us the mathematics that allows us to simulate the motion of objects in our computer environment. While we sadly won't be going into Newton's laws in this book, everything that we do regarding physics started with what he taught humans long ago.

The Nature of Things

Objects with mass not only move, but they also accelerate. When they collide, they bounce based on their relative mass and energy. Physics helps us predict what happens when a force of any kind is applied to an object. While we don't need to understand the mathematics of physics to use it, knowing about the natural laws of mechanics and motion helps us not only figure out why things move, but how to make them move more accurately.

Physics in Unity

For an object in Unity to act under the control of physics, it must have a **Rigidbody** component. This allows the object to respond to forces such as gravity and torque. The Rigidbody component allows you to choose the mass of an object (objects with more mass are more resistant to force from objects with less mass) and drag (how air resistance can slow it down).

Throwing it Out the Window

One thing that you can do on a computer that you can't do in the real world is ignore any of the laws of physics that get in our way. We can decide not to allow an object to be affected by forces like gravity, or to have it magically float in the air. We can make an object immovable, so

that any object that collides with it, no matter the mass, doesn't make it budge. We can prevent it from rotating in just one or every axis.

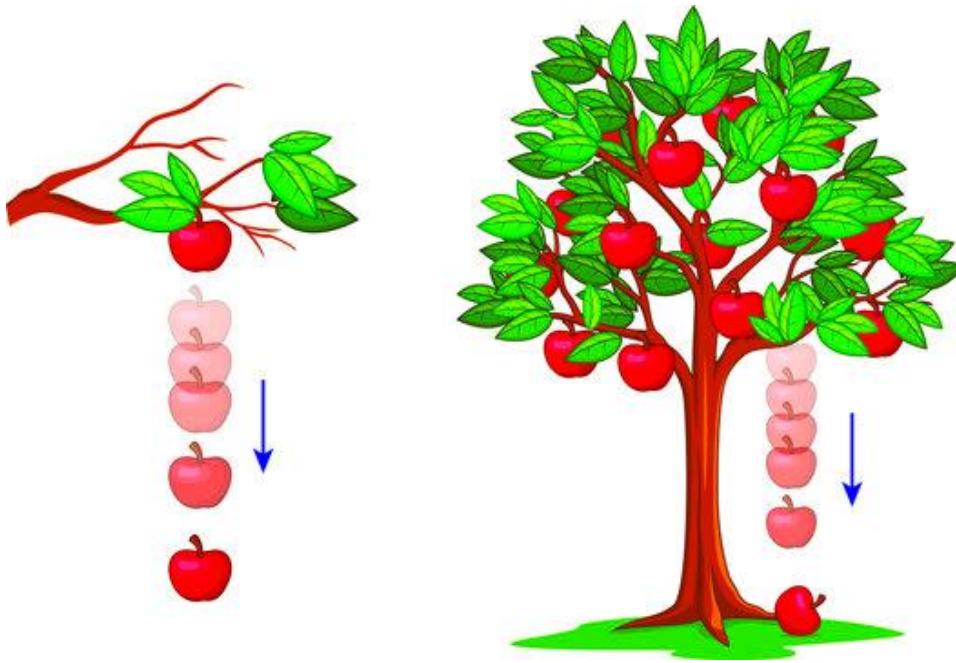


Image Source: https://stock.adobe.com/search/images?k=newton+apple+tree&asset_id=517182457

Colliders

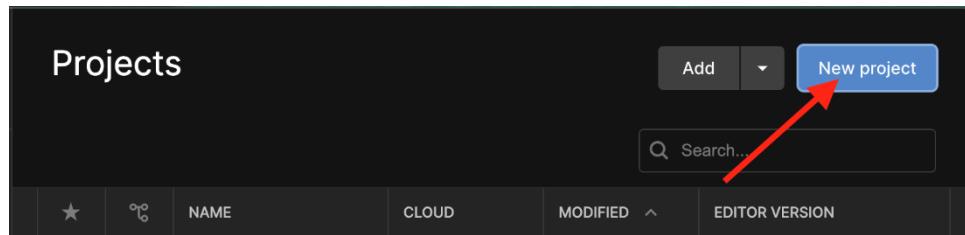
The Rigidbody is just one part of how Unity handles physics. Another part is the collider, which determines how and when one object touches another. As you might guess, it takes a lot of processing to keep track of all the edges of all the objects in a virtual world. To make things run smoother, Unity has simplified colliders that require less computations.

For instance, a model of a truck has many surfaces, from the top to the windshield, down to the wheels. Yet in a game, we can use a six-sided box to represent the top, bottom, and sides of the truck. As far as the game is concerned, it works just as well, and with the extra processing power, you can have many more trucks in your scene!

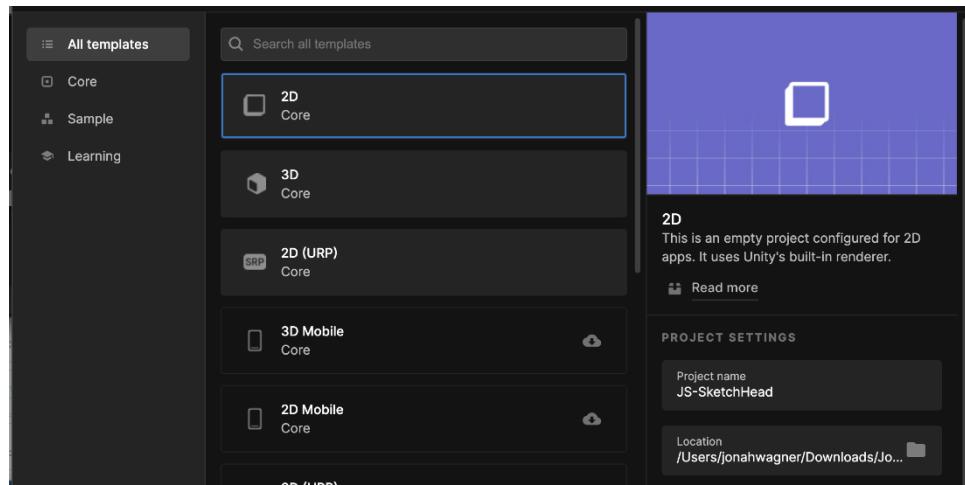
Activity 4: Sketch Head

In this activity, we will continue using physics and colliders, only this time we will add more features to the game, such as a “camera follow” mechanic and a generator object!

- 1 Open the Unity Hub application on your computer. Select the blue **New Project** button in the right-hand corner.

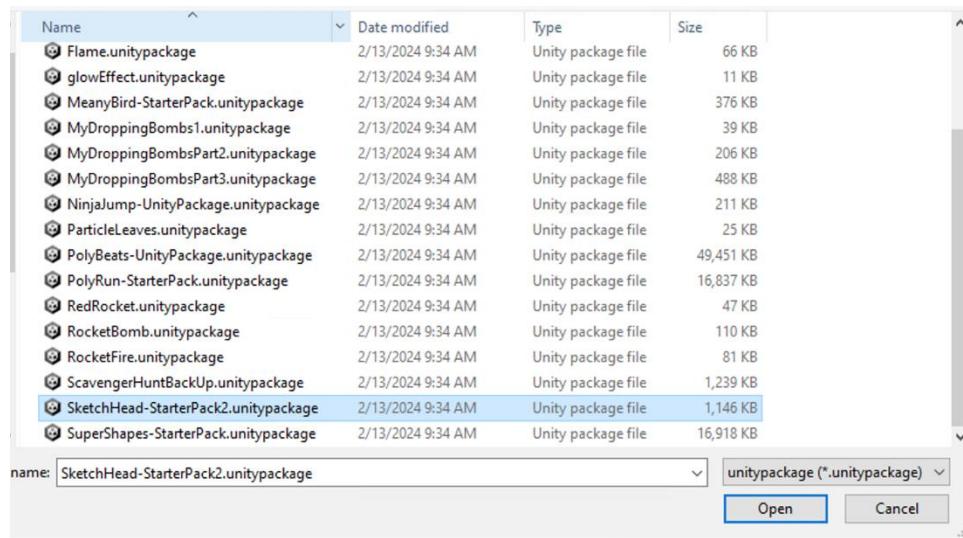
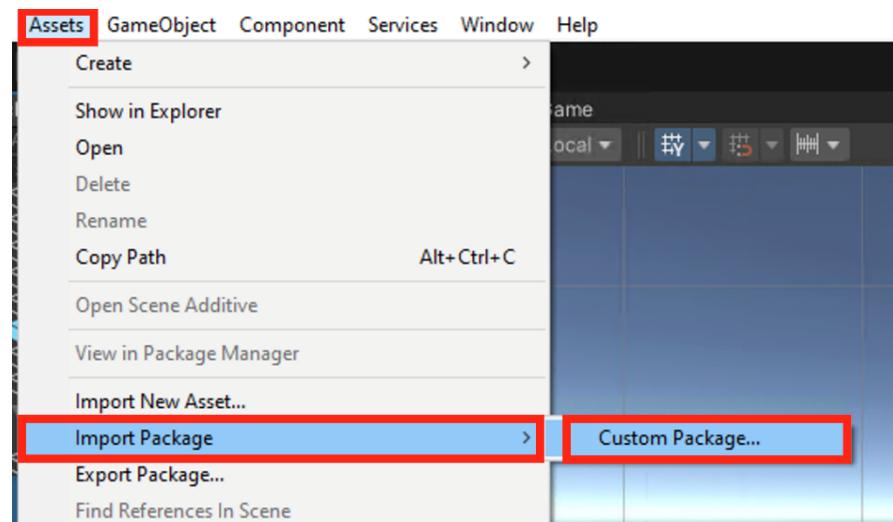


- 2 In the templates column select **2D**. Next, type your first and last initials and the name of the project. For example, John Smith would save the project as JS-SketchHead. Select the folder location where you typically save your Unity files.

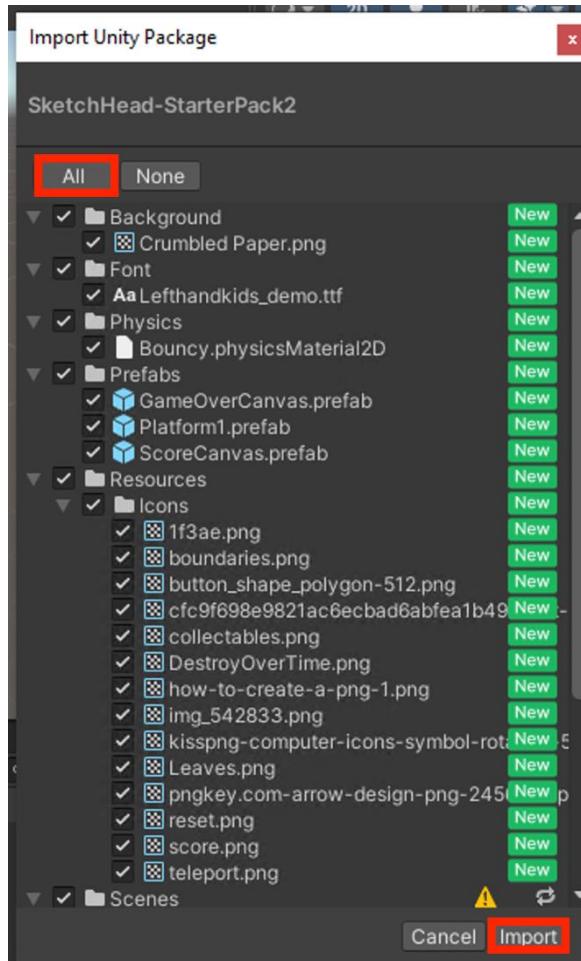


- 3** As soon as Unity has loaded all the necessary assets into the program, it will open. As you can see, we don't have assets or scripts in the project. Let's fix that now.

Go to the **Assets** menu. Select **Import Package** and click **Custom Package**. This allows us to import any package that has been compressed in Unity. In the folder with the Unity Assets, locate and open the **Activity 04 - SketchHead - StarterPack** Unity package. Once you do this, a menu will open displaying everything inside the package.



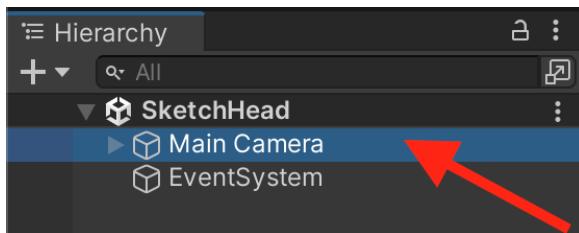
- 4** Since these packages are specially made for each game, we want all materials inside the Unity package selected. If they are not all selected, click the **All** button. Click **Import** and everything will be imported into the project.



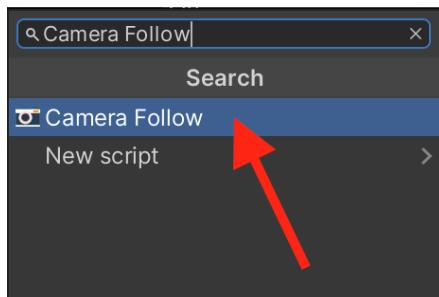
- 5** Now click **Scenes** under the **Assets** window and open the **SketchHead** scene by double-clicking. Like the last game, in the Game window, ensure Free Aspect is selected then click the plus and change the resolution to **600x800**.

6 Let's start configuring the Main Camera game object before we start to configure our player object. We need to select the **Main Camera** in the Hierarchy. This step is needed because the camera will focus on the player by default.

Select the **Main Camera**, go to the Inspector window and click **Add Component**.



7 Once you click **Add Component**, type **Camera Follow** in the search box. Select the **Camera Follow** script.



8 Open the script and add the following:

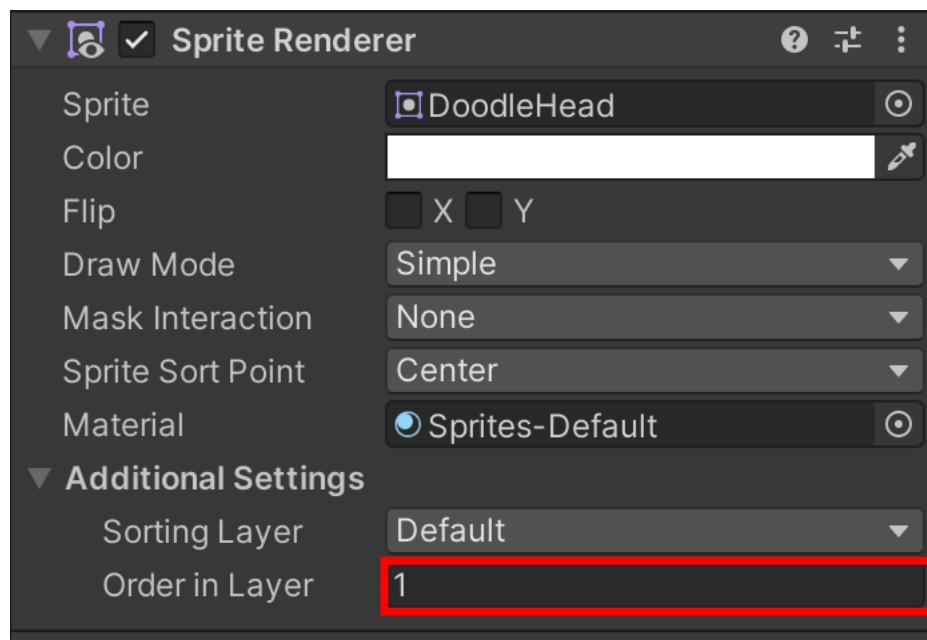
```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class CameraFollow : MonoBehaviour
6  {
7      //Target object position
8      [Header ("Target Object")]
9      public Transform target;
10     // Start is called before the first frame update
11     void Start()
12     {
13
14     }
```

Next type inside the **void Update** function,

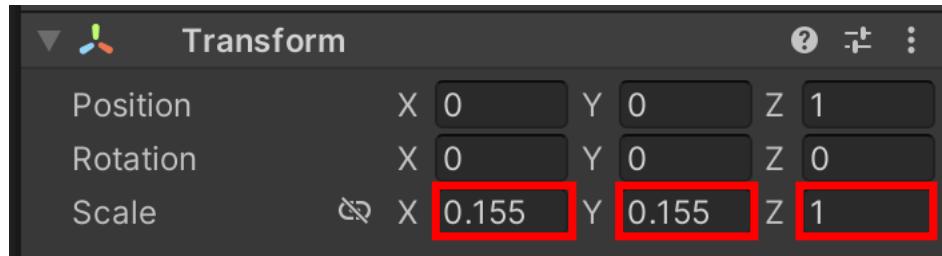
```
// Update is called once per frame
Unity Message | 0 references
void Update()
{
    if (target.position.y > transform.position.y)
    {
        transform.position = new Vector3(0, target.position.y, -10f);
    }
}
```

You can delete the void Start function as it is not needed in this activity. Then save the script and return to Unity!

9 Now, go to the **Sprites** folder and select the **DoodleHead** sprite. Drag the **DoodleHead** sprite into the **Hierarchy**. It should now be shown in the Game view. If not, change the order in layer from **0** to **1** in the **Sprite Renderer** component.

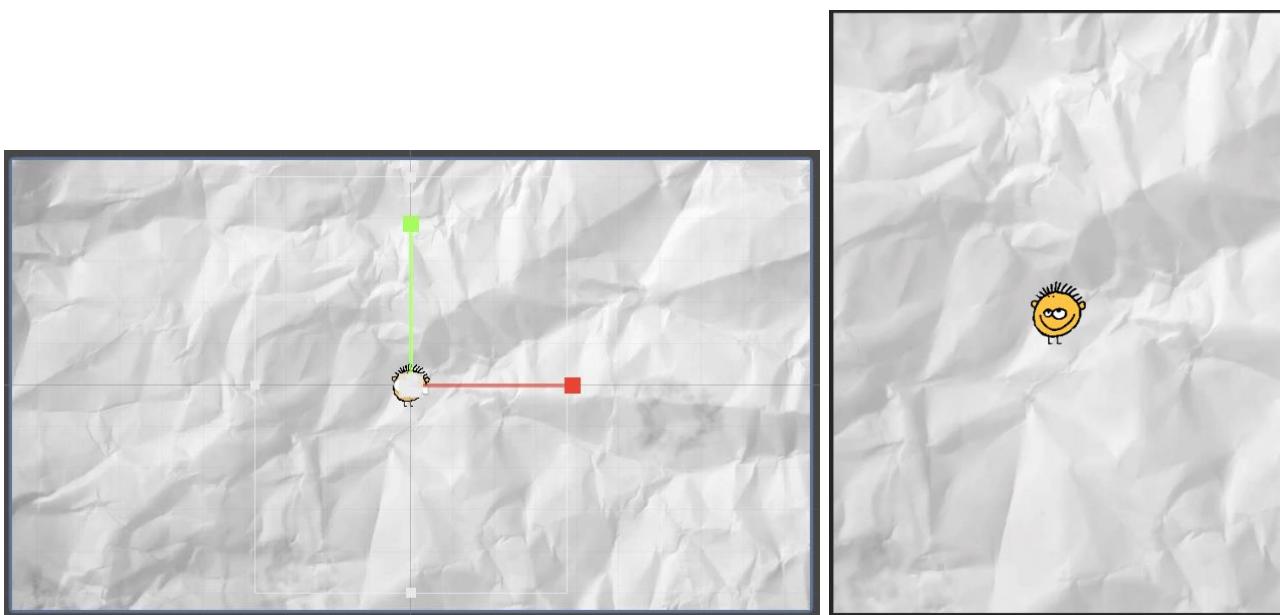


10 Now the DoodleHead is too big for the Game view, so let's change its size. In the Inspector window, with the **DoodleHead** still selected, change the size of the DoodleHead to these scale values:

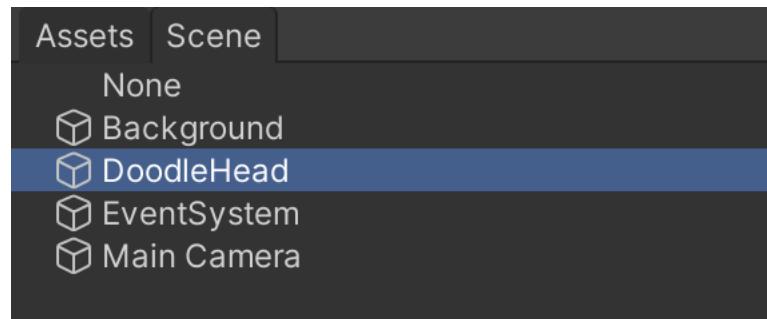
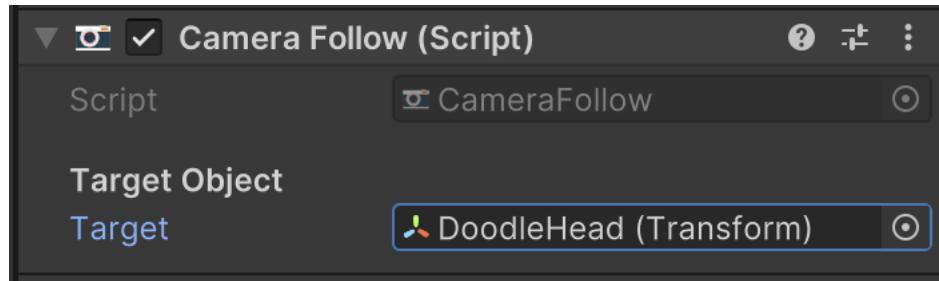


Scale values: X: 0.155, Y: 0.155, Z: 1

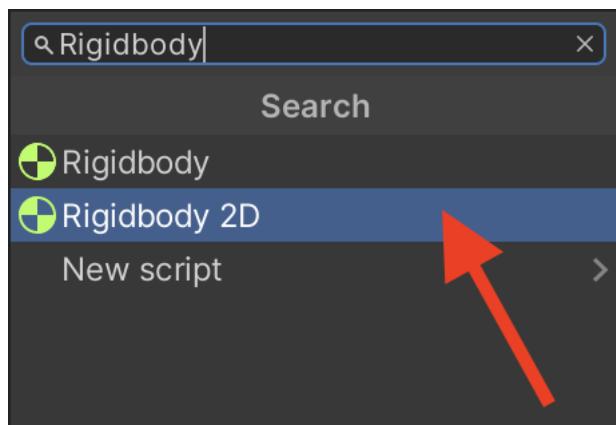
Your DoodleHead should look like this in both the Scene and Game windows:



- 11** Select the **Main Camera** object, where we will select our target object in the Camera Follow script component. Select the **Properties** button to the right of the text box and in the Scene menu, select **DoodleHead**.



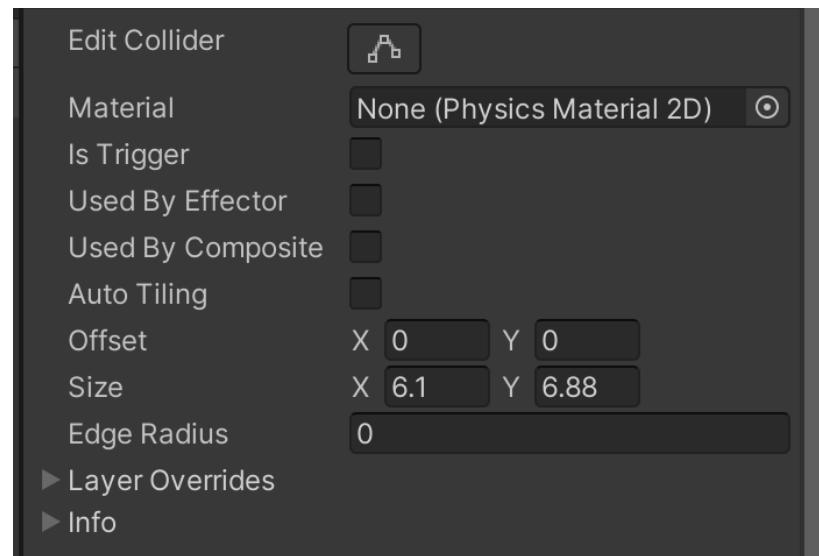
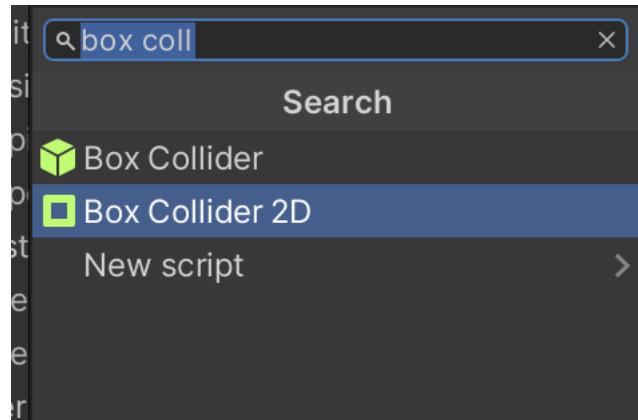
- 12** Now, select the **DoodleHead** object and click **Add Component**. Type **Rigidbody** in the search box and select the **Rigidbody 2D** component.



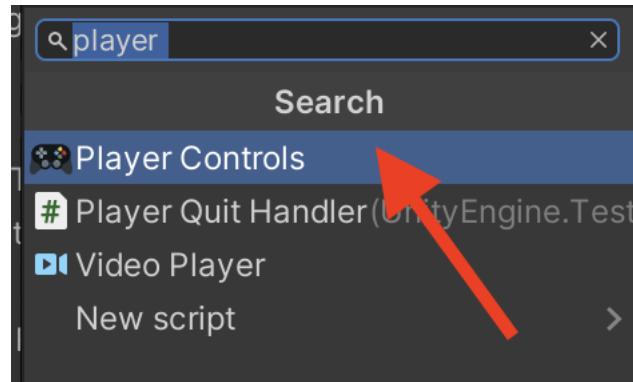
- 13** Select **Constraints** in the **Rigidbody2D** component. Then, select the check box labeled **Freeze Rotation** for the Z axis. Constraints are the rules or limitations applied to the movement and interaction of objects that have a Rigidbody component.



- 14** Afterward, click Add Component with the DoodleHead still selected. Then type Box Collider 2D in the search box and select Box Collider 2D. Match the settings to the image provided below.



- 15** Ensure that the **DoodleHead** is still selected, then select the **Add Component** button in the Inspector menu. Then, type **Player** in the search bar and select **Player Controls**.



- 16** Open the Player Controls script and add the following code inside the class below the void Start function:

```
public class PlayerControls : MonoBehaviour
{
    [Header("Rigidbody Settings")]
    [Tooltip("Rigidbody2D object that is stored")]
    public Rigidbody2D rb;

    [Tooltip("Downward speed of the object")]
    public float downSpeed = 20f;

    [Header("Movement Settings")]
    [Tooltip("Movement speed of the object")]
    public float movementSpeed = 10f;

    [Tooltip("Movement direction of the object")]
    public float movement = 0f;

    private SpriteRenderer spriteRenderer;

    // Start is called before the first frame update
    void Start()
```

Next, type inside the **void Start** function:

```
// Start is called before the first frame update
void Start()
{
    // Store the components in variables
    rb = GetComponent<Rigidbody2D>();
    spriteRenderer = GetComponent<SpriteRenderer>();
}
```

Next, type inside the **void Update** function:

```
// Update is called once per frame
Unity Message | 0 references
void Update()
{
    //Get the horizontal input and calculate the movement direction
    movement = Input.GetAxis("Horizontal") * movementSpeed;

    //Flip the sprite based on movement direction
    spriteRenderer.flipX = movement < 0;
}
```

Create a new **void FixedUpdate** function and add this:

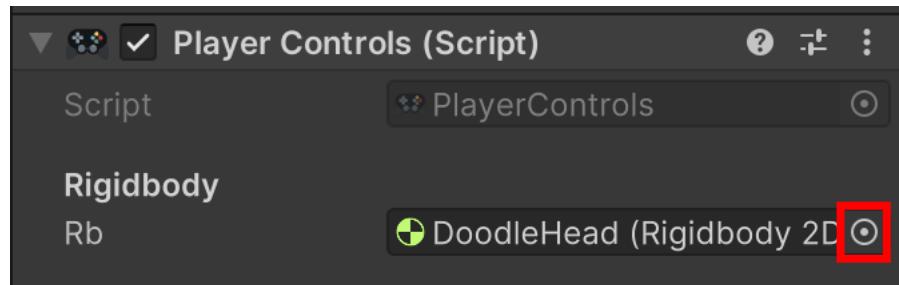
```
void FixedUpdate()
{
    //Vector2 which is (x,y) velocity
    //equals to the velocity of the rigidbody2D
    Vector2 velocity = rb.velocity;
    //Velocity of the x axis equals to
    //the direction movement on the x axis
    // of the character.
    velocity.x = movement;
    //Rigidbody2D velocity equals to
    //velocity of the object
    rb.velocity = velocity;
}
```

Create a new **void OnCollisionEnter2D** function and add this:

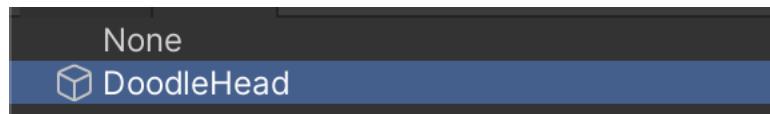
```
private void OnCollisionEnter2D(Collision2D collision)
{
    // If the object collides with something and is moving downwards,
    // adjust the y-component of its velocity
    if (rb.velocity.y <= 0)
    {
        rb.velocity = new Vector2(rb.velocity.x, downSpeed);
    }
}
```

You can now save the script and return to Unity!

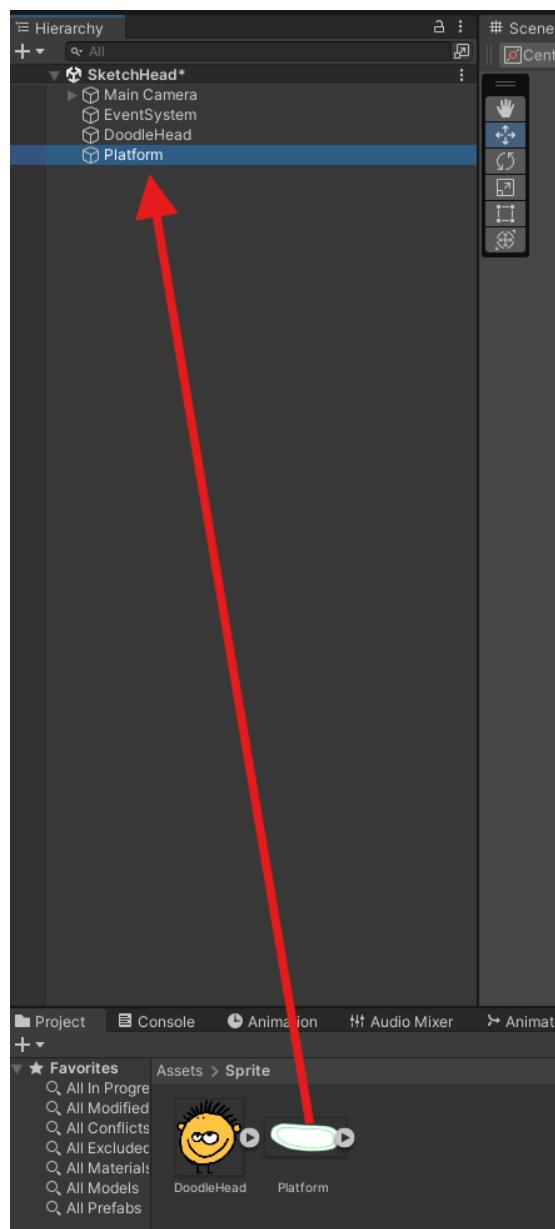
- 17** In the **Player Controls** component, click the **Properties** button next to the **Rigidbody** input field.

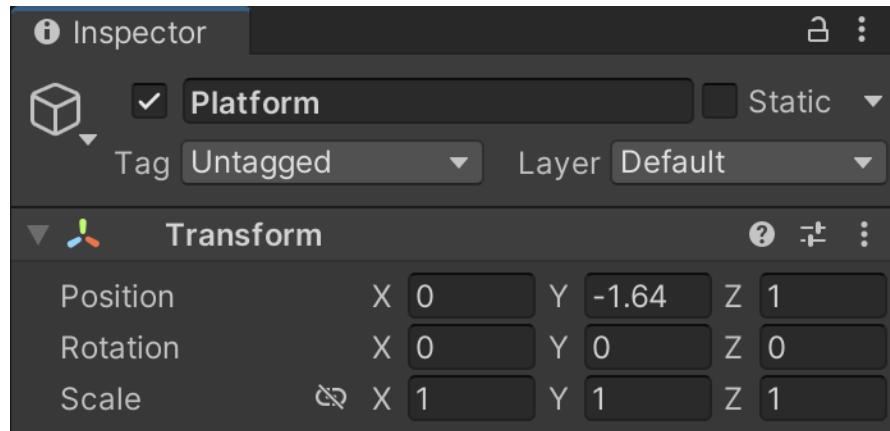


18 Select the **DoodleHead** object.



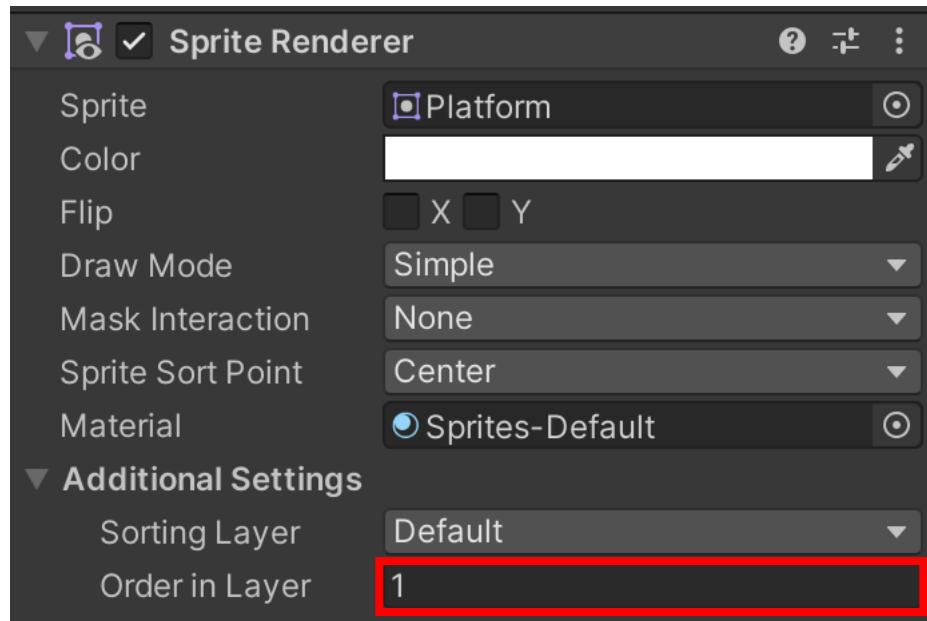
19 Open the **Sprites** folder, then drag the platform to the **Hierarchy**. Move the platform under the **DoodleHead**.





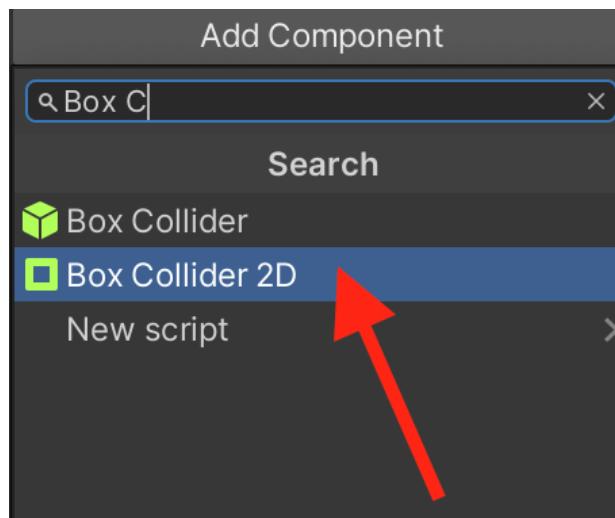
Position values: X: 0, Y: -1.64, Z: 1

Sometimes the sprite still may not show up. In that case, change the order in layer from **0** to **1** in the Sprite Renderer component.

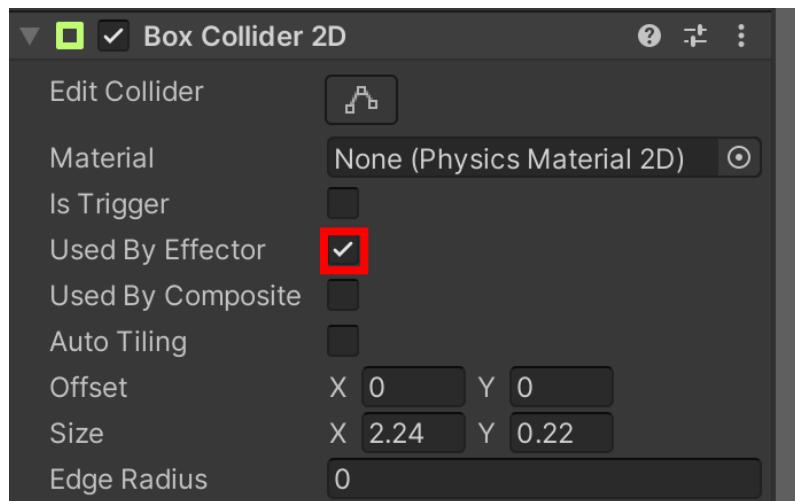


20

With the platform still selected, click **Add Component**. In the search box, type **Box Collider** and select **Box Collider 2D**.

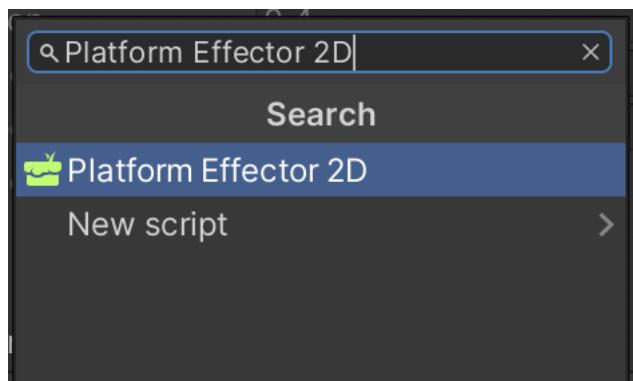


Update the **X** and **Y** size of the Box Collider 2D component to match the image. Select the **Used By Effector** check box.

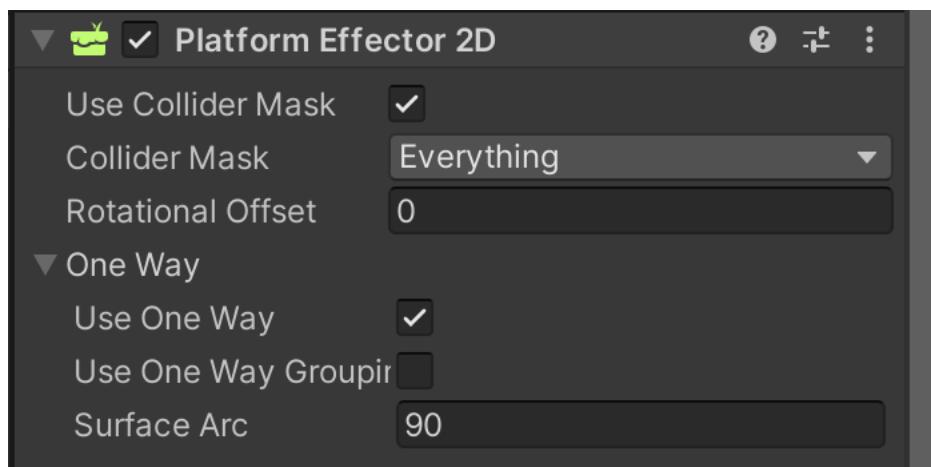


21

Click **Add Component** again. Type **Platform Effector 2D** into the search box and select **Platform Effector 2D**.



Once added, change the **Surface Arc** to 90.



22

Now, let's play the scene. After clicking play, switch back to the scene view while the game is playing. You should see the DoodleHead character jump up very high. You will notice that when the DoodleHead's Y position is less than the Camera position, the Camera will stop following.

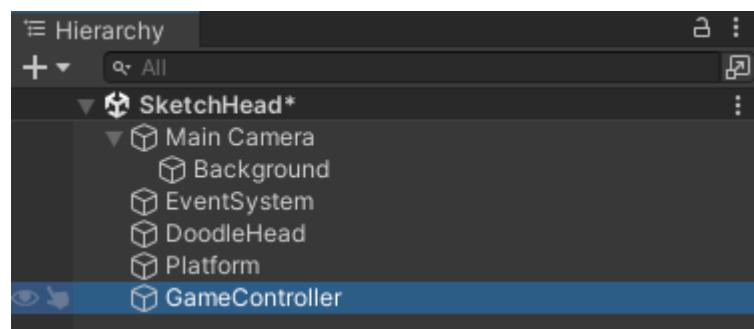
23

Stop the game.

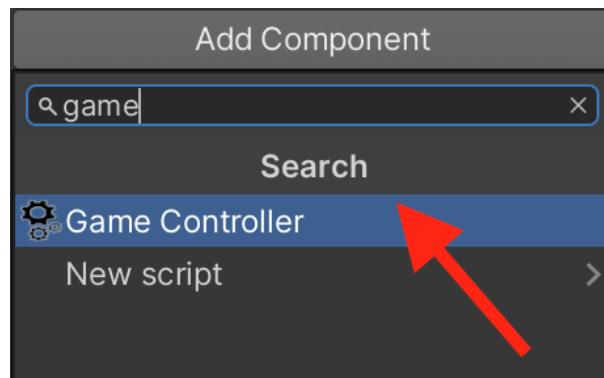
24

Now it's time to spawn more platforms to collide on. This will be done using a method called **object pooling**. This method involves spawning in many objects at the beginning of the game, and then re-using them during the game, instead of destroying and re-creating objects.

Create a new game object and rename it **GameController**.



With the GameController selected, click **Add Component**. In the search box, type **Game**. Find and select the **Game Controller** script.



25

Because there is only one GameController object in the game and other scripts will want to refer to this GameController, it's a good idea to make this into a **singleton**.

Singletons are special types of scripts that allow only one of them to run at a time. When an object with a singleton script is spawned in, it checks if there is already a version of the script in existence, and if not, then it becomes the acting version.

To create a singleton at the top of the GameController script, create a static variable to store the active GameController instance. Remember that static variables keep the same variable even on multiple scripts.

```
© Unity Script (1 Asset Reference) | 3 References
public class GameController : MonoBehaviour
{
    public static GameController instance;
```

26

After the singleton, add some additional variables to store the platform GameObject and the y position to spawn in platforms at.

```
public static GameController instance;

[Header("Platform Object")]
public GameObject platform;
public float yPos = 0;
```

27

Inside the **void Start** function, check if there is not already an instance. If there isn't, set the instance to be the current script, and call the **SpawnInitialPlatforms** function (which will be created in the next step).

Type this code inside the **void Start** function:

```
void Start()
{
    if (!instance)
    {
        instance = this;
        SpawnInitialPlatforms();
    }
}
```

28

Create a new **void SpawnInitialPlatforms** inside the class below **void Start**, then add this code:

```
private void SpawnInitialPlatforms()
{
    for (int i = 0; i < 100; i++)
    {
        SpawnPlatform();
    }
}
```

29

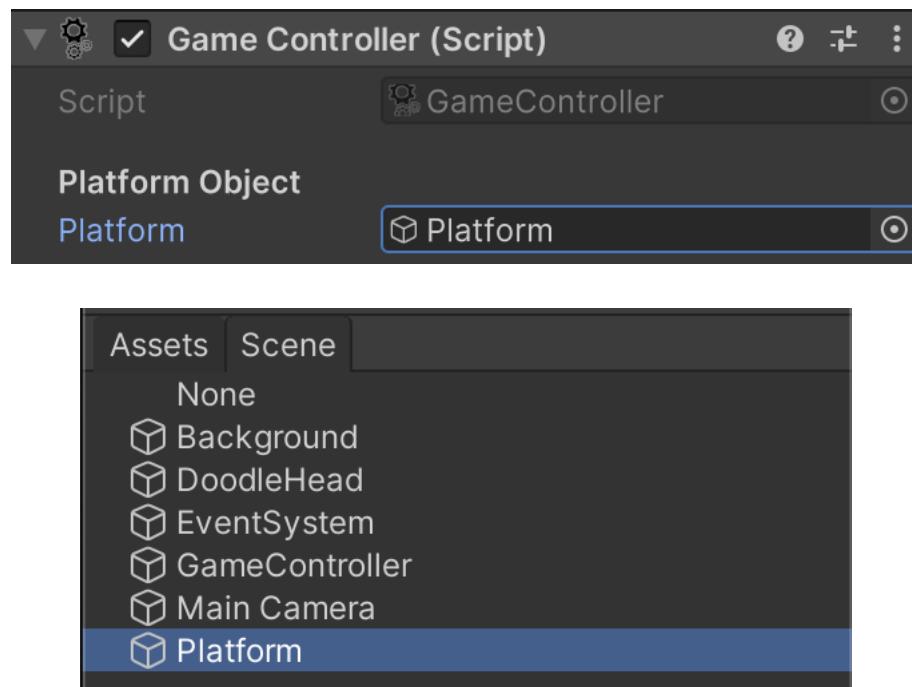
Lastly, create a new **public void SpawnPlatform** inside the class and below **void SpawnInitialPlatforms** and add this code:

```
private void SpawnPlatform()
{
    float xPos = Random.Range(-3f, 3f);
    Instantiate(instance.platform, new Vector3(xPosition, instance.yPos, 0), Quaternion.identity);
    instance.yPos += 2.5f;
}
```

Notice that it is using the **instance** variable's **yPos**! You can now delete the **void Update** function. Save the script and return to Unity!

30

Within the GameController Inspector input field, search and select the **Platform** game object.



31

Now, let's play the scene. There should be platforms spawning for the player to jump on!

32

Stop the game.

33

Now, there are over 100 platforms! However, the player will eventually reach the top. The platforms should move back up to the top of the game after they have moved off screen.

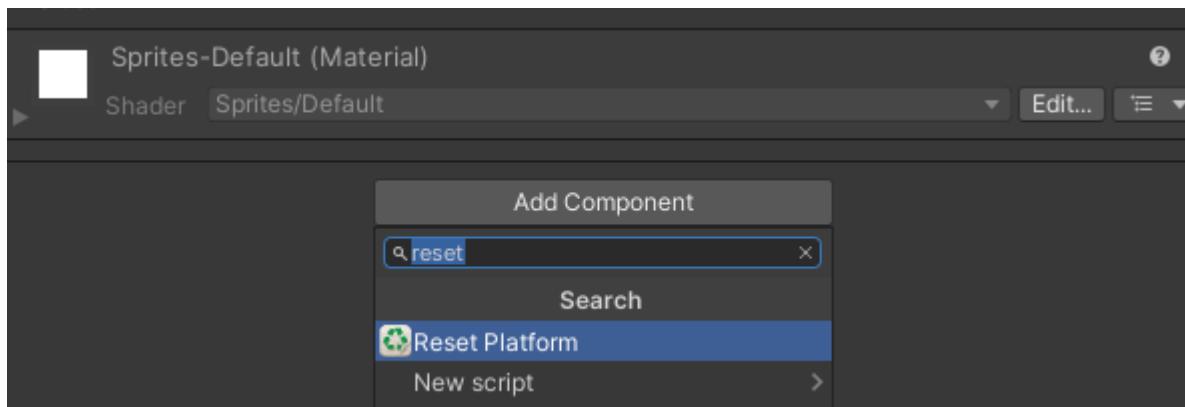
Start by adding a **public static void MovePlatformToTop** function to the GameController script. By making the function static, the function can be called without having to find the GameController object! Add the code in below:

```
public static void MovePlatformToTop(GameObject platform)
{
    float xPos = Random.Range(-3f, 3f);
    platform.transform.position = new Vector3(xPos, instance.yPos, 0);
    instance.yPos += 2.5f;
}
```

Notice again that it is using the instance's yPos!

34

Next, the platforms need to use this method when they are no longer visible. In Unity, select the Platform, then select **Add Component**. In the search bar type **ResetPlatform**. Add the script.



35

Open the script. Delete the Start and Update methods. Add the code below.

```
Unity Script (2 asset references, 1 reference)
public class ResetPlatform : MonoBehaviour
{
    Unity Message | 0 references
    private void OnBecameInvisible()
    {
        GameController.MovePlatformToTop(gameObject);
    }
}
```

This uses a built-in method that runs when the GameObject is no longer on screen. Notice how the **MovePlatformToTop** method can be used now that it's static!

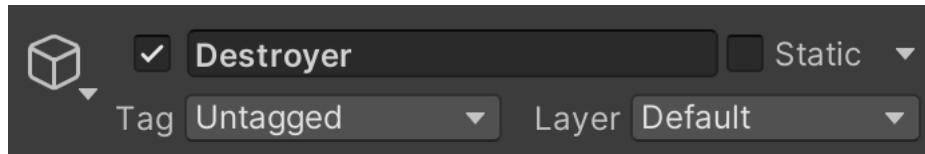
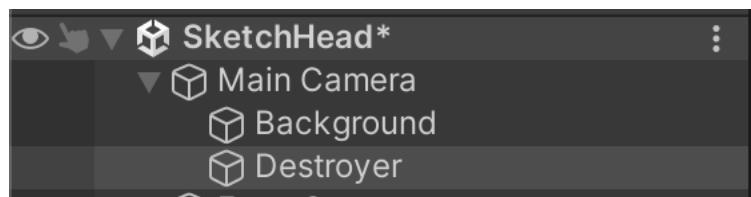
36

Save the script and return to Unity. Notice that the original platform changes y position after it moves off the screen!

37

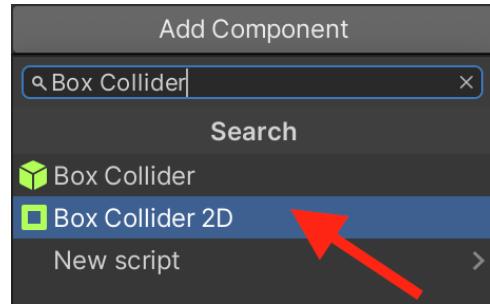
Now that the platforms are taken care of, let's make the player able to lose. To do so, we must add a new GameObject inside the **Main Camera** object. Rename this GameObject to **Destroyer** in the Inspector.

Remember, to move the game object inside the Main Camera, you can drag it up to the Main Camera.

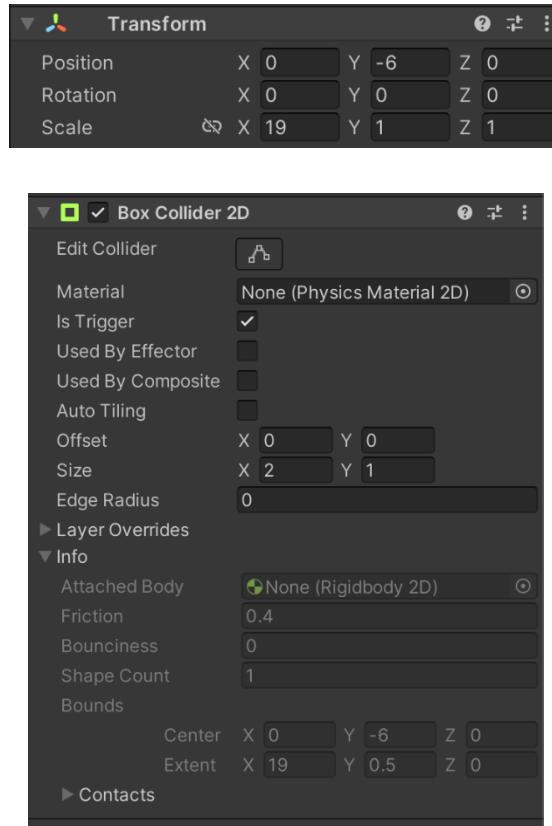


38

With **Destroyer** still selected, click **Add Component**. In the search box, type **Box Collider** and then select **Box Collider 2D**.



Change your GameObject and the Box Collider 2D component to match the information shown in the images below.

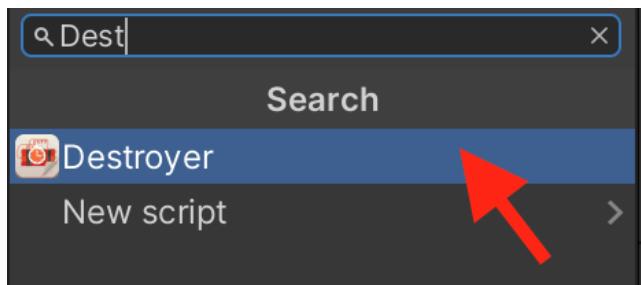


Position values: X: 0, Y: -6, Z: 0; **Scale Values:** X: 19, Y: 1, Z: 1

Ensure that **Is Trigger** is checked in the **Box Collider 2D** component and that you update the **Offset** and **Size** to match the image.

39

With the **Destroyer** GameObject still selected, click **Add Component**. In the search box, type **Destroyer** and select the **Destroyer** component.



40

Open the **Destroyer** Script and delete the Start and Update methods. Create a new **void OnTriggerEnter2D** function inside the class. An error will appear but don't worry, we will resolve that shortly.

```
public class Destroyer : MonoBehaviour
{
    @ Unity Message | 0 references
    public void OnTriggerEnter2D(Collider2D collision)
    {
        GameObject.Find("DoodleHead").SetActive(false);
        GameController.GameOver();
    }
}
```

41

Now that we just added the **void OnTriggerEnter2D** function in the last step, we need to update our code in the Game Controller script to fix the error. Reopen the **Game Controller** script and add the following code inside the class:

```
[Header("Platform Object")]
public GameObject platform;
float yPos = 0;

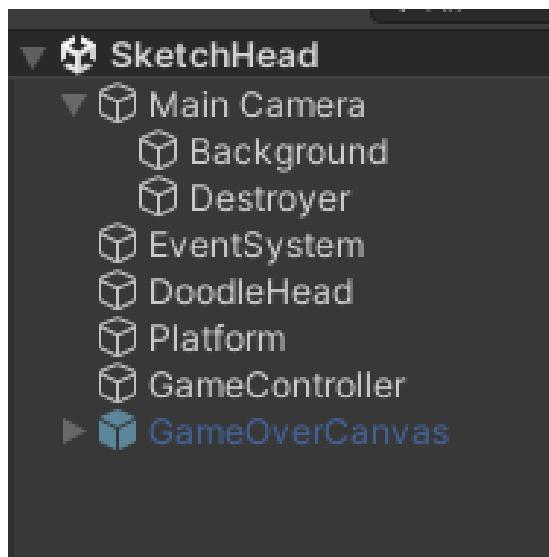
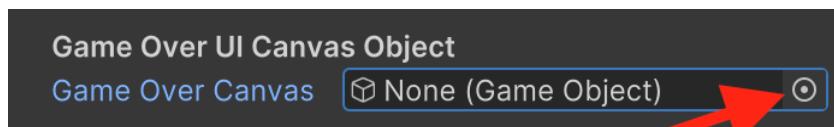
[Header("Game Over UI Canvas Object")]
public GameObject gameOverCanvas;
```

After the **SpawnPlatforms** function, add a **static void GameOver** function and update the code to match below. Again, note that making the method static allows it to be called from the Destroyer script without having to find the component in the game!

```
public static void GameOver()
{
    //Game Over Canvas is set to active
    instance.gameOverCanvas.SetActive(true);
}
```

You can now save the script and return to Unity!

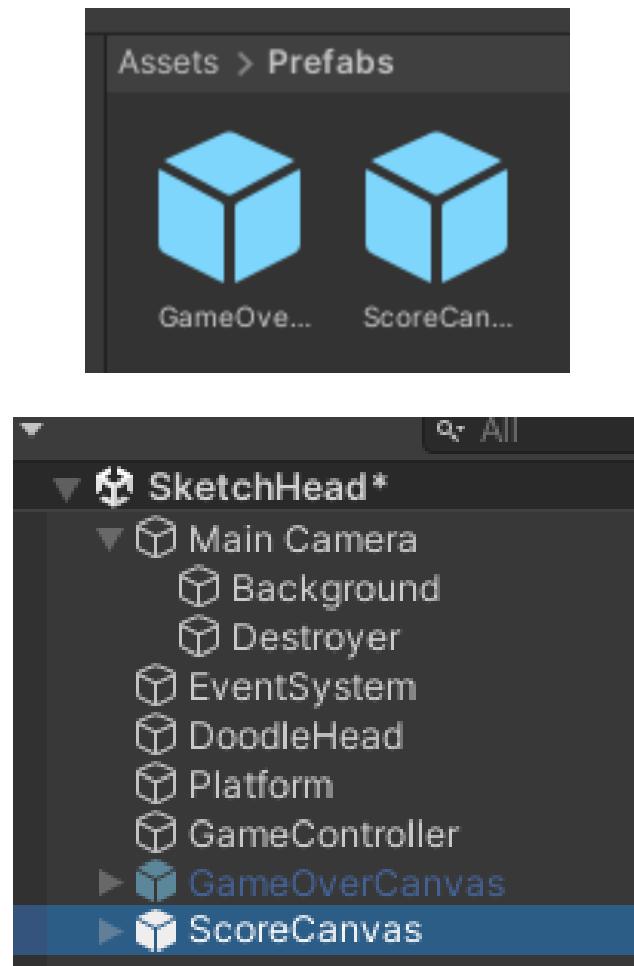
-
- 42** Go to the **Prefabs** section of the **Assets** folder to find the **GameOverCanvas** object, then place it into the **Hierarchy**.
- 43** Select the **GameController** in the Hierarchy, then find the **Game Controller** script component. Find **Game Over UI Canvas Object** then search in the input field for the **GameOverCanvas** game object.



-
- 44** Play the scene. You will notice that if the object collides with the collider, the player will be invisible, and a game over screen appears!
- 45** Stop the game.
-

46

Now for the part you have been waiting for! Let's add scoring to the game! In the Assets folder, go to the Prefabs section then place the ScoreCanvas into the Hierarchy.



47

We need to adjust the code in the Player Controls to update the score. Reopen the **Player Controls** Script that is attached to the DoodleHead and add the following code inside the class:

```
[Header("Rigidbody Settings")]
[Tooltip("Rigidbody2D object that is stored")]
public Rigidbody2D rb;

[Tooltip("Downward speed of the object")]
public float downSpeed = 20f;

[Header("Movement Settings")]
[Tooltip("Movement Speed of the object")]
public float movementSpeed = 10f;

[Tooltip("Movement direction of the object")]
public float movement = 0f;

private SpriteRenderer spriteRenderer;

//Score of game
[Header("Score Text")]
public Text scoreText;
private float topScore = 0.0f;
```

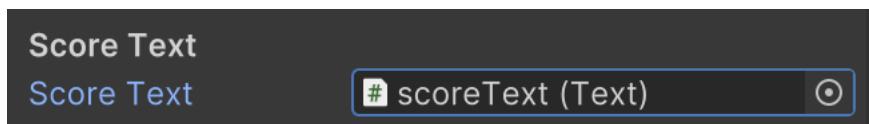
Inside the **void Update** function in the same script, add the following code:

```
//if players velocity is greater than 0
//and position on the y axis is greater
//than the score
if (rb.velocity.y > 0 && transform.position.y > topScore)
{
    //score equals players position
    topScore = transform.position.y;
}
//Text for the score equals to the top score
scoreText.text = "Score: " + Mathf.Round(topScore).ToString();
```

Now save the script and return to Unity!

48

Select the **DoodleHead** in the Hierarchy. Under the Inspector, scroll down to find the **Score Text**. Click on the circle and replace none with **scoreText**.



49

Play the scene. You will notice that the higher the object goes, the more the score increases.

50

Great job! You can now **save** your game, then **export** it. **Submit** your game on the Dojo so a Code Sensei can grade it!

Prove Yourself: TrickHead



Let's add some fake platforms to make playing SketchHead a little...trickier. Open your SketchHead file and make the following changes:

- 1. Duplicate the Platform.**
- 2. Rename the duplicated Platform(1) to FakePlatform.**
- 3. Change the color of the FakePlatform and turn off its collision.**
- 4. Add the FakePlatform to the scene.**
- 5. Open the GameController Script and add this code inside the class before void Start:**
 - a. *public GameObject fakePlatform;*
- 6. Add the following code inside the void SpawnPlatforms:**
 - a. *Instantiate(instance.fakePlatform, new Vector3(xPosition, instance.yPos, 0), Quaternion.identity);*

SCRIPTING

By now you probably know that scripting is an integral part of making games. Unity utilizes the **C-Sharp (C#)** coding language for all its scripts.

By now, you also probably know that while scripts have slightly different ways of using them, they all use variables, conditionals, functions, and loops. **C#** is no different from other coding languages in that aspect.



Bits and Pieces of Code

In Unity, scripts are attached to objects to tell these objects what to do. The behavior can be very specific (put the object at a specific position) or general (keep track of the time). Scripts in Unity are modular, meaning that the same script can be attached to several objects and a single object can have several scripts attached to it. The advantage of this is that a simple script (such as for movement) can be used in a variety of projects and if a project needs something more, then an additional script can be written to support the original one.

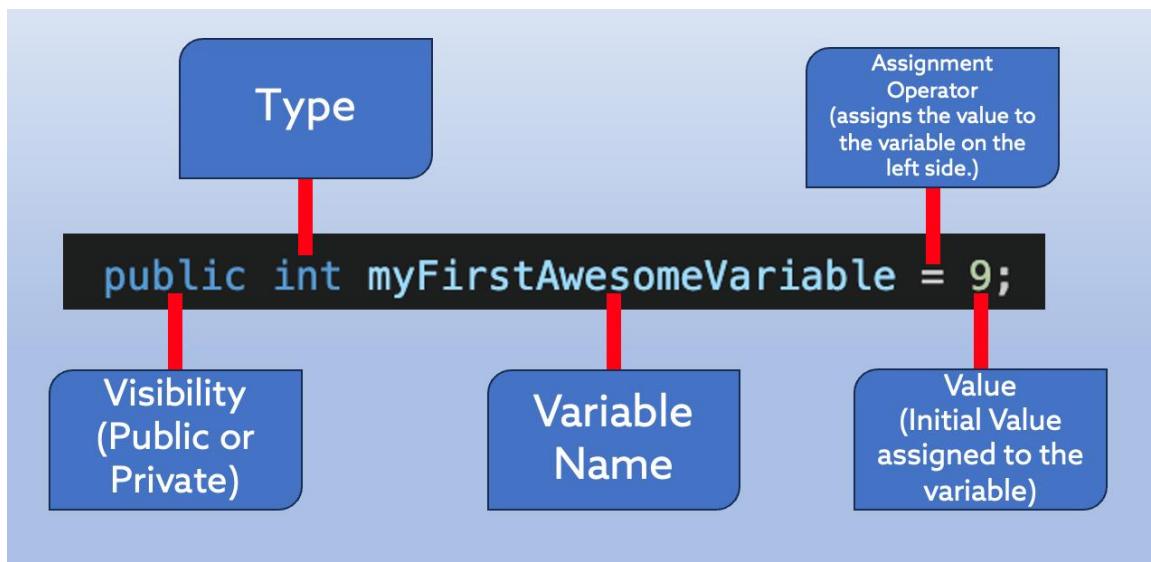
Anatomy of a Script

The main parts of a C# script are variables, functions, and classes. You will also notice that at the top of a script is something called a using directive. These specify elements of C# that the script will be using most frequently and are included by default in every new script you create. There are other directives for other things (such as the user interface) that will be explained later. For now, we will talk about variables and functions.

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class Demo : MonoBehaviour
6  {
7      public int myFirstAwesomeVariable;
8      private bool mySecondCoolVariable;
9
10     // Start is called before the first frame update
11     void Start()
12     {
13
14     }
15
16     //Update is called once per frame
17     void Update()
18     {
19
20     }
21 }
```

Variables

Variables in C# have 3 main parts: **visibility** (usually public or private), **type**, and a **name**. When declaring a variable, you'll decide if the variable needs to be seen outside of the script. If you are declaring a speed variable and you want to adjust it without having to edit the script, you would make it public. Else, make it private. The type is what kind of variable. It could be an int (integer), float (floating), bool (boolean) or even a GameObject or another script. The final part is the name of the variable and, optionally, the value of that variable.



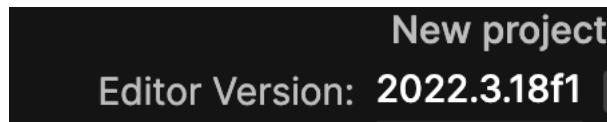
Functions

Functions are where things happen in a script. When a new script is created, two functions are automatically created, **Start()** and **Update()**. Start is called when the game starts. Update is being called constantly. All other functions must be called by name before they do anything. Like variables, functions can be public or private, depending on whether or not you need other scripts to call them. Most of the time, a script starts with a **void** type, meaning that the function is just running and not returning any data.

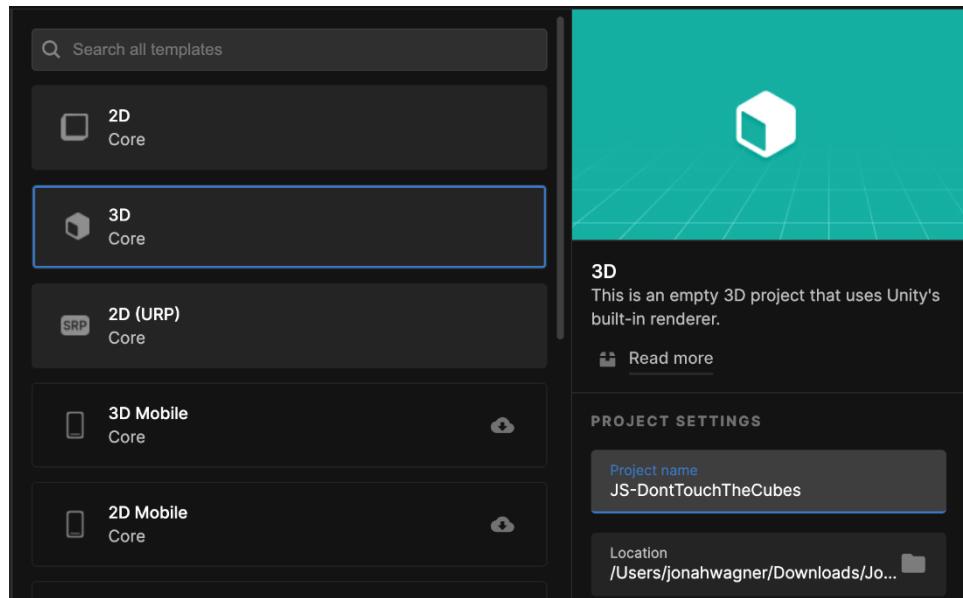
Activity 5: Don't Touch the Cubes

In this activity, you will design a simple game where gravity, colliders, and physics come into play.

- 1 Open the Unity Hub application on your computer. Select the **New Project** button in the upper right-hand corner. Make sure the **2022.3 LTS** version is being used.

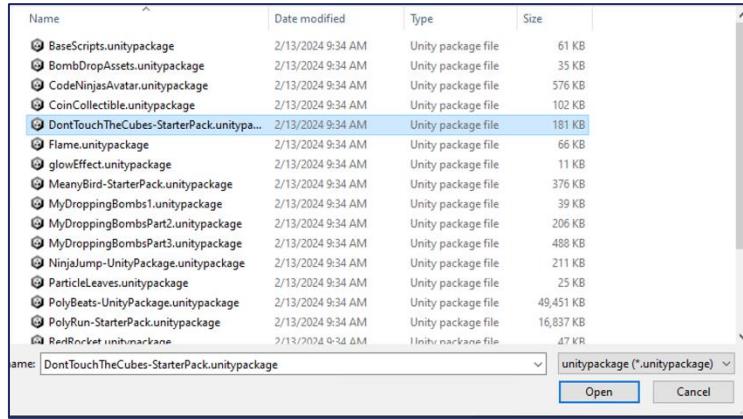


- 2 Select **3D** in the Templates column. Next, type in your first and last initials plus the name of the project. For example, John Smith would save their project as JS-DontTouchTheCubes. Select the folder location where you normally save your Unity games. Then select the **Create** button!

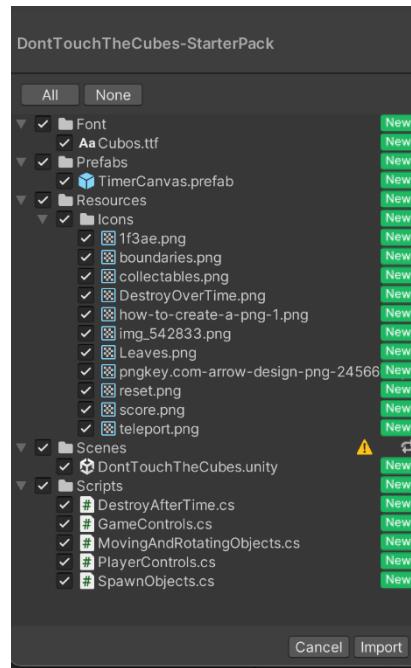


- 3 Once Unity loads all the necessary assets into the program you will see that a blank project exists. We're about to fix that.

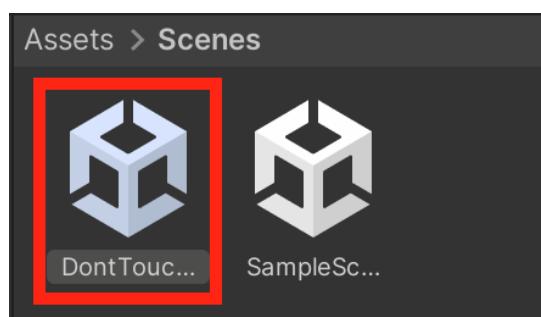
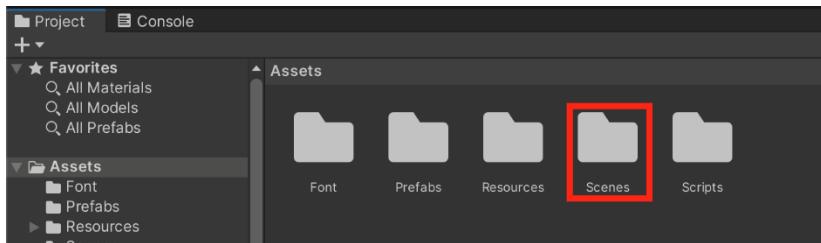
4 Go to the **Assets** tab at the top, select **Import Package**, and then click on **Custom Package**. By selecting **Custom Package**, we can import compressed packages into Unity. Find your Unity assets folder. Then select the **Activity 05 - DontTouchTheCubes-StarterPack.unitypackage** file. Once you select the file Unity will then show a menu of what is inside the package.



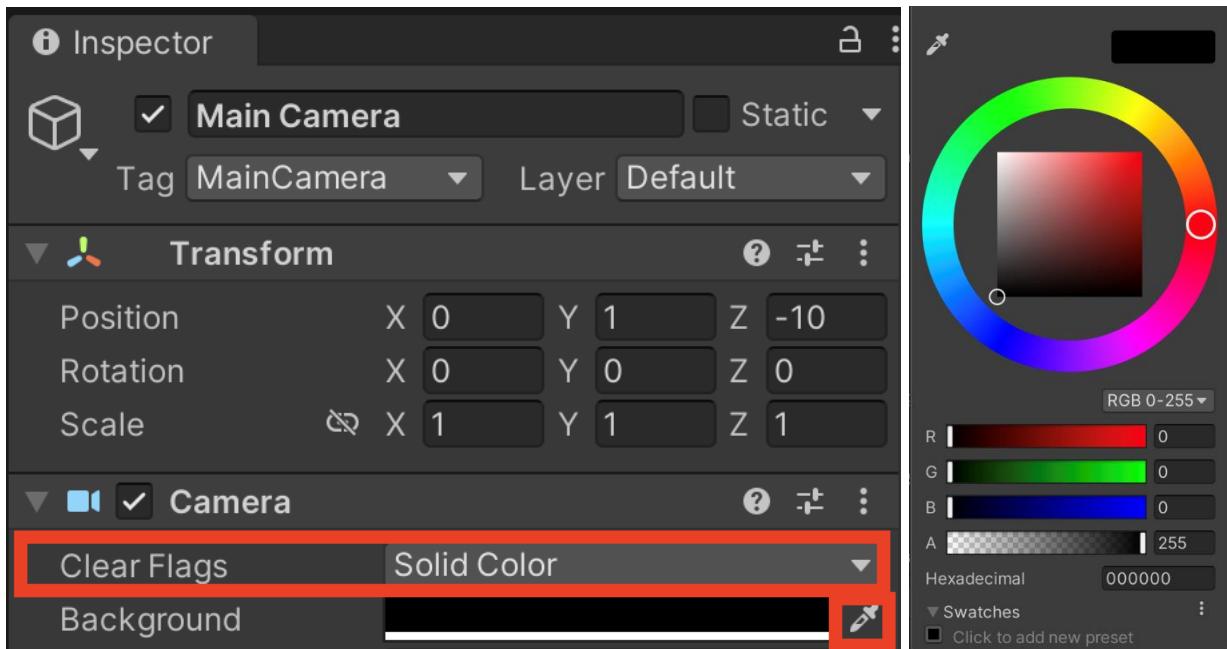
5 Since this is file is built just for this game, we want to make sure all the material inside the package is selected. If all the material is not selected, click the **All** button to select everything. Click **Import** and all the material will be imported into the project.



6 Now click **Scenes** under the **Assets** window and open the *DontTouchTheCubes* scene. Even though you've imported the scene, there is nothing in the Scene window or the Game window. However, we do have the materials that we need in our folder to start creating the game.



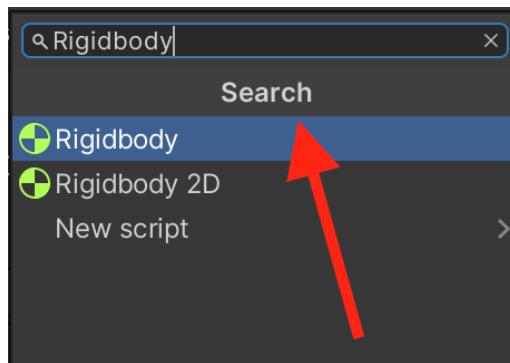
7 Select the **Main Camera** game object. Then, find the **Clear Flags** dropdown menu, and choose **Solid Color**. Change the background color to black.



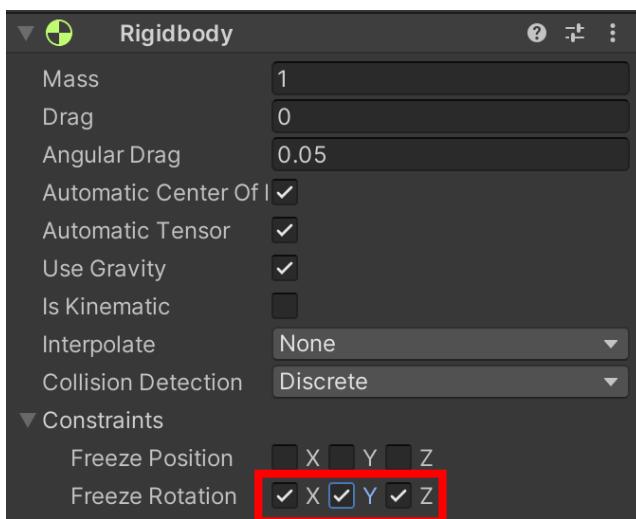
- 8** With the **Main Camera** game object still selected, change the rotation to **90** on the X axis.



- 9** Select the **Main Camera** game object again and click **Add Component**. In the Search Box, type **Rigidbody** and then select **Rigidbody**.

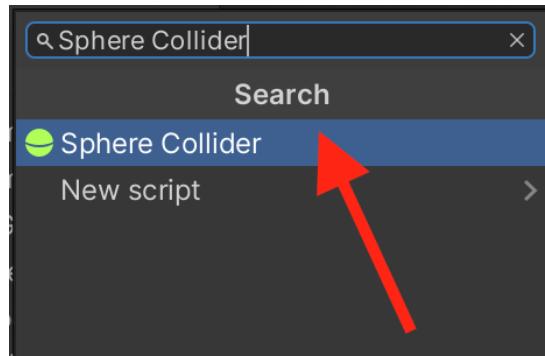


In the **Rigidbody** component, click the arrow for the **Constraints** drop-down section. Click on all boxes next to **Freeze Rotation** for all 3 axes. Then set the **Drag** to **1**.



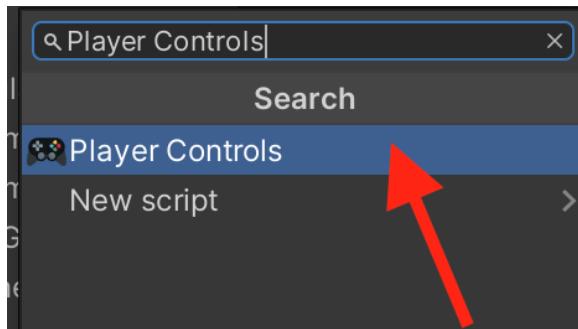
10

Make sure you continue to have the **Main Camera** game object selected. Then select **Add Component**. In the search box, type **Sphere Collider**, then select **Sphere Collider**.



11

Make sure the **Main Camera** is still selected, then select **Add Component**. In the search box, type **Player Controls** and select the **Player Controls** script.



12

Open the **Player Controls** script, added to **Main Camera**, then add the following code before the **void Start** function inside the class:

```
public class PlayerControls : MonoBehaviour
{
    [Header("Rigidbody")]
    //Rigidbody component
    public Rigidbody rb;
    // Start is called before the first frame update
    void Start()
```

13

Next, within the **void Start** function, insert the code below:

```
void Start()
{
    //variable rb equals to
    //component rigidbody
    rb = GetComponent<Rigidbody>();
    //Game is at normal pace
    Time.timeScale = 1f;
}
```

14

Within the **void Update** function, insert the code below:

```
void Update()
{
    // Rotating the game object on the z-axis
    // multiplied by the game frame rate by 7
    transform.rotation *= Quaternion.Euler(0, 0, 7 * Time.deltaTime);

    // Time scale of the game plus
    // physics and other fixed frame rate updates
    Time.timeScale += Time.fixedDeltaTime * 0.01f;

    // Movement and rotation of the camera
    // on the y-axis horizontally
    rb.velocity += transform.rotation * (Vector3.right * Input.GetAxisRaw("Horizontal") * 10f * Time.deltaTime);

    // Movement and rotation of the camera
    // on the y-axis vertically
    rb.velocity += transform.rotation * (Vector3.up * Input.GetAxisRaw("Vertical") * 10f * Time.deltaTime);

    // Keeping the camera position in bounds
    // Refactor the clamping into separate variables for clarity
    float clampedX = Mathf.Clamp(transform.position.x, -30f, 30f);
    float clampedZ = Mathf.Clamp(transform.position.z, -30f, 30f);

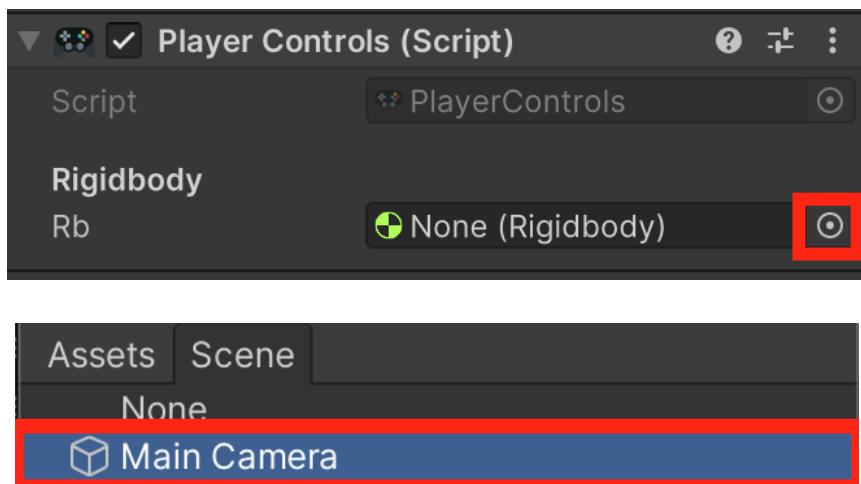
    // Apply the clamped values to the transform's position
    transform.position = new Vector3(clampedX, 0, clampedZ);
}
```

You can now save your script and return to Unity!

Clamping is a function in Unity that restricts a value within a specified range. By clamping an object's position, it helps make sure that it cannot move outside the set minimum and maximum limits.

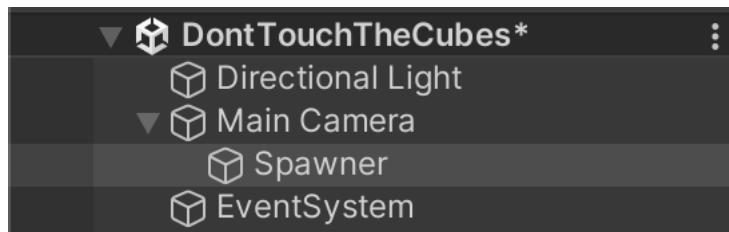
15

With the **Main Camera** selected, locate the **Player Controls** script component inside the Inspector. Find the **Rigidbody** input field, then select the properties circle button. Select the **Main Camera** in the Scene panel.



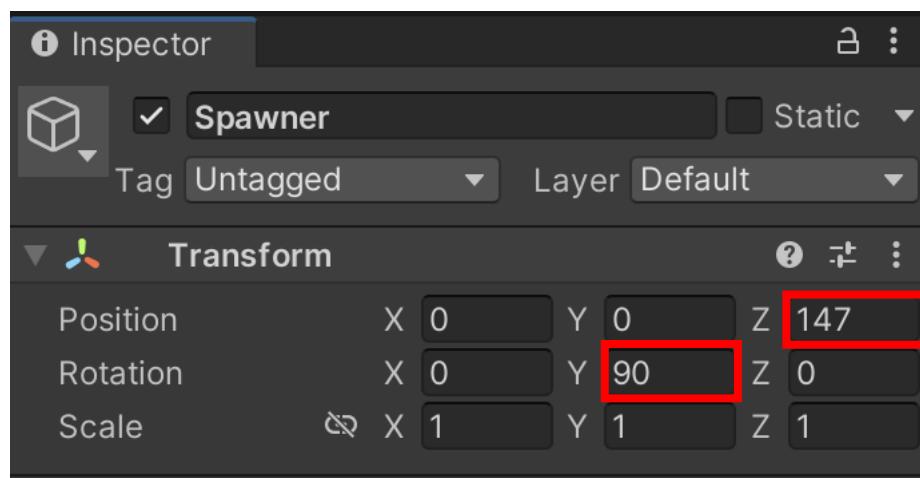
16

Now you will create a **Spawner** object, which is an object that is made for creating multiple objects from its location. First, we need to create one game object. Right click on **Main Camera** then select **Create Empty**. Rename the object to **Spawner** in the inspector.



17

Change the position and rotation of the Spawner to match the values shown below.

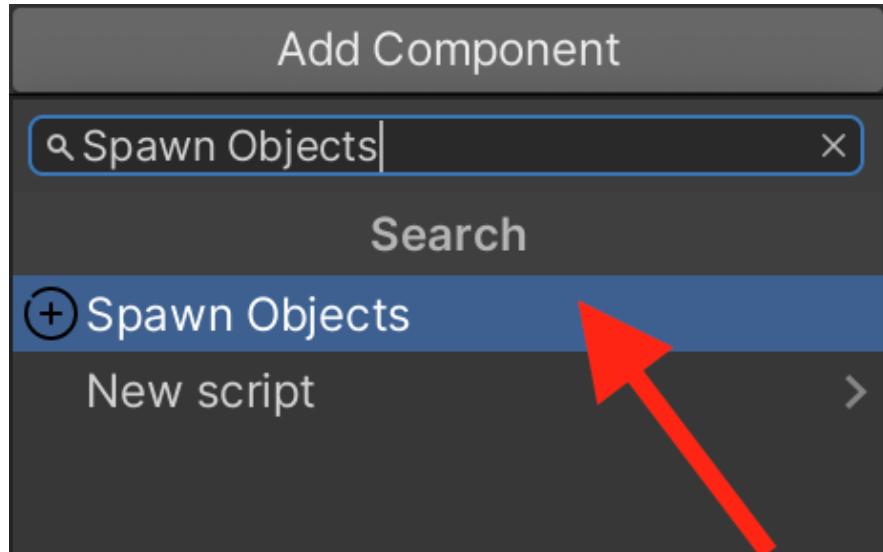


Position values: X: 0, Y: 0, Z: 147

Rotation values: X: 0, Y: 90, Z: 0

18

With the **Spawner** still selected, click **Add Component** in the Inspector. In the search box, type **Spawn Objects**. Then, select the **Spawn Objects** script.



Open the **Spawn objects** script and add the following code inside the class before the **void Start** function:

```
public class SpawnObjects : MonoBehaviour
{
    //Cube that is going to be spawned
    [Header ("Spawn Cube Object")]
    public GameObject spawnCube;
    //Difficulty of the game
    [Header ("Default Difficulty")]
    public float difficulty = 40f;
    //Time for the next cube to be spawned
    float spawn;

    // Start is called before the first frame update
    void Start()
    {

    }
```

19

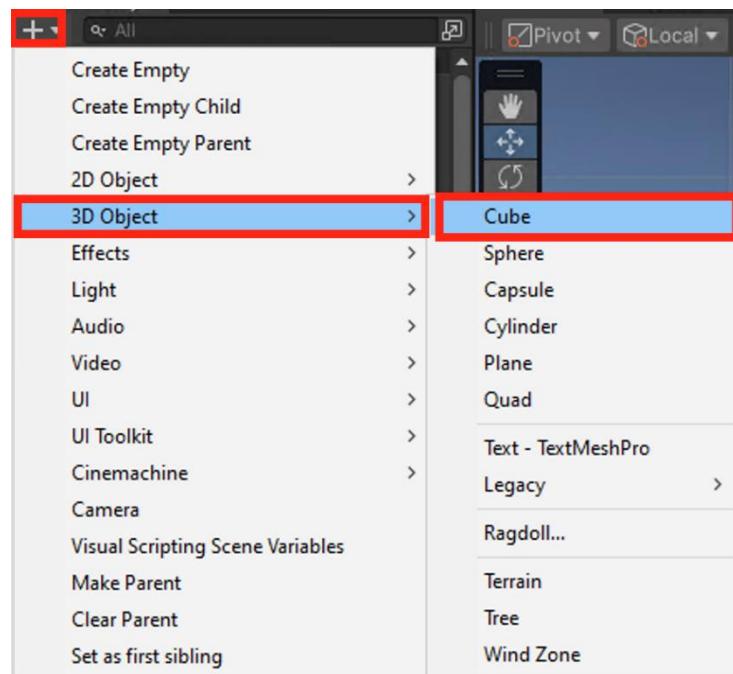
Delete the **void Start** function; we don't need it here. Inside the **void Update** function, add the following code:

```
// Update is called once per frame
void Update()
{
    spawn += difficulty * Time.deltaTime;
    while(spawn > 0)
    {
        spawn -= 1;
        Vector3 v3Pos = transform.position + new Vector3(Random.value * 40f - 20f, 0, Random.value * 40f - 20f);
        Quaternion qRotation = Quaternion.Euler(0, Random.value * 360f, Random.value * 30f);
        GameObject createObject = Instantiate(spawnCube, v3Pos, qRotation);
    }
}
```

You can now save the script and return to Unity!

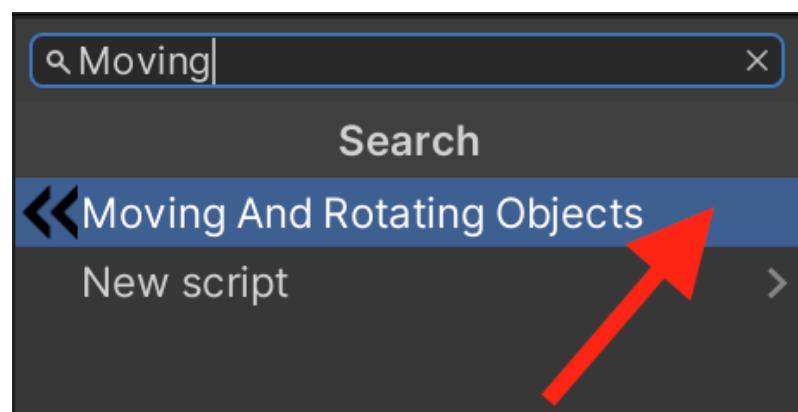
20

We now need to create cubes for the game! To do this, select the **+** button in the top left of the hierarchy. Select **3D Object** from the drop-down menu, then click **Cube**.



21

Select the **Cube** in the hierarchy, then in the Inspector, click **Add Component**. In the search box, type **Moving** and then select the **Moving and Rotating Objects** script.



22

Open the **Moving and Rotating Objects** script and add the following code inside the class and before the **void Start** function:

```
//Default moving speed  
[Header ("Default Movement Speed")]  
public float moveSpeed = 10f;  
//Default Rotating Speed  
[Header ("Default Rotation Speed")]  
public float rotateSpeed = 50f;
```

23

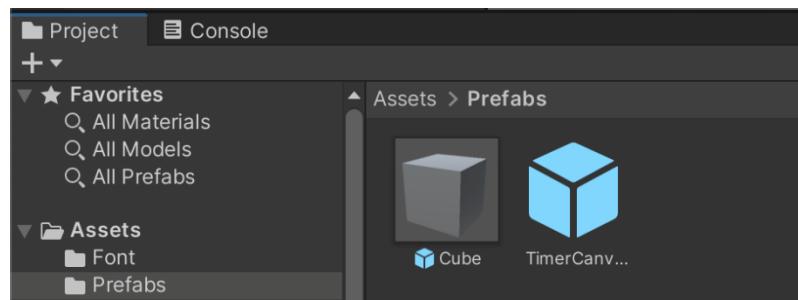
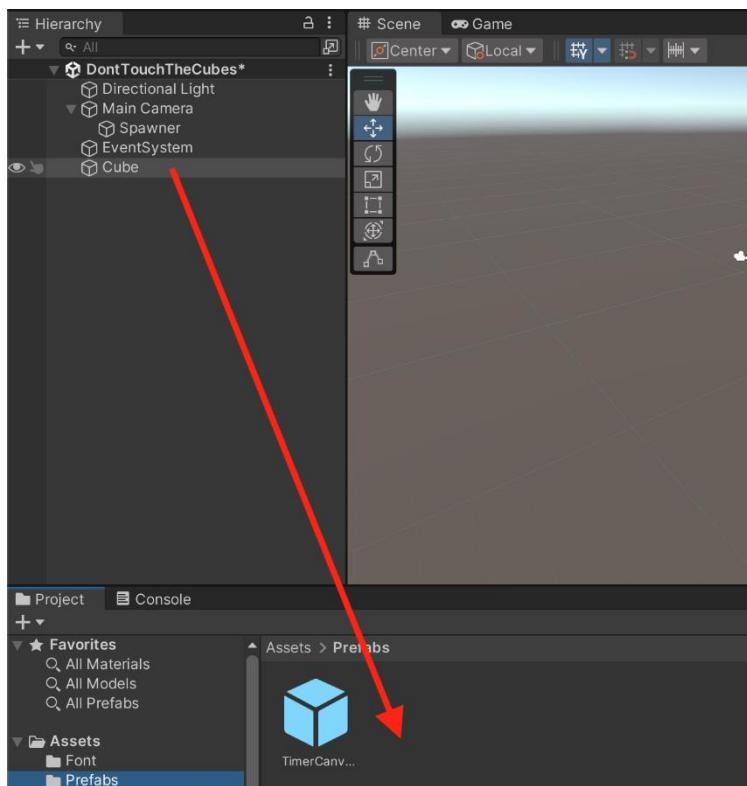
Delete the **void Start** function; we don't need it here. Inside the **void Update** function, add this code:

```
// Update is called once per frame  
void Update()  
{  
    //object moving on the y axis based on move speed  
    transform.Translate(0, moveSpeed * Time.deltaTime, 0);  
    //rotate on the x axis based on rotate speed  
    transform.Rotate(Vector3.up * rotateSpeed * Time.deltaTime);  
}
```

You can now save the script and return to Unity!

24

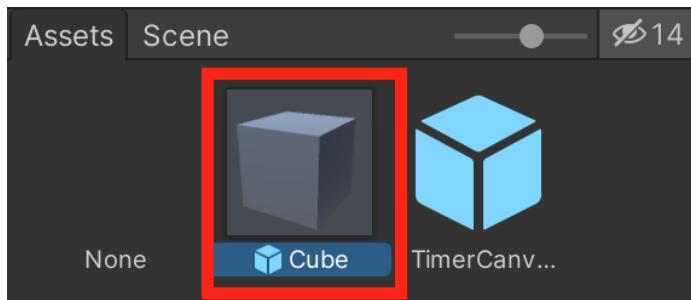
In the Hierarchy keep the **Cube** selected, then select the **Prefabs** folder in the Project menu and drag the **Cube** into the folder.



You can delete the **Cube** from the Hierarchy as you won't need it to show up in the scene anymore!

25

Select the Spawner. In the Inspector, find the **Spawn Objects** script component. Add cubes to the **Spawn Cube** input field by clicking on the Properties button and selecting the **Cube** prefab in the **Assets** window.



26

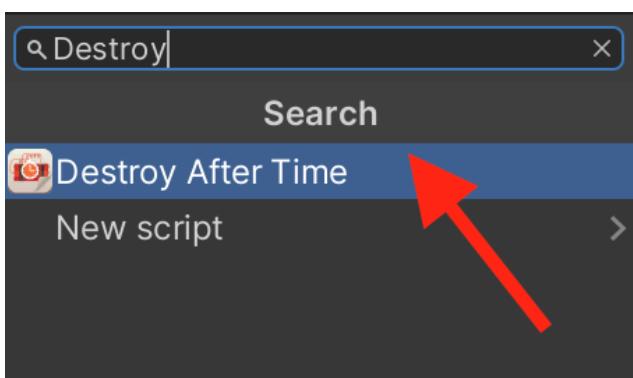
Play the game! You should see that the cubes will begin to spawn and move toward the screen.

27

Stop the game.

28

We are moving downwards but the cubes are still active off the screen. To fix this we need to destroy these cubes by using a **Destroyer** script. Select the **Cube** in the **Prefab** folder then click **Add Component**. Type **Destroy** in the search box and select the **Destroy After Time** script.



29

Open the **Destroy After Time** script and add the following code inside the class and before the **void Start** function:

```
public class DestroyAfterTime : MonoBehaviour
{
    [Header("Destruction Timer")]
    // After this time, the object will be destroyed
    public float timeToDestruction;
    // Start is called before the first frame update
    void Start()
    {
    }
```

30

Inside the **void Start** function, add the following code:

```
// Start is called before the first frame update
void Start()
{
    //Execute function based on timeToDestruction
    Invoke("DestroyObject", timeToDestruction);
}
```

Create a new **void DestroyObject** function by adding this code:

```
// This function will destroy this object
void DestroyObject()
{
    //Destroy Gameobject
    Destroy(gameObject);
}
```

You can delete the **void Update** function as we don't need it here. Now save the script and return to Unity!

31

While you still have the Cube selected, in the inspector find the **Destroy After Time** script component. Change the **Time to Destruction** to 12 seconds.



32

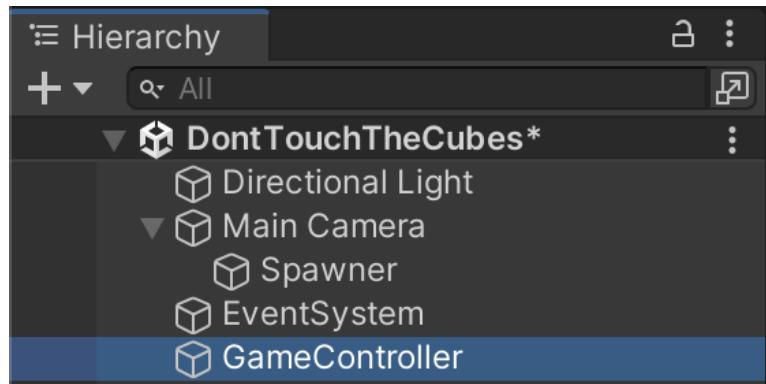
Play the game but this time, don't maximize it. You should see the cubes being destroyed by looking at the hierarchy.

33

Stop the game.

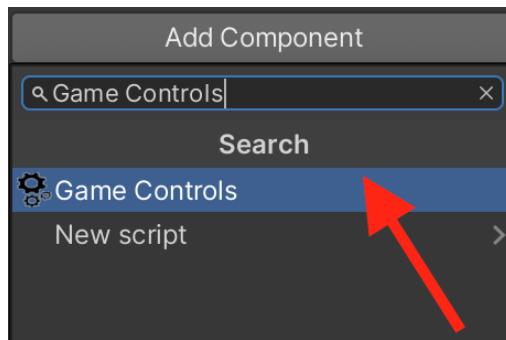
34

In the hierarchy, add a new empty **GameObject** to the game. Rename it **GameController**.



35

With the **GameController** GameObject selected, click **Add Component**. In the search box, type **Game Controls**, then add the **Game Controls** script.



36

Open the **Game Controls** script and add the following code inside the class and before the **void Start** function:

```
//Timer text object  
private Text timerText;  
//Timer counter for adding score  
private int timerCount;
```

37

Inside the **void Start** function, add the following code:

```
void Start()  
{  
    //Game is at a playing state  
    Time.timeScale = 1f;  
}
```

Before we save the script, you can delete the **void Update** function in this script as we do not need it here. You can now save the script and return to Unity!

38

We are almost ready to test out the game! First, we need to add a new function to our player controls in order to have the game reset if we touch a collider. Open the **Player Controls** script in the **Main Camera** object.

39

Create a new **void OnCollisionEnter** function, then add the following:

```
void OnCollisionEnter(Collision collision)
{
    SceneManager.LoadScene(0);
}
```

You can now save the script and return to Unity!

40

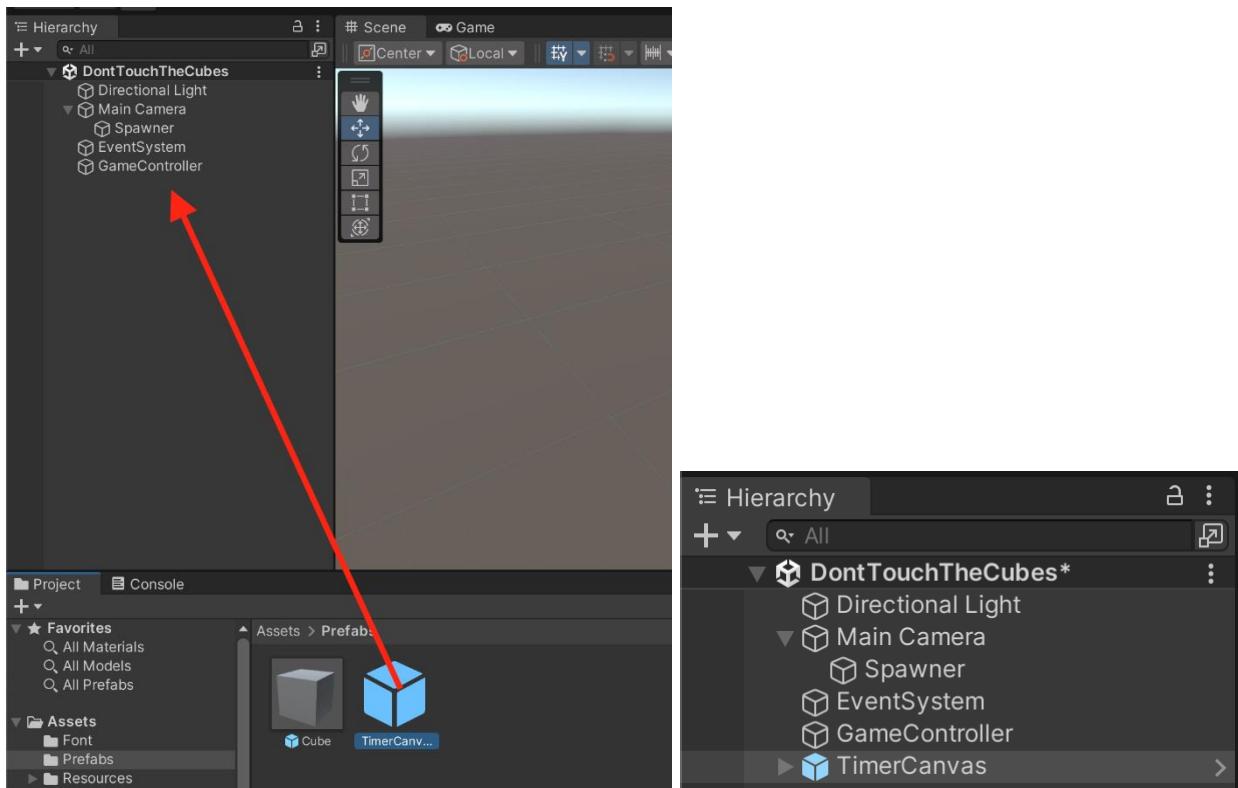
Play the game! Notice that if you touch the cubes, the game will reset.

41

Stop the game.

42

Let's make the game even more fun by adding scoring to the game! Go to the **Prefabs** folder and drag **TimerCanvas** to the **Hierarchy**.



43

In the hierarchy, select the **Game Controller**. In the inspector, open the **Game Controls** script to add scoring.

44

Inside the **void Start** function, add the following code:

```
void Start()
{
    //Game is at a playing state
    Time.timeScale = 1f;
    //Executing a coroutine
    StartCoroutine(CountTime());
    //Timer text equals finding
    //the score object and using
    //the text component
    timerText = GameObject.Find("Score").GetComponent<Text>();
}
```

Under the **void Start** function, create a new **IEnumerator CountTime** function inside the class and add this code:

```
IEnumerator CountTime()
{
    //After 1 second
    //1 point is added to the score
    //and will repeat the function
    yield return new WaitForSeconds(1f);
    timerCount++;
    timerText.text = "Score: " + timerCount;
    StartCoroutine(CountTime());
}
```

You can now save the script and return to Unity!

45

Play your completed game! How high of a score are you able to get?

46

Stop the game. **Save**, **Export**, and **Submit** your game.

Prove Yourself: Don't Touch the Chopsticks



A cube is a 3-dimensional shape with 6 sides made up of squares.

A square is a 2-dimensional shape that has 4 sides of equal length.

The shapes you can see in the above image are called rectangular prisms. 2 of the sides are still squares, but the length of the shape has been changed so that 4 of the sides are rectangles.

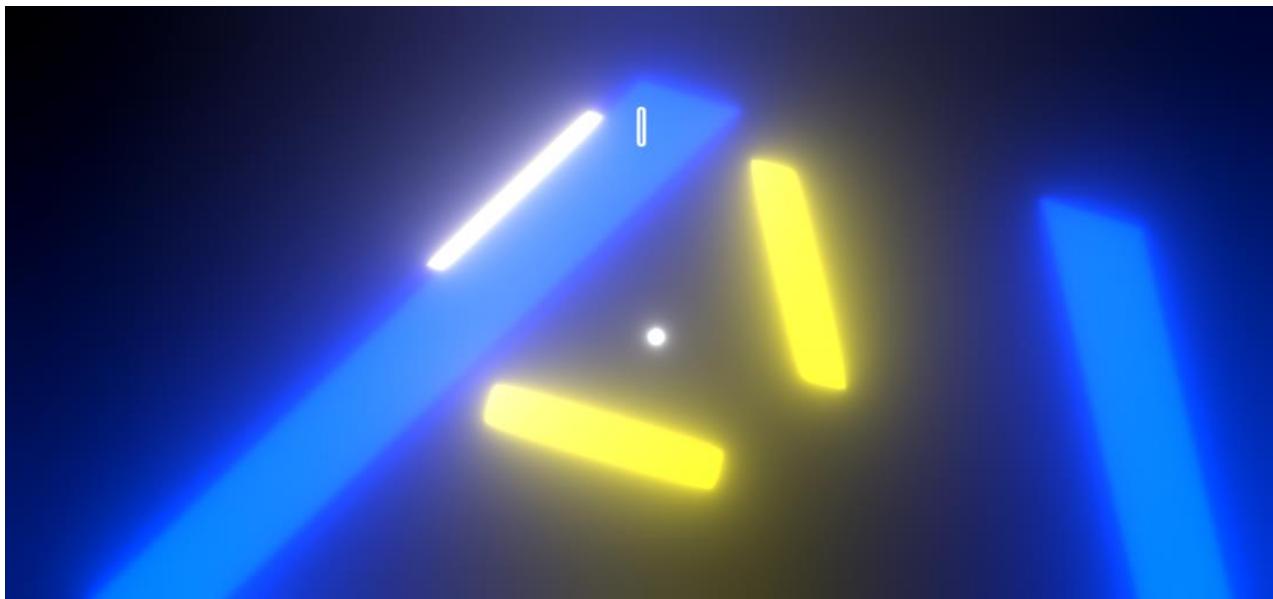
To test your understanding of how to use the Unity interface and 3D shapes, go into your complete Don't Touch the Cubes game, and do the following:

- 1. Resize the cubes to become rectangular prisms as in the image above.**
- 2. Adjust the rotation and movement speed of the obstacles.**
- 3. Try adjusting the Time to Destruction variable if the objects in your game disappear before you can reach them.**
- 4. Adjust the difficulty variable by making it lower to make the game easier and higher for a more difficult game.**

Prefabs

When you're making a game, the last thing that you want to do is manually make the dozens (or even hundreds) of objects that the player must deal with one at a time. It is much easier just to set it up one time and then let the computer take care of the rest.

The Prefab in Unity is just that. Any combination of **GameObjects** and components can be set aside for when you need them again. A prefab could be a combination of models to quickly build a scene, or it can be an assortment of characters that the player must deal with. The bonus of using a prefab is that any changes that you make to the original get applied to all copies.

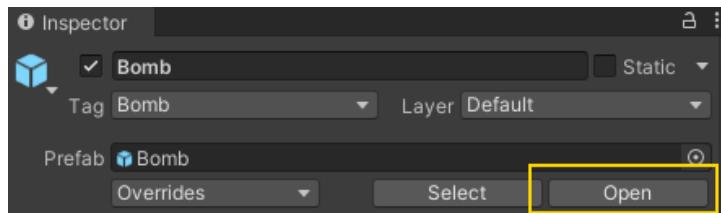


Making a Big Deal out of Many, Little Deals

Any time that you need to reuse a **GameObject** in your scene, you should make it into a **Prefab**. Making a prefab is as simple as dragging a **GameObject** from the Hierarchy into your **Prefs** folder. Both the original object and the Prefab are now marked with blue cubes to indicate that the object in the scene is an instance of the prefab from your folder.

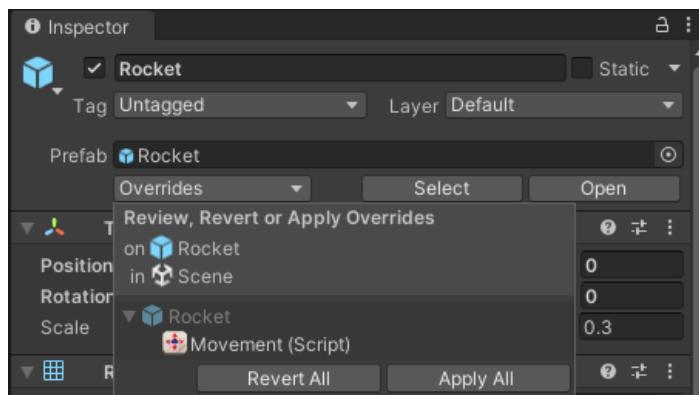
Opening a Prefab

To edit an asset in the **Prefs** folder, select it and click **Open Prefab**. This opens Prefab Mode where you can make changes to the original prefab that then get applied to all instances of that prefab in your scene.



Overrides

Any changes that you make to a prefab in your scene only affect that instance of the prefab. If, for some reason, you want those changes to be saved as part of the original prefab (and applied to all the prefabs in a scene), you must click on **Override** in the **Inspector** panel and select **Apply All**. Any unapplied overrides in a prefab show up in bold text in the **Inspector** panel.



Instantiating

Prefabs can be taken out of the Prefabs folder and added to your game through code at run time. To do so, you will need to define that variable in your code (usually through a public GameObject variable) and use the Instantiate command to create new instances. Instantiate requires three things: the variable for the prefab, the position where to put it in the scene, and the desired rotation.

```
Instantiate(bombPrefab, new Vector3(randomX, spawnY, 0), bombPrefab.transform.rotation);
```

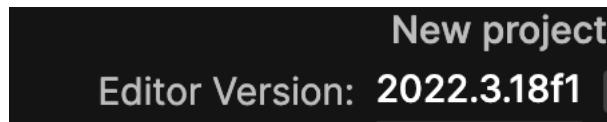
Prefabs and Variables

When instantiating a prefab from the **Prefabs** folder, care must be taken with what is attached to it as a public variable. Prefab scripts have no problem with public variables that are self-contained, such as a number for a speed setting or a boolean value. However, if you have a variable that requires linking to another GameObject in the scene, the prefab will forget that information when the Prefab is instantiated (the exception is if that object is another prefab in the same folder). In those cases, it's best to use a private variable and define that variable using `GameObject.Find("ObjectName")` in the Start function of the script.

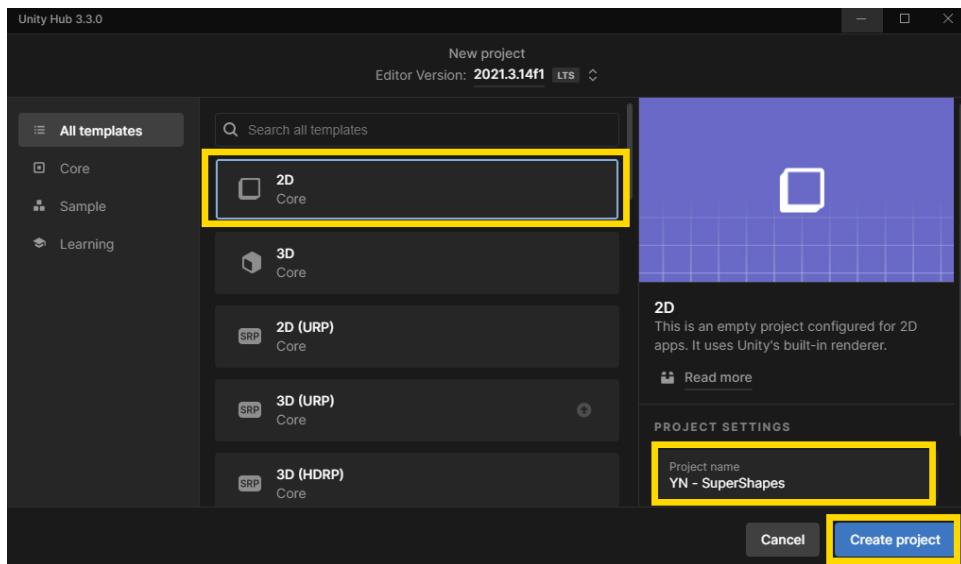
Activity 6: SuperShapes

In this lesson, you will create a simple game where gravity, colliders, and physics come into play.

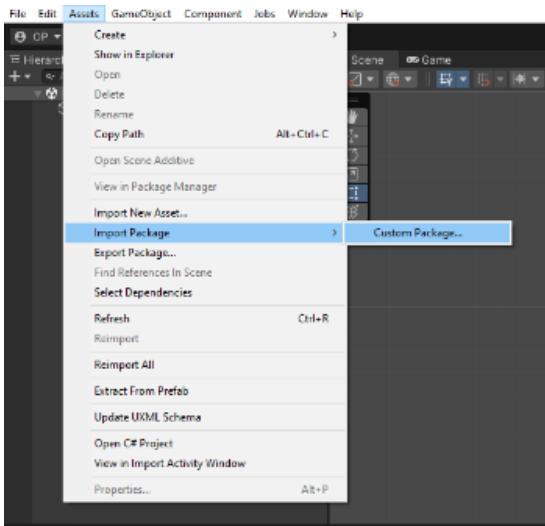
- 1 Open the Unity Hub application on your computer. Select the **New Project** button in the upper right-hand corner. Make sure the **2022.3 LTS** version is being used.



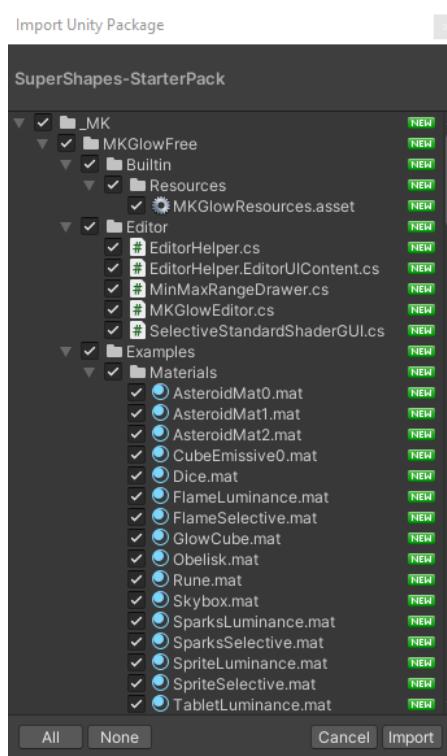
- 2 Select **2D** in the Templates column. Next, type in your first and last initials with the name of the project. For example, John Smith would save their project as JS-SuperShapes. Select the folder location where you want to save your project. Click the **Create** button.



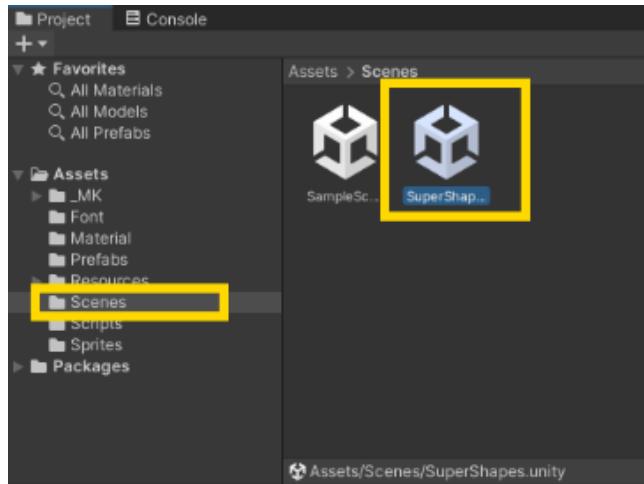
3 Once Unity has opened, just like with previous tasks, we'll want to install a package. Go to **Assets** and select **Import Package** and click on **Custom Package...**.



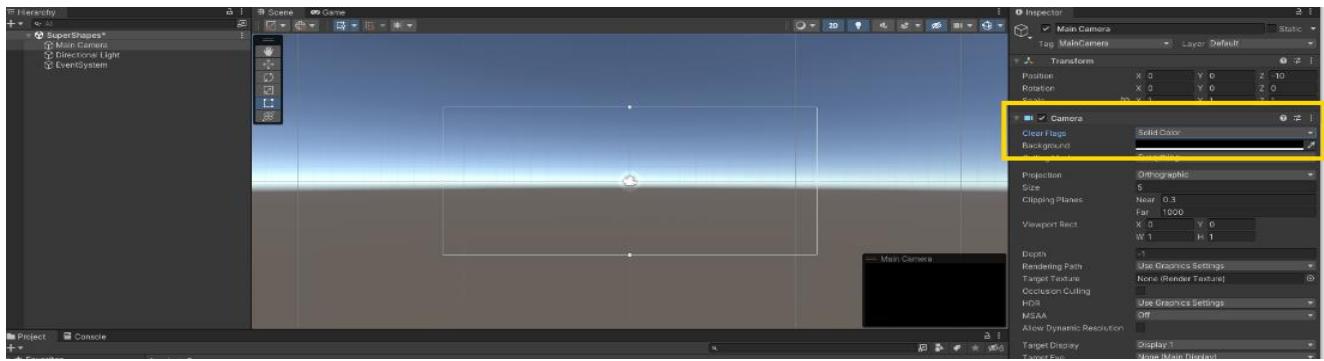
4 Select the **Activity 06 - SuperShapes-StarterPack.unitypackage**. All of the materials inside the package should be selected, if they are not, select **All** and click the **Import** button.



5 With the package installed, we want to load the *Supershapes* scene. It's a blank scene at the moment, but we will add to it. Go to the **Scenes** folder and double-click **SuperShapes**.

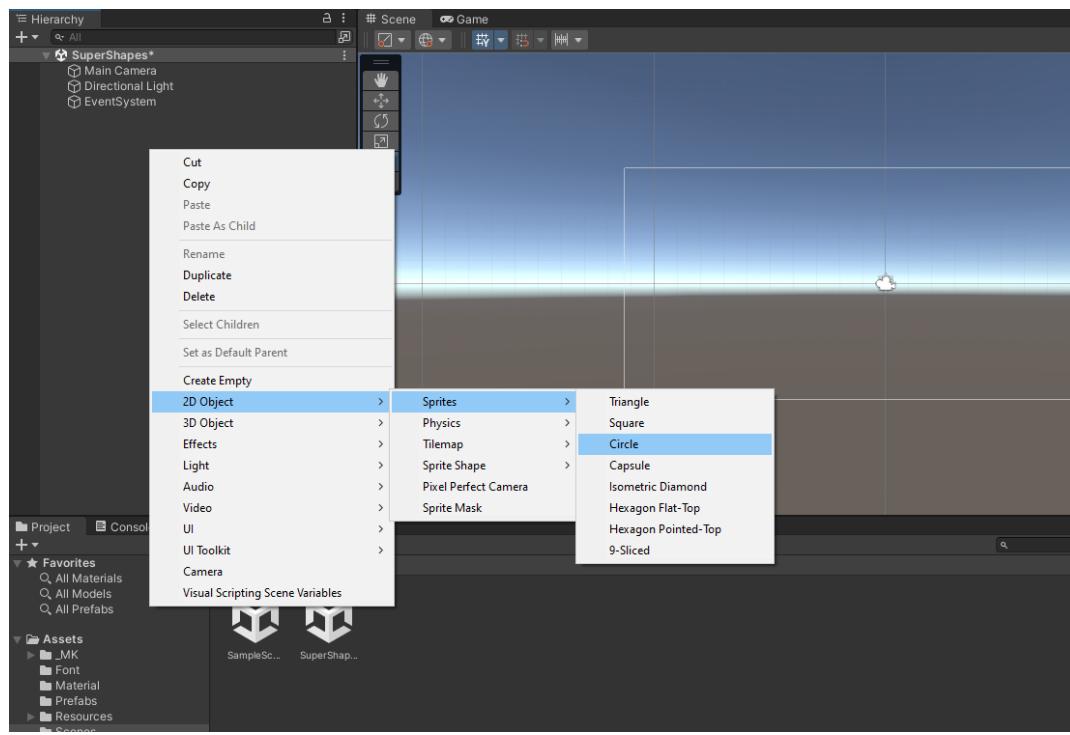


6 First, let's change the background color. Select the **Main Camera** gameobject and change the **Clear Flags** at the top from **Skybox** to **Solid Color**. Next, change the **Solid Color** to black.



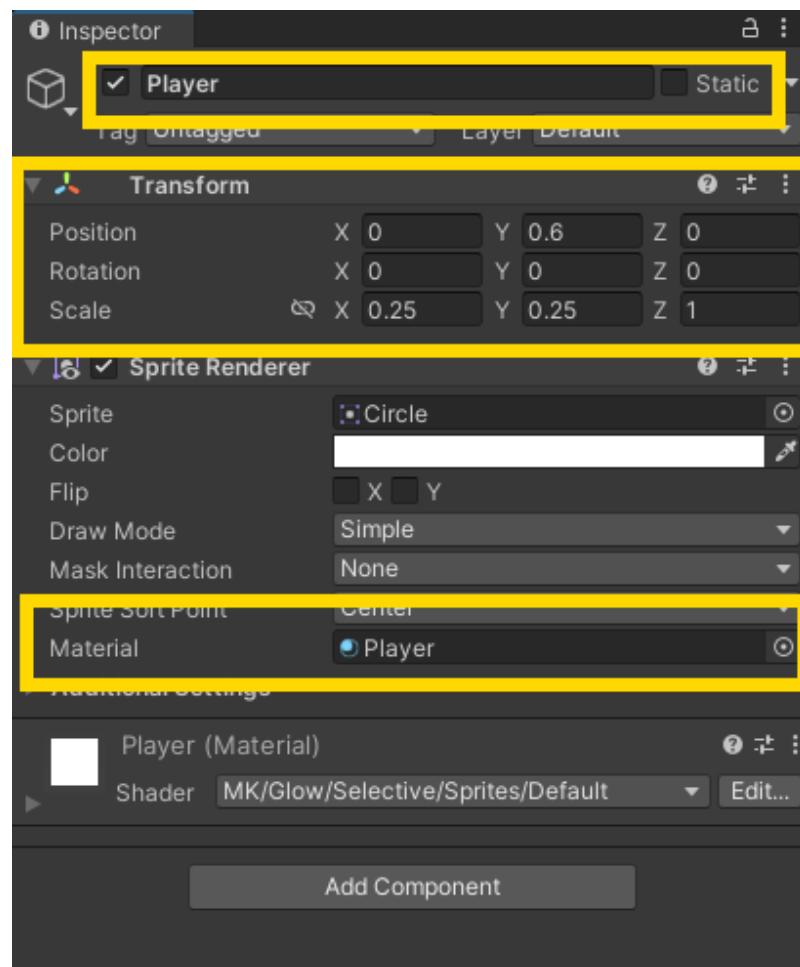
7

We are going to add a player. Instead of creating an empty game object, we are going to go to **2D** to **Sprites** and add a **Circle**, make sure you rename the object to **Player**.

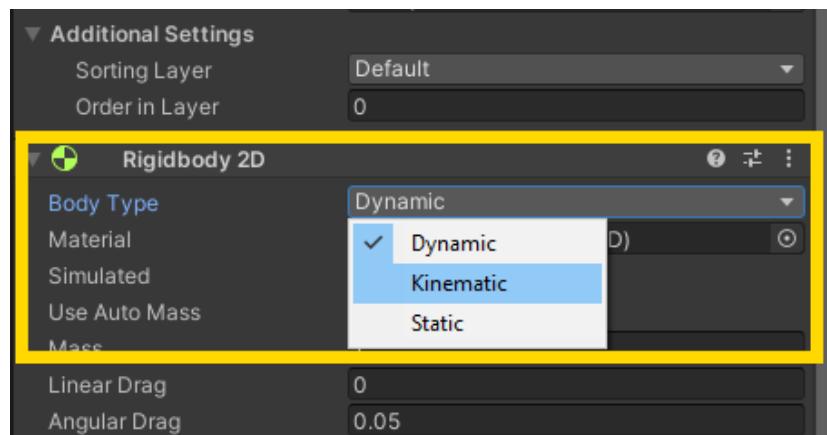
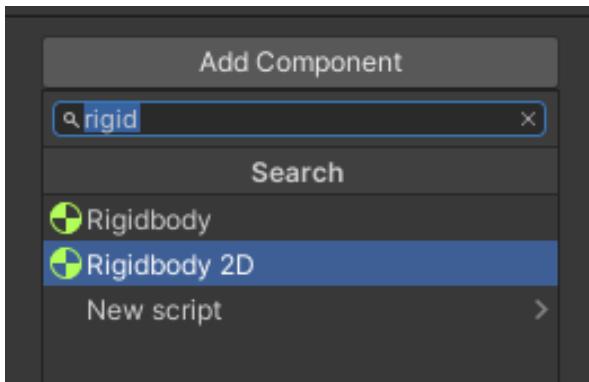


8

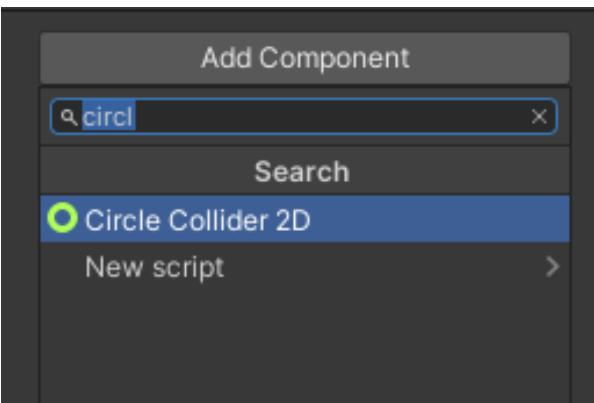
Let's move the player up slightly by **0.6**, and shrink its scale to something around **0.25** in both the **x** and **y**. While we are changing things, let's also change the material on the **Sprite Renderer** from **default** to **player**. Your player should currently look like this in the Inspector.



9 Click **Add Component**. Type **Rigidbody** in the search box and select the **Rigidbody 2D** component. Change the **Body Type** to **Kinematic**.

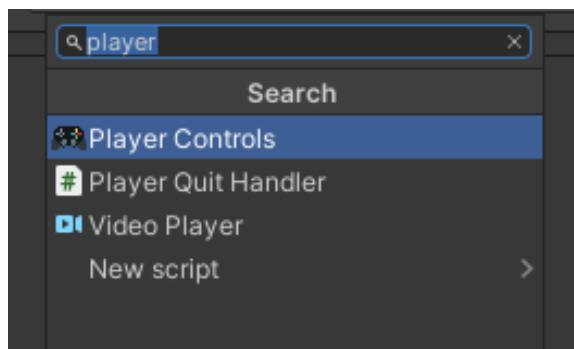


10 Click **Add Component** again. Type **Circle Collider** in the search box and select the **Circle Collider 2D** component.



11

Click **Add Component** one last time. Type **Player Controls** in the search box. Select the **Player Controls** script.



12

Open the **Player Controls** script and add the following variables to the top, just below the class line.

```
Unity Script | 0 references
public class PlayerControls : MonoBehaviour
{
    //movement speed
    [Header("Default Movement Speed")]
    public float moveSpeed = 200;
    //movement float
    float movement = 0;
```

13

Inside the **void Start** function, we are going to add a line that sets our Time Scale to its default, 1. This will be important later so that when we restart the game, everything is able to move as normal.

```
// Start is called before the first frame update
void Start()
{
    //Timescale is the speed the game runs at
    Time.timeScale = 1;
```

- 14** Inside the **void Update** function, add the following. This will set our movement variable to either -1, 0 or 1 depending on what the player is inputting. If they input 'Left', then we get -1. If they input, 'Right' we get 1. If they don't input anything, we get 0.

```
20  
21  
22 // Update is called once per frame  
23     Unity Message | 0 references  
24 void Update()  
25 {  
26     //Movement equal to Left and Right input  
27     movement = Input.GetAxisRaw("Horizontal");  
28 }
```

- 15** Below **void Update**, create a **void FixedUpdate** function. Add the following:

```
27  
28     Unity Message | 0 references  
29 private void FixedUpdate()  
30 {  
31     //Object transformation rotates around  
32     //an object using the settings.  
33     transform.RotateAround(Vector3.zero, Vector3.forward, movement * Time.fixedDeltaTime * -moveSpeed);  
34 }
```

- 16** Create a **void OnTriggerEnter2D** function, then add the following:

```
34  
35     Unity Message | 0 references  
36 private void OnTriggerEnter2D(Collider2D collision)  
37 {  
38     //Set the game speed to 0  
39     Time.timeScale = 0;  
40 }
```

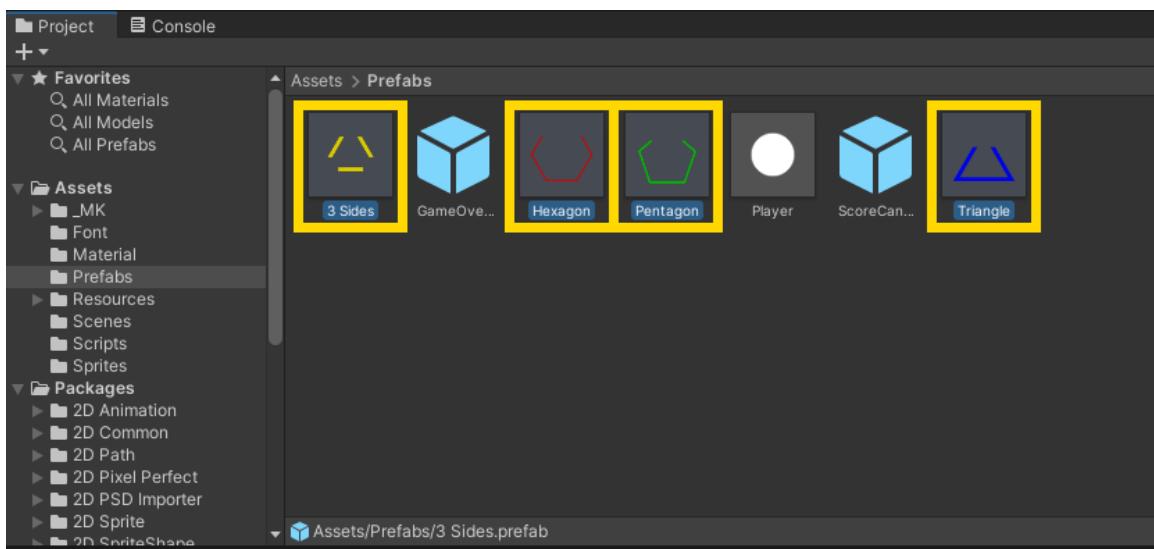
Save the script and return to Unity.

- 17** Play the game. Use the left and right arrows to move the player. You should see the player rotate in a circle.

- 18** Stop the game.

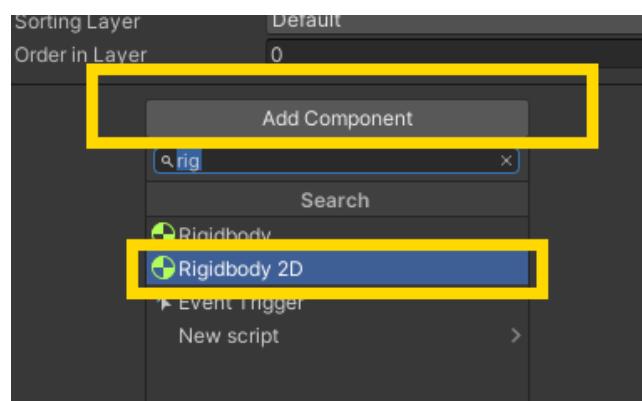
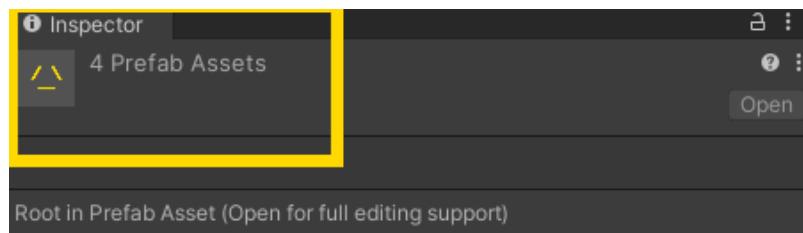
19

Go into the **Prefabs** folder and you will see four shaped polygons: 3 Sides, Hexagon, Pentagon, and Triangle. The next few steps apply to all four of the shapes listed above. Hold down **Ctrl** on your Keyboard and **one by one**, click each shape to select them all.

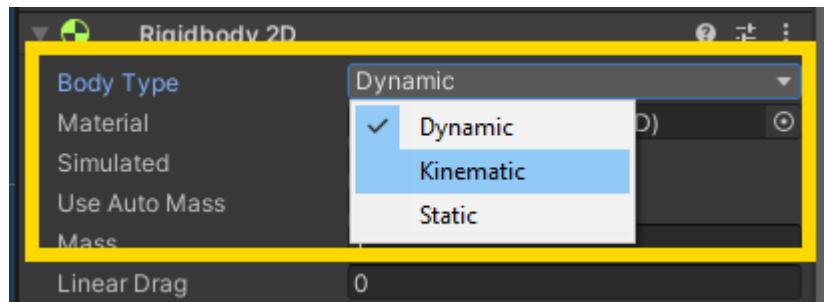


20

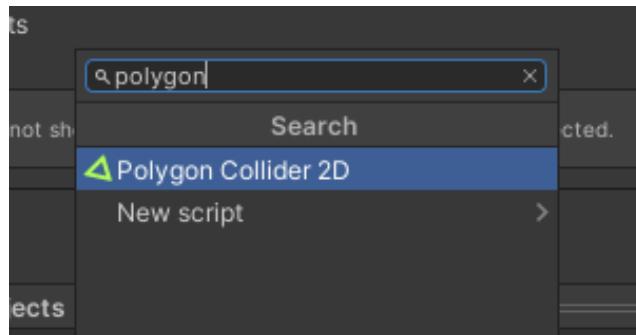
With all four shapes selected, click **Add Component**. Type **Rigidbody2D** in the search box and select the **Rigidbody 2D** component. To confirm all are selected, you should see “**4 Prefab Assets**” at the top.



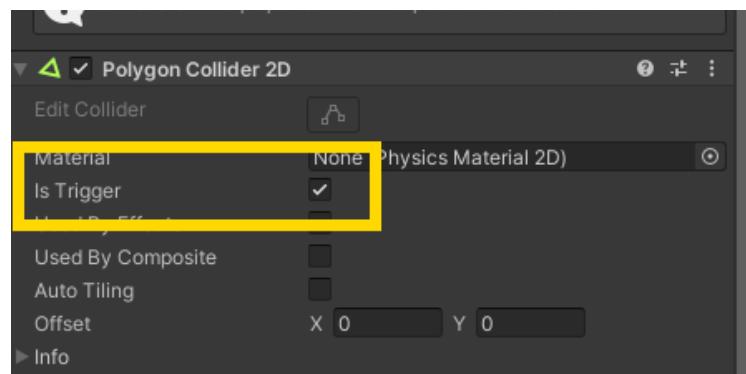
21 Change the Rigidbody's **Body Type** to Kinematic.



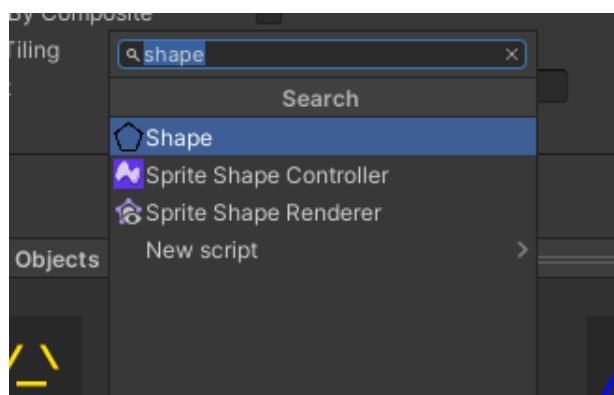
22 Next, with the four shapes still selected, click **Add Component** again. Type **Polygon Collider 2D** in the search box and select the **Polygon Collider 2D** component.



23 Under the **Polygon Collider 2D**, select the checkbox for **Is Trigger**.



- 24** Once more, click **Add Component**. Type **Shape** in the search box and select the **Shape** script component.



- 25** Open the **Shape** script, then add the following variables just below the bracket for the class line.

```
④ Unity Script (12 asset references) | 0 references
5  public class Shape : MonoBehaviour
6  {
7      //Rigidbody for the object
8      [Header("Rigidbody Object")]
9      public Rigidbody2D rb;
10     //Shrinking Speed
11     [Header("Default Shrinking Speed")]
12     public float shrinkSpeed = 3;
13 }
```

- 26** Inside the **void Start** function, add the following:

```
15
16
17 // Start is called before the first frame update
18 ④ Unity Message | 0 references
19 void Start()
20 {
21     //Rotation of the rigidbody
22     //at a random range
23     rb.rotation = Random.Range(0, 360);
24     //local scale for the Hexagon
25     //equals one for all axes (x,y,z) times ten
26     //meaning - localScale = (1,1,1) * 10
27     transform.localScale = Vector3.one * 10;
28 }
```

27 Inside the **void Update** function, add the following:

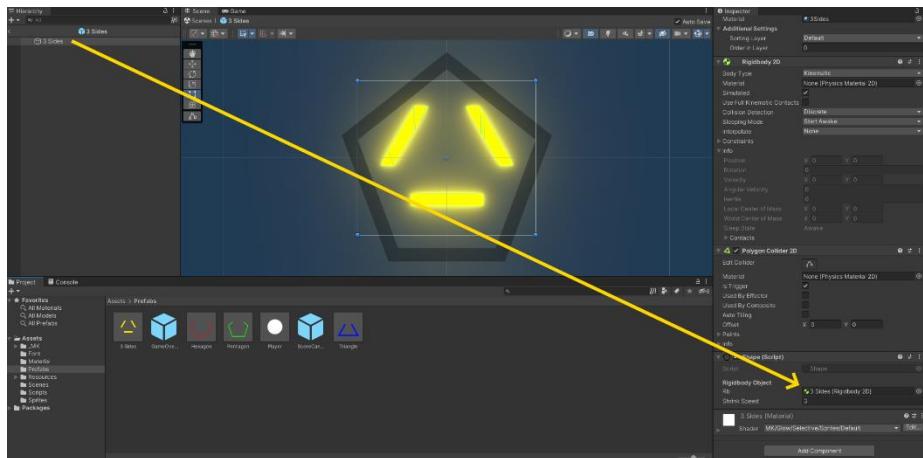
```
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
    // Update is called once per frame
    void Update()
    {
        //slowly shrink our localScale by
        //our shrinkSpeed multiplied by game rate
        transform.localScale -= Vector3.one * shrinkSpeed * Time.deltaTime;

        //When the object gets too small
        if (transform.localScale.x < 0.05f)
        {
            //Destroy Object
            Destroy(gameObject);
        }
    }
}
```

Save the script and return to Unity.

28 Go back into the **prefabs** folder and select one of the shape prefabs. Although we have been editing them together, we need to do them one at a time for this next step.

Double click into a shape prefab and place the **Rigidbody2D** into the **Inspector** field by dragging it.

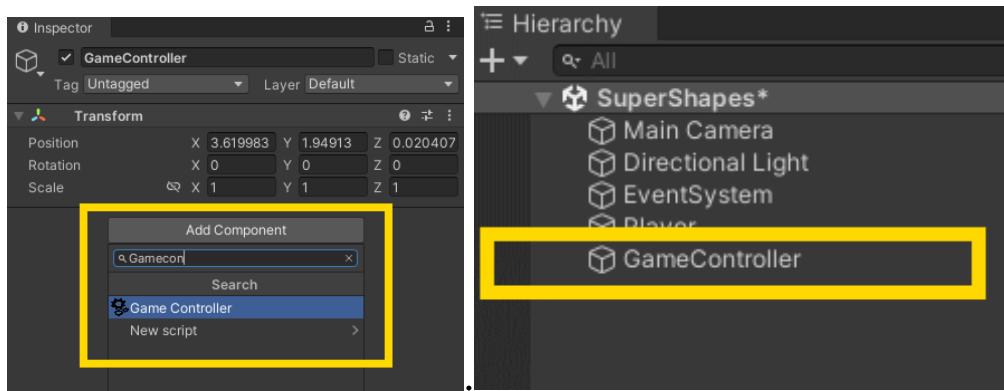


29 Repeat step 28 with the remaining shapes.

All your shapes should have a **Rigidbody2D** set to **Kinematic**, a **Polygon Collider** set to **Is Trigger**, and the **Shapes** Script with the correct **Rigidbody2D** in the **RB** field.

30

Before you play the game, add a Spawner to spawn multiple random shapes. See if you can add a new GameObject on your own and rename it **GameController**. Click **Add Component**. Type **Game Controller** in the search box and select the **Game Controller** script component.



31

Open the **Game Controller** script and add the following variables:

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4  using UnityEngine.SceneManagement;
5
6  public class GameController : MonoBehaviour
7  {
8
9
10 //The [] tells Unity that we want an Array
11 [Header("Shape Objects")]
12 public GameObject[] shapePrefabs;
13 //The first object will spawn after
14 //the spawnDelay and then every spawnTime
15 [Header("Default Spawn Delay Time")]
16 public float spawnDelay = 2;
17 [Header("Default Spawn Time")]
18 public float spawnTime = 3;
```

32

Inside the **void Start** function, add the following:

```
17
18
19
20
21 // Start is called before the first frame update
22 [Header("Message")]
23 void Start()
{
    InvokeRepeating("Spawn", spawnDelay, spawnTime);
}
```

33

Create a **void Spawn** function just below **void Start** by adding the following:

```
17  
18 // Start is called before the first frame update  
19     @UnityMessage(0 references)  
20     void Start()  
21     {  
22         InvokeRepeating("Spawn", spawnDelay, spawnTime);  
23     }  
24  
25     0 references  
26     void Spawn()  
27     {  
28         //Get a random number between the range of 0 and our  
29         //array length  
30         int randomInt = Random.Range(0, shapePrefabs.Length);  
31         //spawn new hexagon that was picked randomly  
32         Instantiate(shapePrefabs[randomInt], Vector3.zero, Quaternion.identity);  
33     }
```

34

Create a **void Game Over** function, add the following function just below the end of **void Spawn**:

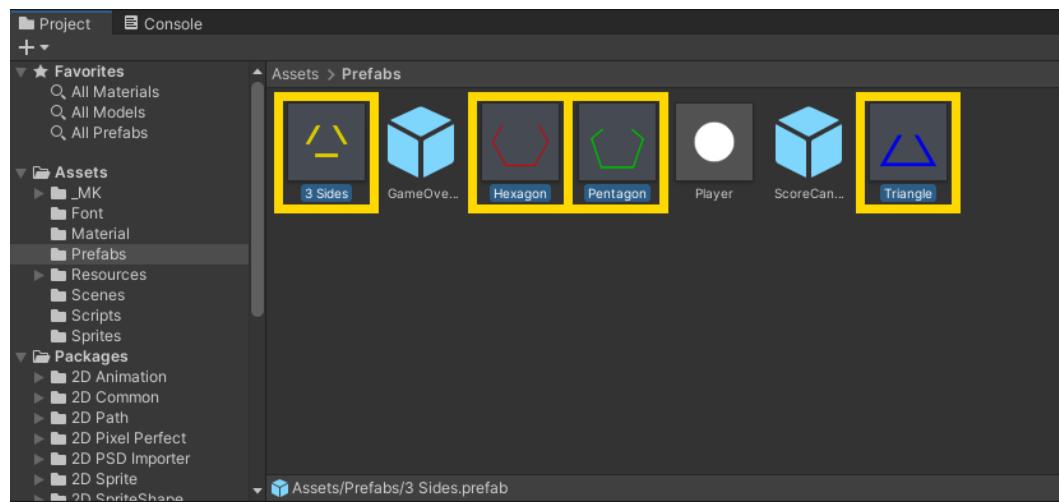
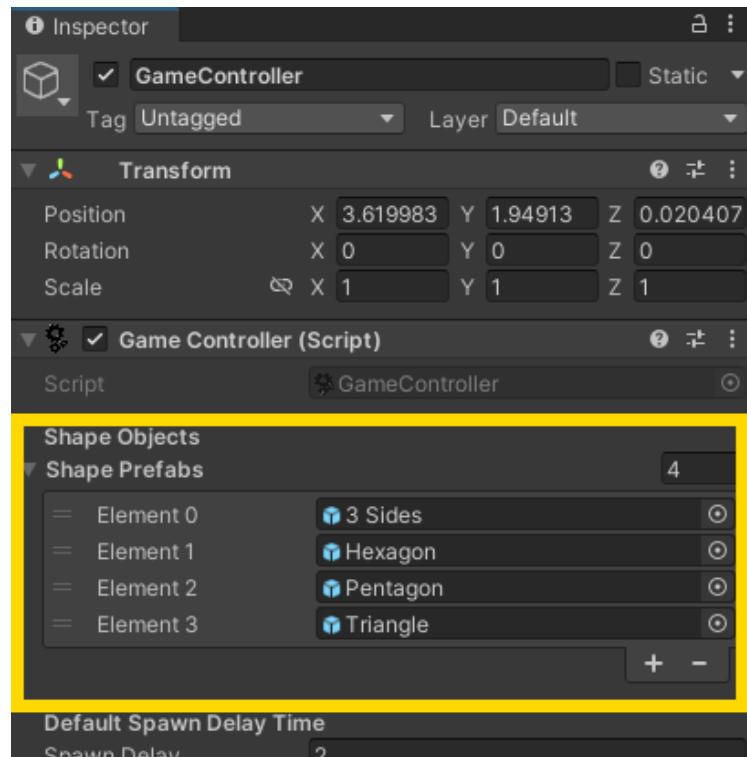
```
33  
34     0 references  
35     public void GameOver()  
36     {  
37         CancelInvoke("Spawn");  
38     }
```

Save the script and return to Unity.

35

Inside the Project window, you will find the **prefabs** folder that contains the four Prefabs we want to add to our array. Next, select the **Game Controller** and one by one drag the prefabs into the **Shape Prefabs Array**.

If you are having trouble with this, you can also type the number **4** into the array **Size** and drag them in that way, or search for them with the little dot next to each element.



36

Play the game. All of the shapes will begin to appear and shrink. If you touch one of the objects, the game will end.

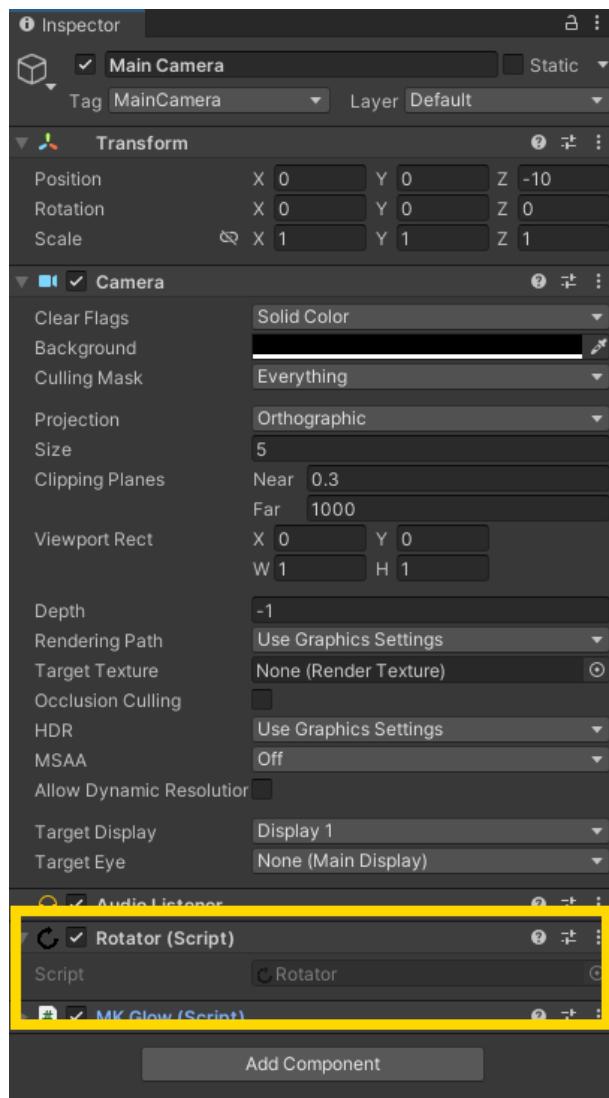
37

Stop the game.

38

Now to add more difficulty to the game. Start by selecting the **Main Camera** and loading the **Rotator** script component attached.

If you don't see the **Rotator** script, you may need to add it by pressing **Add Component** and typing **Rotator**.



39

Delete the **void Start** function. Inside the **void Update** function, add the following:

```
5  Unity Script (1 asset reference) | 0 references
6  public class Rotator : MonoBehaviour
7  {
8      // Update is called once per frame
9      void Update()
10     {
11         transform.Rotate(Vector3.forward, Time.deltaTime * 30);
12     }
13 }
```

40

Save the script and return to Unity. Play the game. The **Main Camera** will begin to rotate, which will add more difficulty to the game.

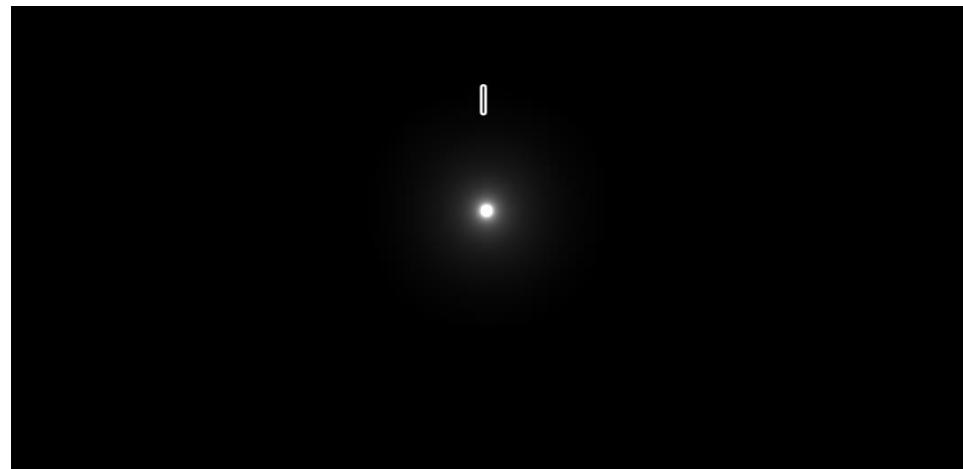
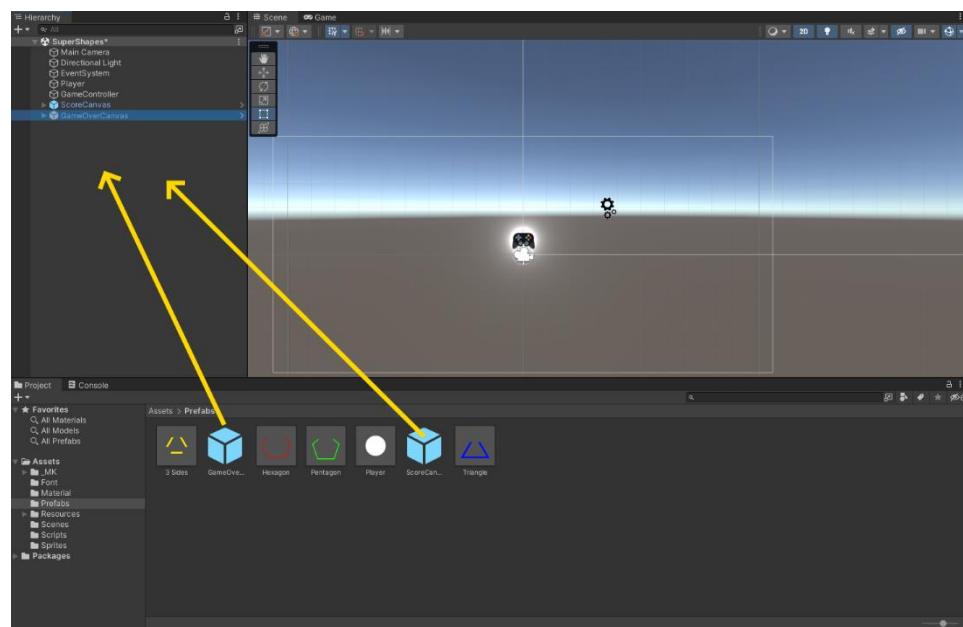
41

Stop the game.

42

Time to add scoring and the Game Over screen. To do so, add both the **ScoreCanvas** and **GameOverCanvas** to the **Hierarchy** from the **Prefabs** folder. If you play the game, you'll notice that the score is now seen in the Game window.

If you cannot see the score. Double click it in the Hierarchy, it may be too high and outside of the canvas's white lines.



43

Reopen the **Game Controller** script and add the following variable:

```
④ Unity Script (1 asset reference) | 0 references
6  public class GameController : MonoBehaviour
7  {
8      //The [] tells Unity that we want an Array
9      [Header("Shape Objects")]
10     public GameObject[] shapePrefabs;
11     //The first object will spawn after
12     //the spawnDelay and then every spawnTime
13     [Header("Default Spawn Delay Time")]
14     public float spawnDelay = 2;
15     [Header("Default Spawn Time")]
16     public float spawnTime = 3;
17
18     [Header("Game Over UI Canvas")]
19     public GameObject gameOverCanvas;
20
```

44

Inside the **void GameOver** function, add the following:

```
36
37     0 references
38     public void GameOver()
39     {
40         CancelInvoke("Spawn");
41         //Set the gameOverCanvas to be visible
42         gameOverCanvas.SetActive(true);
43     }
```

Save the script and return to Unity.

45

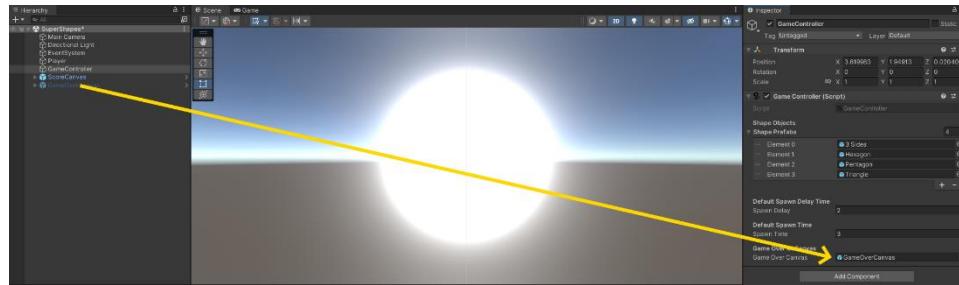
Reopen the **Shapes** script and inside the **void Update** function, add the following inside the **if** statement:

```
28
29     // Update is called once per frame
30     ④ Unity Message | 0 references
31     void Update()
32     {
33         //slowly shrink our localScale by
34         //our shrinkSpeed multiplied by game rate
35         transform.localScale -= Vector3.one * shrinkSpeed * Time.deltaTime;
36
37         //When the object gets too small
38         if (transform.localScale.x < 0.05f)
39         {
40             //Destroy Object
41             Destroy(gameObject);
42             Score.score++;
43         }
44     }
```

Save the script and return to Unity.

46

With the **GameController** selected, search and select the **GameOverCanvas** game object in the input field of the **Game Controller** script component. You can also drag it from the Hierarchy.



47

In the **PlayerControls** script, replace the code inside the **OnTriggerEnter2D** function with the following:

```

34
35
36 //This function runs if we overlap with a collider set to Trigger
37 @Unity Message | 0 references
38
39 private void OnTriggerEnter2D(Collider2D collision)
40 {
41     GameObject.Find("GameController").GetComponent<GameController>().GameOver();
42 }

```

48

Save the script and return to Unity.

49

Reopen the **GameController** script and add the following to the **GameOver** function:

```

36
37     public void GameOver()
38     {
39         CancelInvoke("Spawn");
40         //Set the gameOverCanvas to be visible
41         gameOverCanvas.SetActive(true);
42         Time.timeScale = 0;
43     }

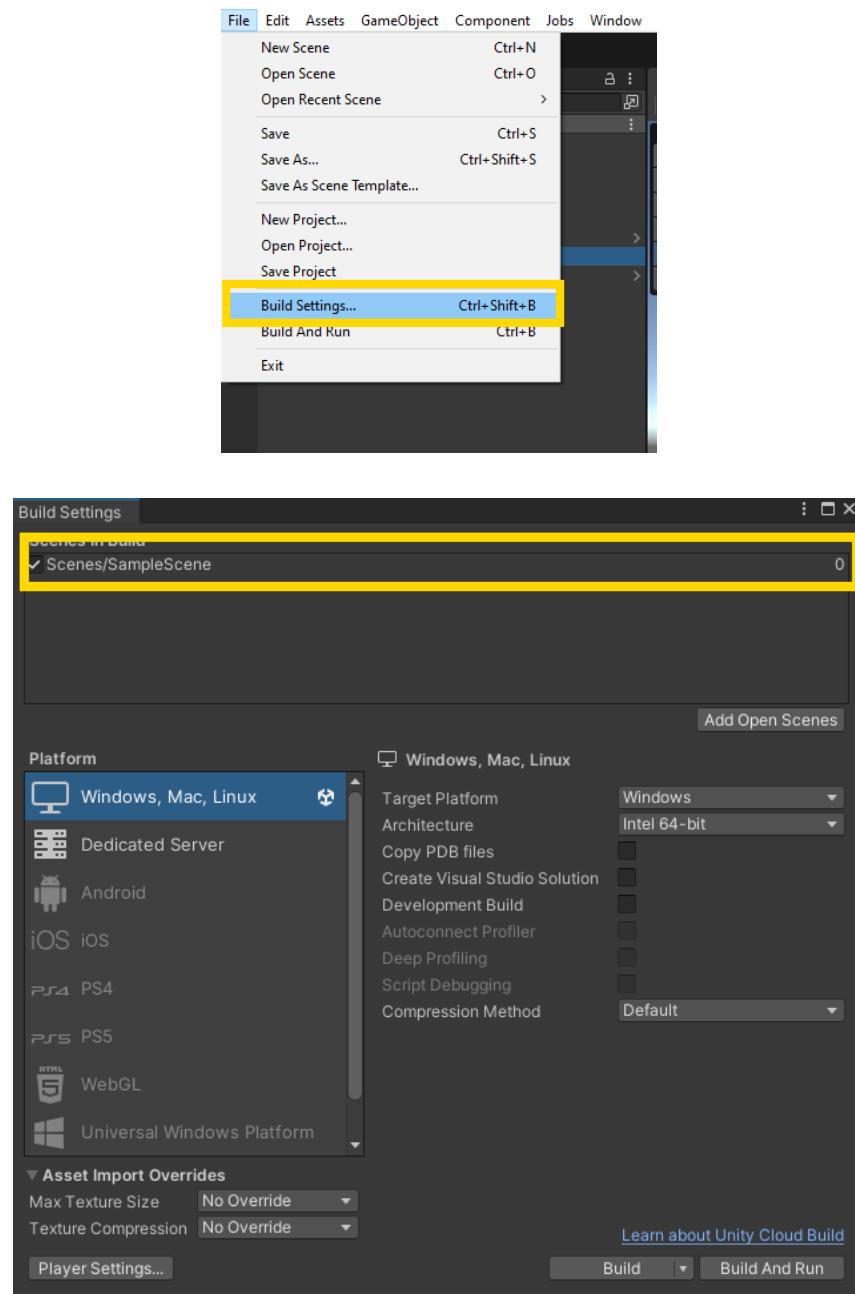
```

50

Save the script and return to Unity.

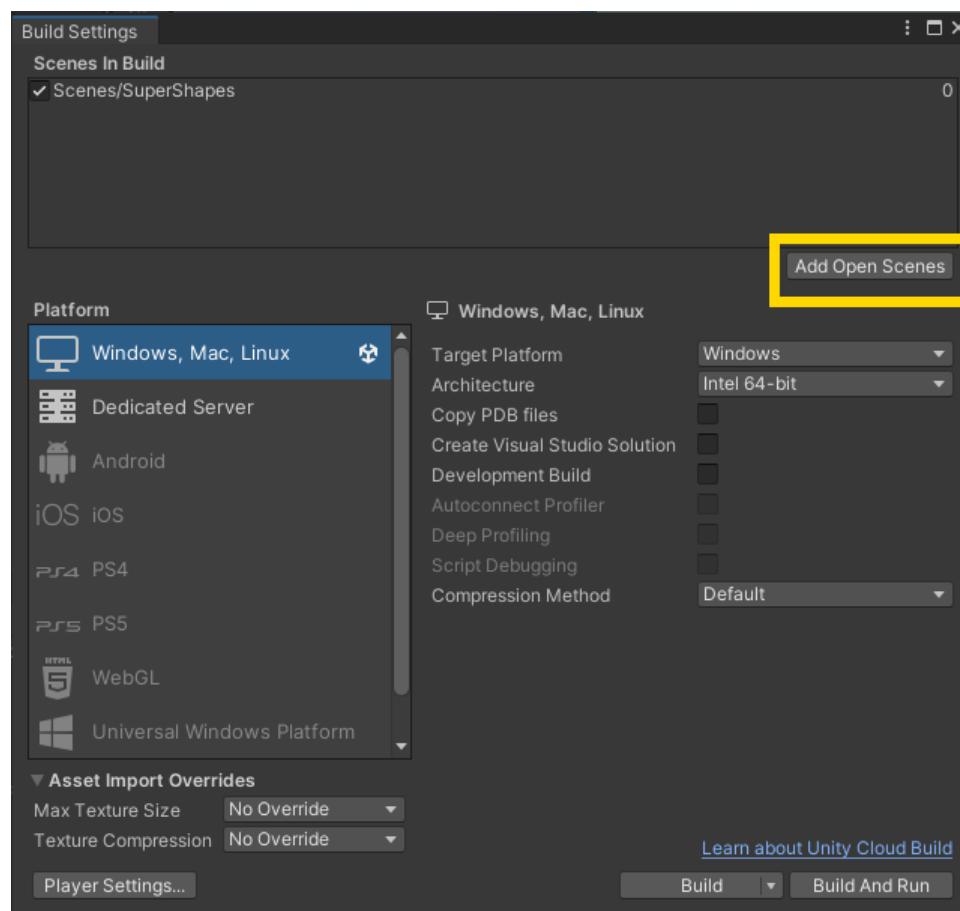
51

In order to get the play again button to work, we will have to adjust the build settings. Click the **File** tab then **Build Settings**. Select the Scenes/SampleScene and delete it.



52

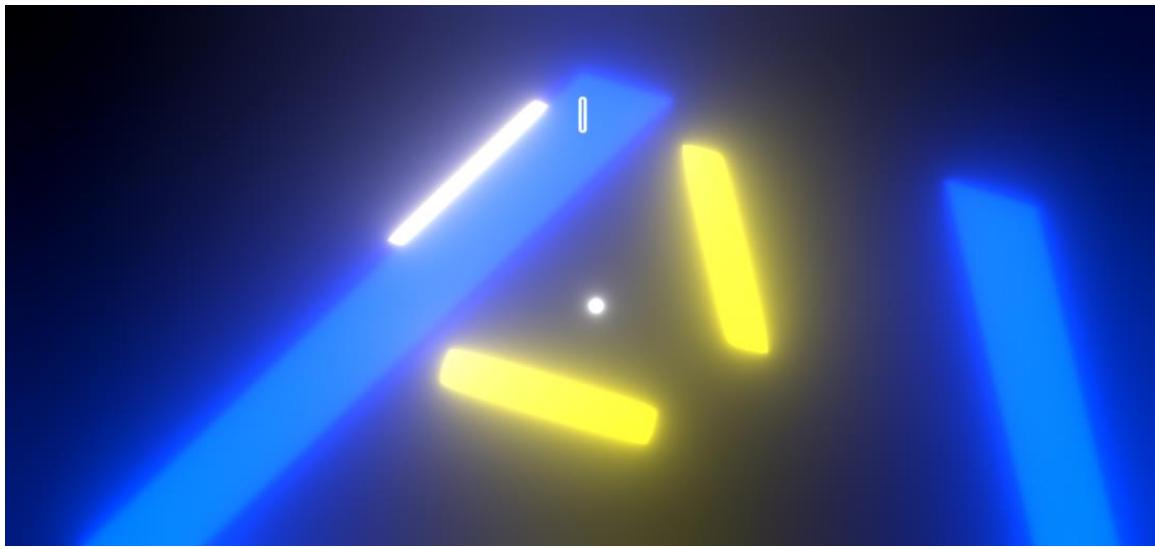
Once deleted, click **Add Open Scenes**. This will allow for the game to restart.



53

Play your game and see what you get. **Save** your game and **Export** it. **Submit** your game.

Prove Yourself: Super Duper Shapes



In Super Shapes, the player would dodge through the gaps in shapes one at a time. Now, in the Super Duper Shapes Prove Yourself, several shapes come at the player at once.

To build it, you will:

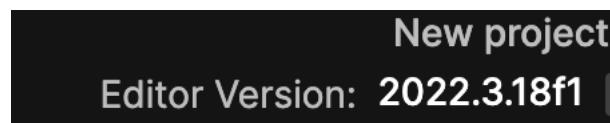
- 1. Duplicate each of the 4 original prefabs.**
- 2. Change the shape and size of the duplicated prefabs.**
- 3. Change the shrink and rotation speed of the shapes so that there is enough room for the player to still survive.**
- 4. Add the new shapes to the array so they correctly spawn.**

It's up to you how these new shapes look. Try combining two shapes together or make harder versions of the original shapes.

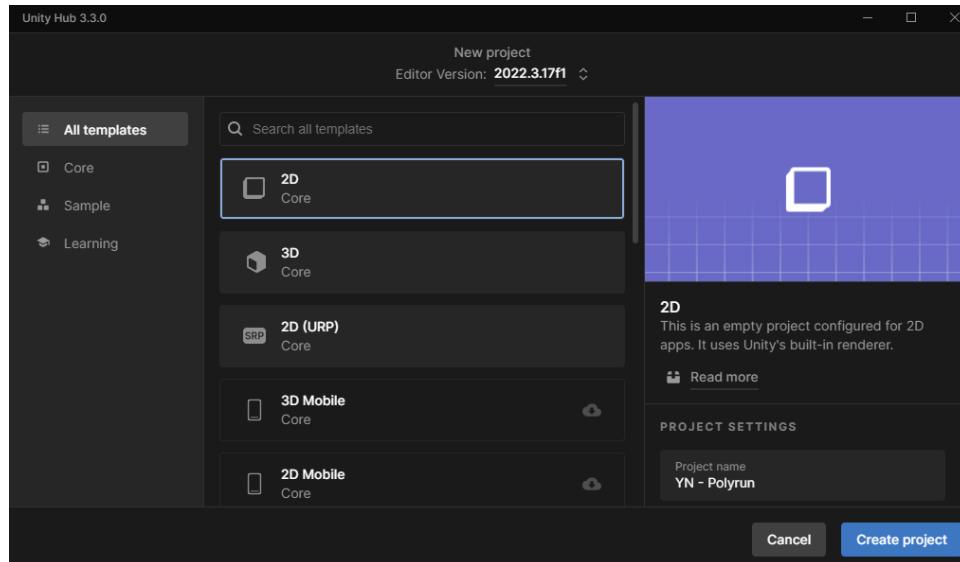
Activity 7: PolyRun

It's time to make another "runner" game!

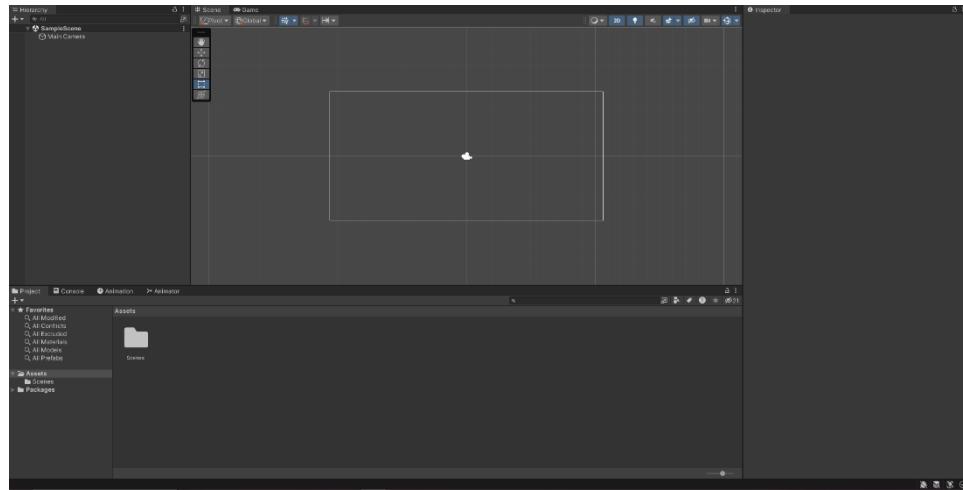
- 1 Open the Unity Hub application on your computer. Select the **New Project** button in the upper right-hand corner. Make sure the **2022.3 LTS** version is being used.



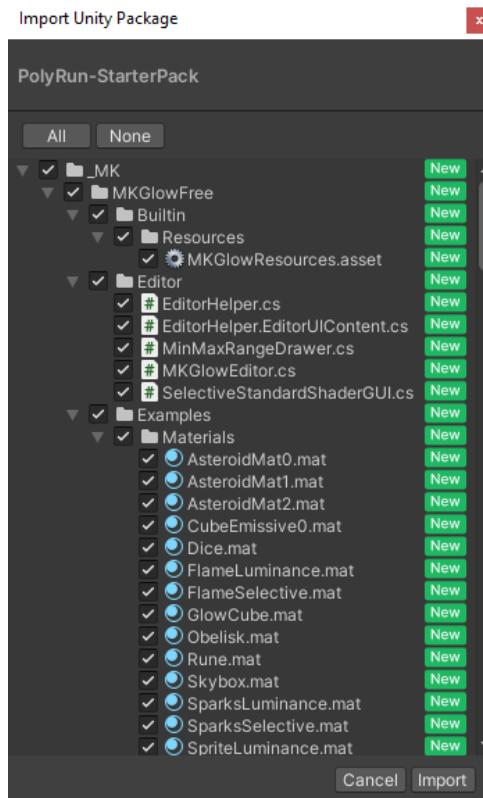
- 2 Select **2D core** in the Templates column. Next, type your first and last initials with the name of the project. For example, John Smith would save their project as **JS-PolyRun**. In the image below YN is used for "Your Name". Select the folder location where you want to save your project. Click the **Create** button.



3 As soon as Unity has loaded all the necessary assets into the program, this is what you'll see. Notice that we have no assets or scripts in the project. Let's fix that.

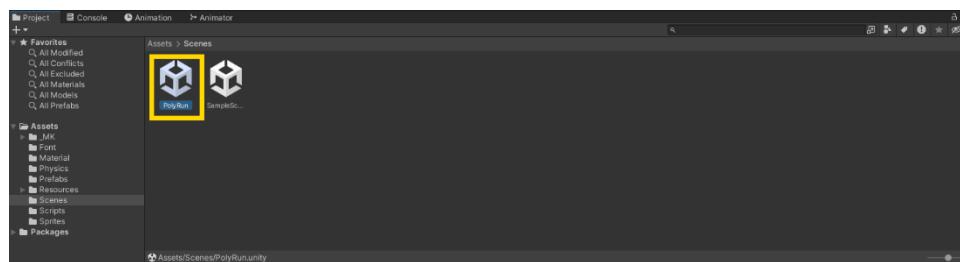


- 4** Go to the **Assets** menu. Select **Import Package** and click **Custom Package**. Select the **Activity 07 - PolyRun-starterpack UnityPackage** file. Once loaded, Unity will show a menu of what is inside the package.

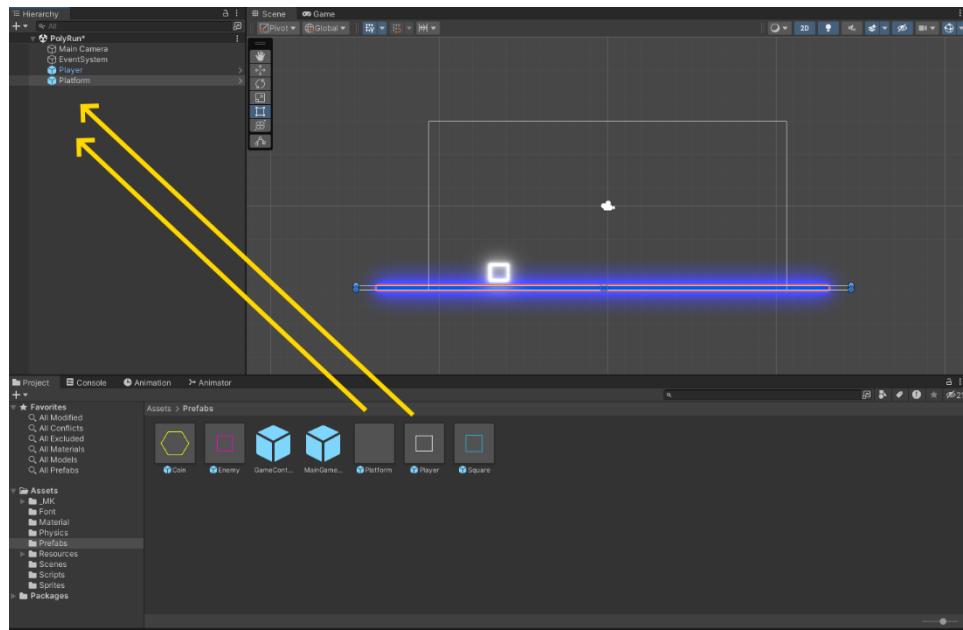


All of the material inside the Unity package should be selected. If not, click **All** to select everything, then click **Import**.

- 5** Even though a scene is open, there isn't anything in the Scene or Game windows. This is because we start in **SampleScene**; we want to go into the **Polyrun** scene. Go into **Scenes** and click the **Polyrun** scene.



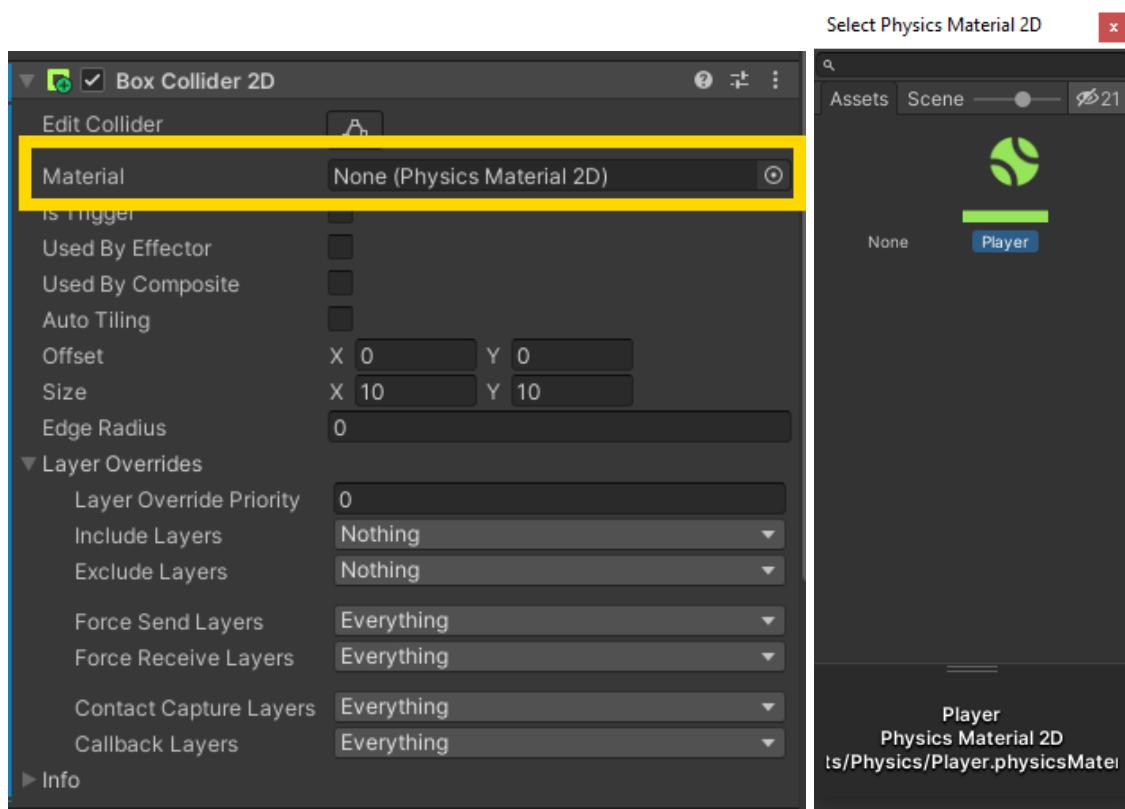
6 Go into the **Prefabs**, then drag the **Player** and **Platform** into the **Hierarchy**.



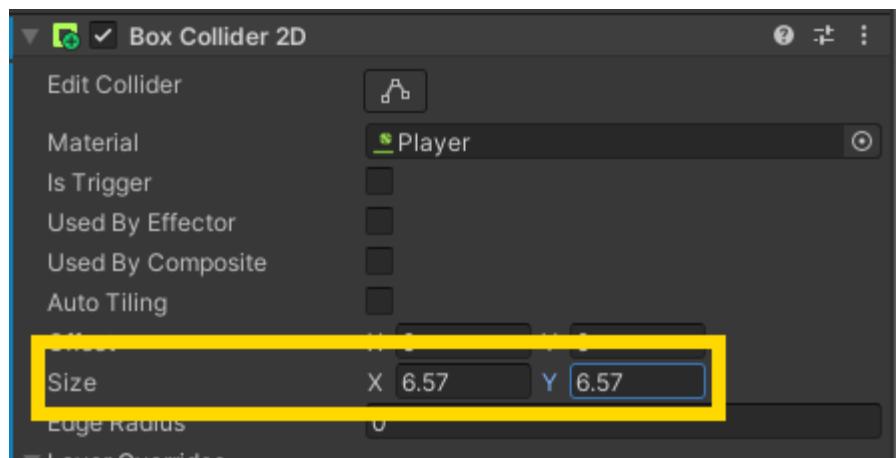
7 Next, select the **Player** GameObject. We need to add three components: a Box Collider 2D component, a Rigidbody 2D component, and... what's the third one? Can you figure it out?

If you can't figure it out, don't worry, the answer is on step 13.

8 Inside the **Box Collider 2D**, select the **Material** option and then the **Player Physics Material 2D** that appears.

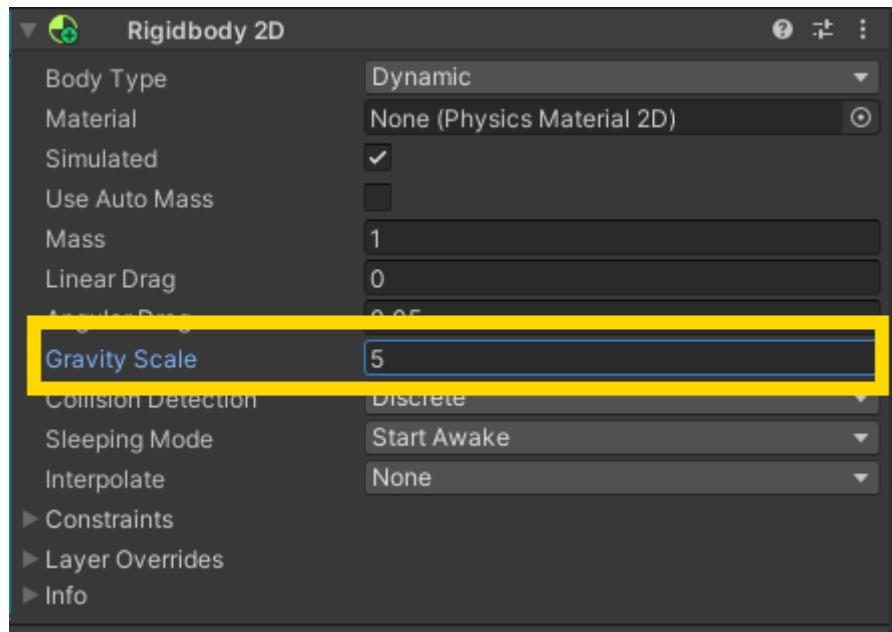


9 Change the size of the **Box Collider 2D** values. **X: 6.57, Y: 6.57**

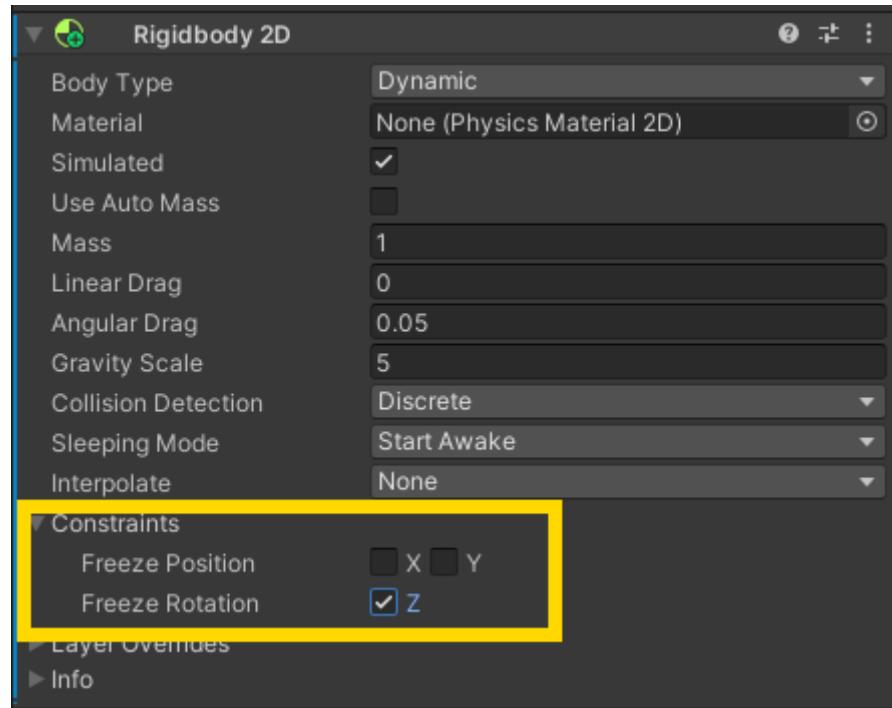


10 Next, click **Add Component**. Type **Rigidbody2D** in the search box and select the **Rigidbody 2D** component.

11 Change the **Gravity Scale** of the Rigidbody 2D values to **5**.

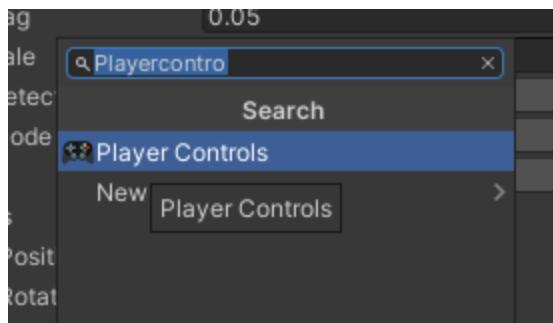


12 Click **Constraints**. Check the box next to **Freeze Rotation** for the Z-axis rotation constraints. This will stop physics from acting on its rotation, but still on its X and Y position.



13 Click **Add Component** one more time. We are going to add the final component that was talked about in step 7. We need to add the **Player Controls** script. Did you guess correctly?

Search for **Player Controls** and select the **Player Controls** script component.



14 Open the **Player** script and add the following variables:

```
5  Unity Script | 0 references
6  public class PlayerControls : MonoBehaviour
7  {
8      //Jumping power for the player object
9      [Header("Default Jumping Power")]
10     public float jumpPower = 6f;
11     //True or false if is on the ground
12     [Header("Boolean isGrounded")]
13     public bool isGrounded;
14     //position of the object
15     float posX = 0.0f;
16     //rigidbody2D of the object
17     Rigidbody2D rb;
```

15 Inside the **void Start** function, add the following:

```
17
18
19  // Start is called before the first frame update
20  void Start()
21  {
22      //Variable rb equals to Rigidbody2D
23      //component on the object
24      //this script is attached to
25      rb = GetComponent<Rigidbody2D>();
26
27      //posX = starting position
28      posX = transform.position.x;
```

-
- 16** Create the **void Update** function. Inside the function, add the following:

```
--  
30 // Update is called once per frame  
31 @Unity Message | 0 references  
32 void Update()  
33 {  
34     //Only jump if we press space and if isGrounded is true  
35     if (Input.GetKeyDown(KeyCode.Space) && isGrounded)  
36     {  
37         rb.AddForce(Vector3.up * jumpPower * rb.mass * rb.gravityScale * 20f);  
38     }  
}
```

Save the script and return to Unity.

-
- 17** If you were to go back into Unity, you'll see that the **isGrounded** checkmark doesn't change. We need to add some more code to detect if we are on the ground.

-
- 18** If you went to test the game, stop the game.

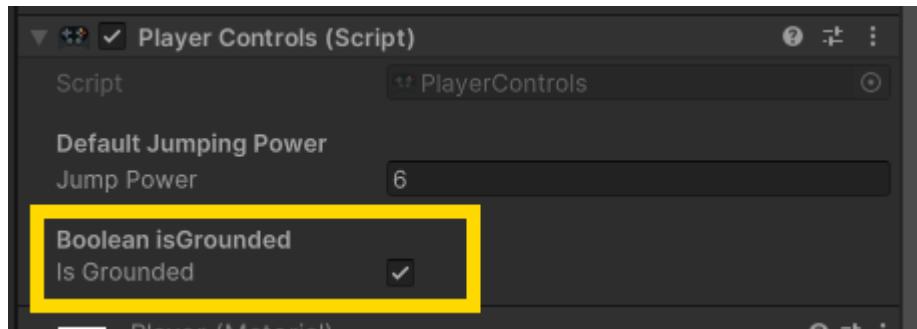
-
- 19** Inside the **Player Controls** script, add a new **void OnCollisionEnter2D** function. Add the following code inside.

```
@Unity Message | 0 references  
private void OnCollisionEnter2D(Collision2D collision)  
{  
    //If the object we collide with has the  
    //tag Ground  
    if(collision.gameObject.tag == "Ground")  
    {  
        //isGrounded is set to true  
        isGrounded = true;  
    }  
}
```

@

Save the script and return to Unity.

20 Playtest without maximizing the scene. While the scene is playing, select the **player** object and check to see if **isGrounded** is checked.



21 After confirming that, press the spacebar to make your player object jump. Then, stop the scene.

22 Now if you make the object jump, it will continue jumping upward. To correct that, go back into the **Player Controls** script and add two more functions.

First, create a function called **void OnCollisionStay2D**. Inside that function, add the following code:

```
Unity Message | 0 references
private void OnCollisionStay2D(Collision2D collision)
{
    //If the object we collide with has the
    //tag Ground
    if (collision.gameObject.tag == "Ground")
    {
        //isGrounded is set to true
        isGrounded = true;
    }
}
```

23 The next function to create is **void OnCollisionExit2D**. Inside the function, add the following:

```
Unity Message | 0 references
private void OnCollisionExit2D(Collision2D collision)
{
    //If the object we collide with has the
    //tag Ground
    if (collision.gameObject.tag == "Ground")
    {
        //isGrounded is set to true
        isGrounded = false;
    }
}
```

24 Playtest the scene to see if the ability to jump is working properly now. If it isn't working, double check the functions you just added. Make sure you are setting **isGrounded** to **false** in the **OnCollisionExit2D** function and not the others.

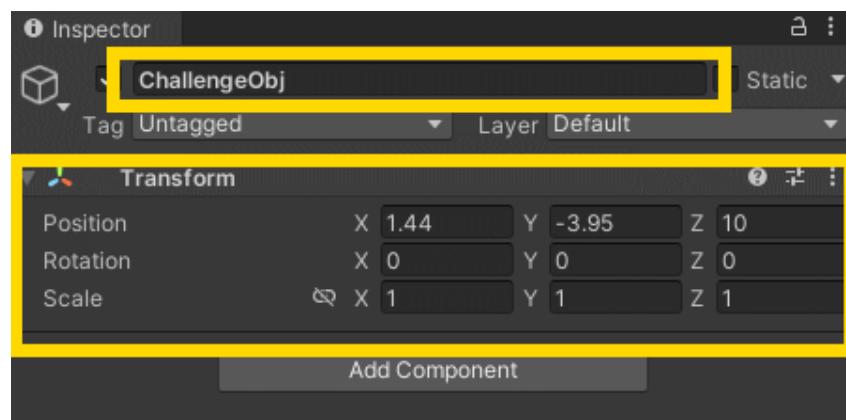
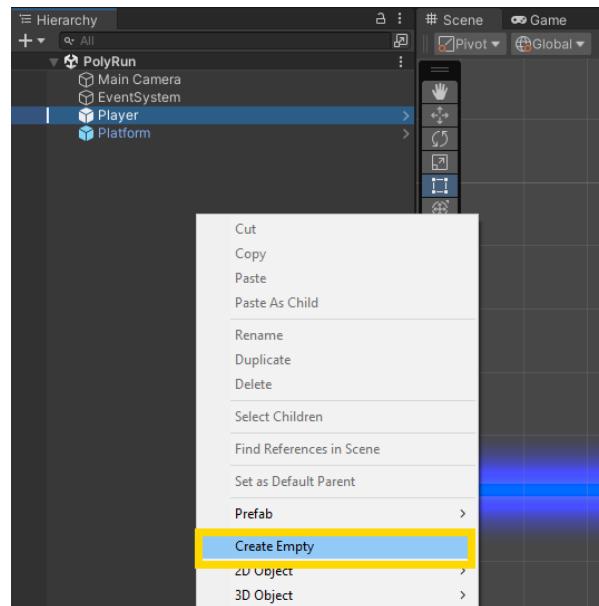
Stop the scene.

25 Save the script and return to Unity.

26 Now that we have the moving player, let's create an obstacle for the game.

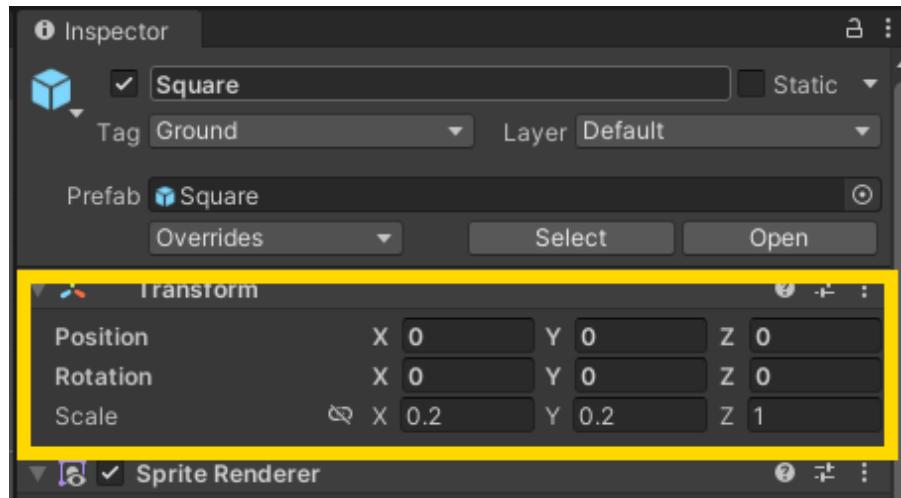
You can create any type of obstacle you want. These steps will show you how.

First, create an **Empty GameObject** and call it **ChallengeObj**. For now, we'll also change the position to something shown in the image.

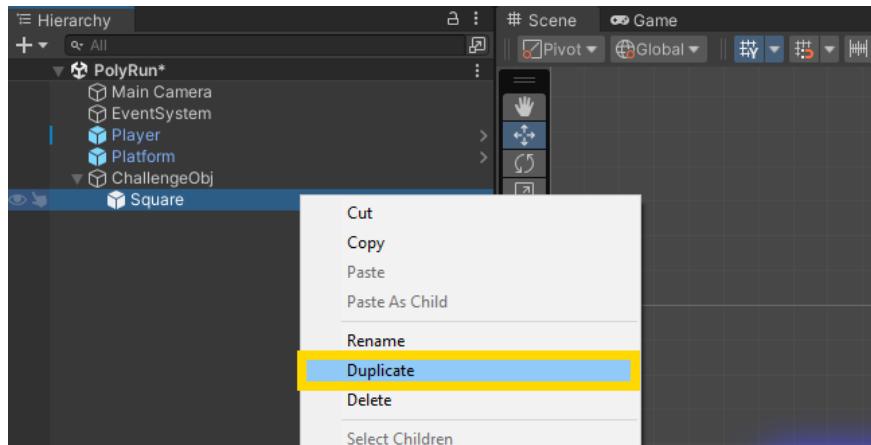


27 Go into the **Prefabs** folder and drag the **Square** to the **ChallengeObj**. It should be a child of the **ChallengeObj**.

This will give us our square that the player will need to jump over. Make sure to reset its position to **x:0, y:0, and z: 0 (0,0,0)**.



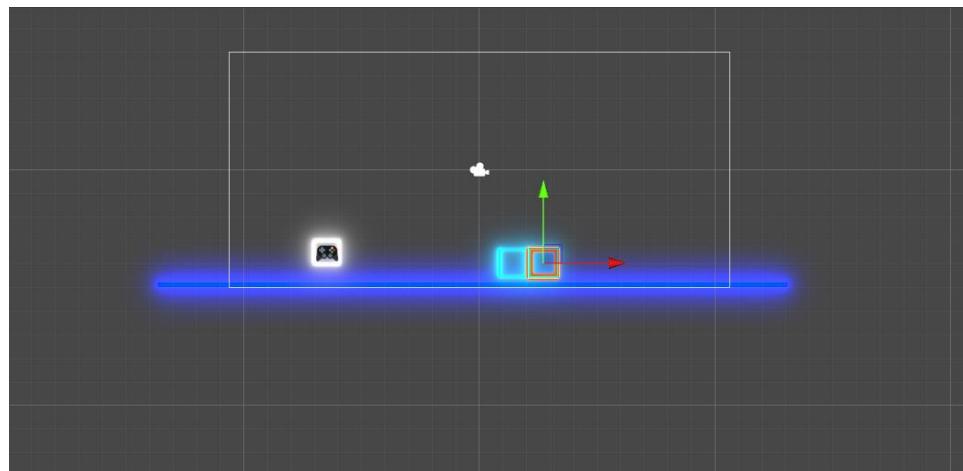
28 Next, right click the **Square** object and select duplicate.



29

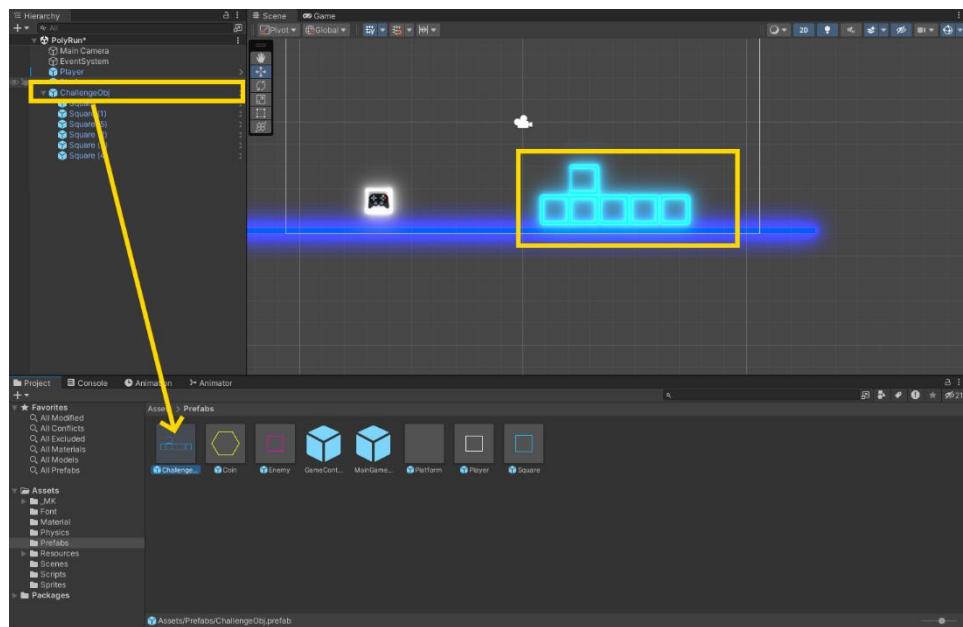
Move the new duplicated square to the right of the original square, as shown below.

We are going to repeat this step a few more times until we have our new object. You can make any shape, or choose one similar to the one below.



30

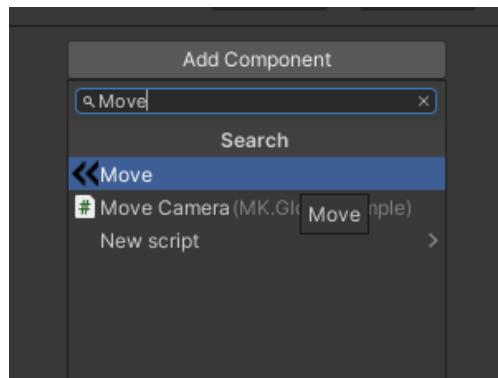
Once you've created your **ChallengeObj**, we need to turn it into a prefab. Creating a **Prefab** is simple. Drag the **ChallengeObj** from the **Hierarchy** to the **Prefs** folder.



31

Next, select the prefab you just made. We are going to be editing it a bit more. You should see **ChallengeObj (Prefab Asset)** at the top, so you know you are editing the prefab.

To make the challenge object move from right to left, first click **Add Component**. Type **Move** in the search box and select the **Move** script component.



32

Open the **Move** script and add the following variable:

```
5  public class Move : MonoBehaviour
6  {
7      [Header("Default Speed")]
8      public float speed = 6;
9  }
```

33

Inside the **void Update** function, add the following code:

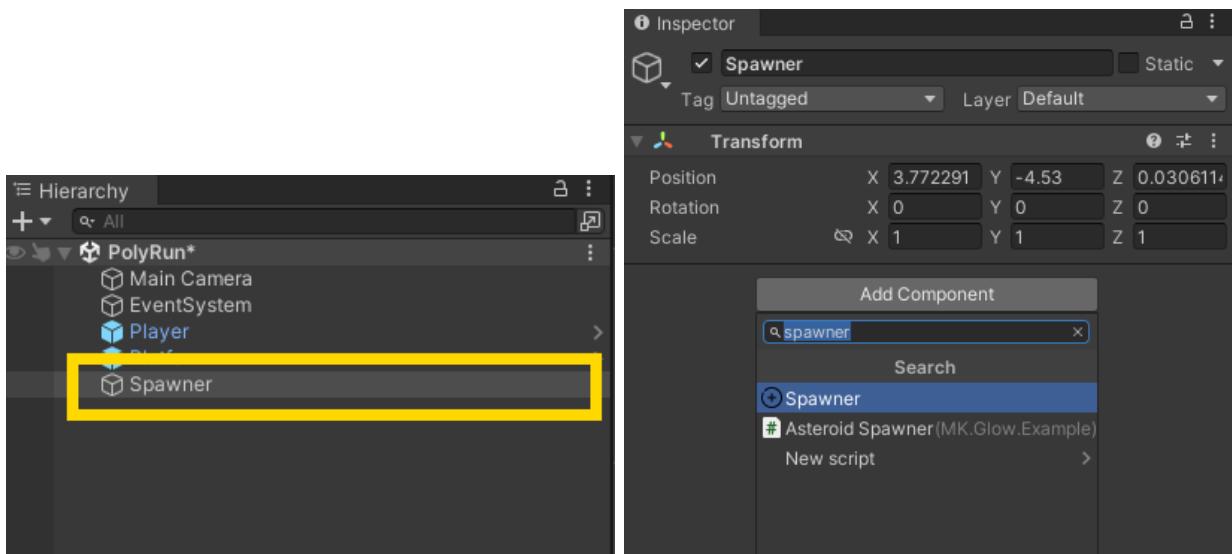
```
15
16      // Update is called once per frame
17      @Unity Message | 0 references
18      void Update()
19      {
20          //Add vector3.left (-1,0,0) to our transform, times by speed
21          transform.position += Vector3.left * speed * Time.deltaTime;
      }
```

Save the script and return to Unity.

34

You'll want to spawn the **ChallengeObj** as multiple clones. Start by creating a new **GameObject** and renaming it to **Spawner**.

35 Select the Spawner and click **Add Component**. Type **Spawner** in the search box and select the **Spawner** script component.



36 Open the **Spawner** script and add the following variables:

```
 5  public class Spawner : MonoBehaviour
 6  {
 7      [Header("ChallengeObj Game Object")]
 8      public GameObject challengeObject;
 9      [Header("Default Spawn Delay Time")]
10     public float spawnDelay = 1f;
11     [Header("Default Spawn Time")]
12     public float spawnTime = 2f;
13 }
```

37 Inside the **void Start** function, add the following:

```
14
15  // Start is called before the first frame update
16  void Start()
17  {
18      //start the function after
19      //spawnDelay (1) and Repeat the function
20      //InstantiateObject every spawnTime (2)
21      InvokeRepeating("InstantiateObjects", spawnDelay, spawnTime);
22 }
```

38

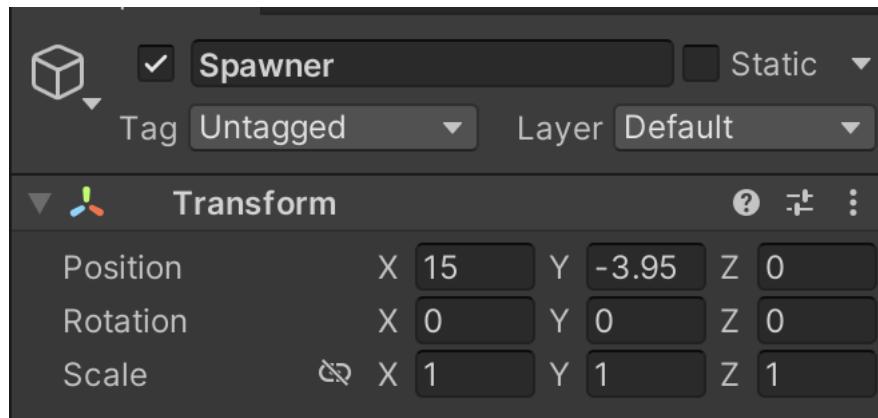
Create a new **void** function called **InstantiateObjects**. Add the following:

```
29
30     void InstantiateObjects()
31     {
32         Instantiate(challengeObject, transform.position, transform.rotation);
33     }
```

Save the script and return to Unity.

39

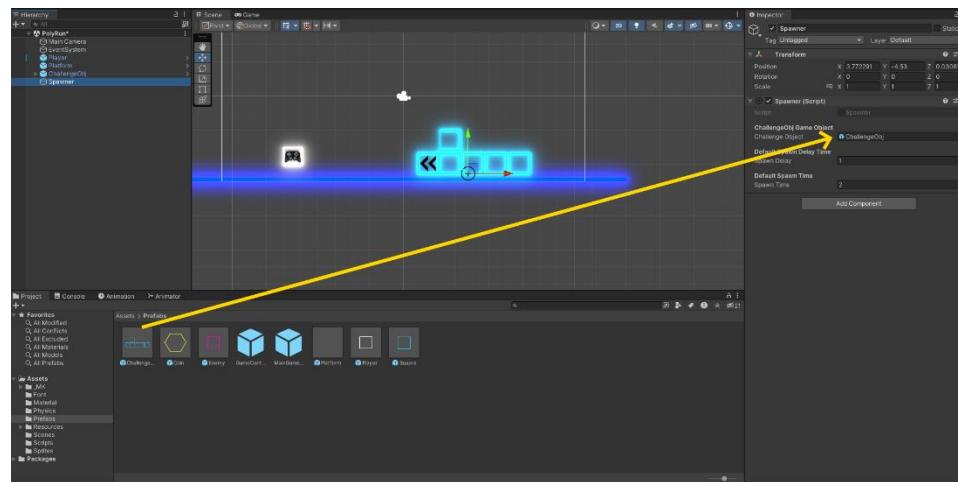
Next with the **Spawner** still selected set the position in the **inspector** to match the values below.



Position Values: **X:15, Y: -3.95, Z:0**

40

With the **Spawner** selected, search and select the **ChallengeObj** game object from the **Assets** in the Spawner Script component or drag it from the prefabs folder into the slot.



41 Delete the original **ChallengeObj** gameobject from the hierarchy.

42 Play the game. During game play, alter the Spawn time to adjust how fast you want the objects to spawn. If you are satisfied with the current settings, change the spawn time when you stop the game.

43 Stop the game.

44 You may have noticed that the player object gets pushed to the side if it doesn't jump onto the platform in time. To fix this, you will need to add to both the Player and the Spawn Controller scripts.

Open the **PlayerControls** script first. Create a new **void GameOver** function, add the following:

```
17 // Start is called before the first frame update
18 // Unity Message | 0 references
19 void Start()
20 {
21     //Variable rb equals to Rigidbody2D
22     //component on the object
23     //this script is attached to
24     rb = GetComponent<Rigidbody2D>();
25
26     //posX = starting position
27     posX = transform.position.x;
28
29 }
30
31
32 //The function to call when we GameOver
33 void GameOver()
34 {
35     Time.timeScale = 0;
36 }
37
```

We also need to reset our **TimeScale**, so add a small line **on Start** to reset it back to **1**. Otherwise, our game will stay paused!

45 Inside the **void Update** function, add the following:

```
38 // Update is called once per frame
39 // Unity Message | 0 references
40 void Update()
41 {
42     //Only jump if we press space and if isGrounded is true
43     if (Input.GetKeyDown(KeyCode.Space) && isGrounded)
44     {
45         rb.AddForce(Vector3.up * jumpPower * rb.mass * rb.gravityScale * 20f);
46     }
47
48     //If the player position is
49     //less than where it started
50     if(transform.position.x < posX)
51     {
52         //Execute Gameover function
53         GameOver();
54     }
55 }
56 
```

Save the script and return to Unity.

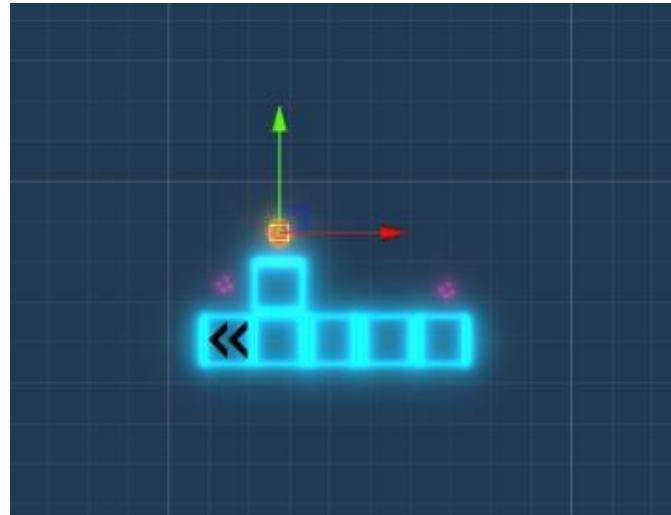
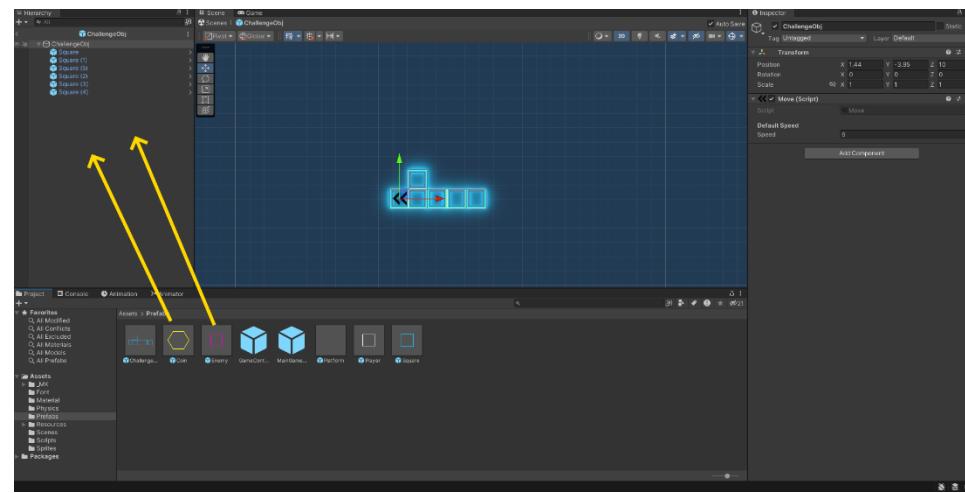
46 Play the game. Does the GameOver function work?

47 Stop the game.

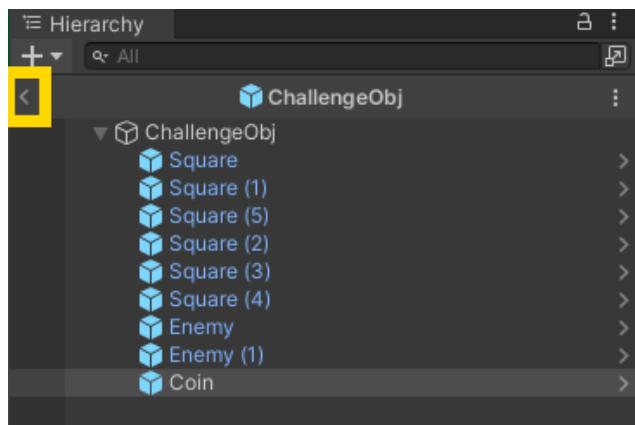
48 To add more objects to the ChallengeObj, go into the **Prefabs** folder. Double-click the **ChallengeObj** to open the prefab. You'll see the Scene window turn blue as this is where you would edit the prefab.

49

In the prefab folder, drag the **coin** prefab and **2 enemy** prefabs into the scene and place the objects according to the image below.



- 50** Click the arrow next to the prefab name to go back to the regular scene. The prefab will save automatically.



- 51** Go back to the **PlayerControls** script. Inside the **void OnCollisionEnter2D** function, add the following:

```
57     private void OnCollisionEnter2D(Collision2D collision)
58     {
59         //If the object we collide with has the
60         //tag Ground
61         if(collision.gameObject.tag == "Ground")
62         {
63             //isGrounded is set to true
64             isGrounded = true;
65         }
66
67         if(collision.gameObject.tag == "Enemy")
68         {
69             GameOver();
70         }
71     }
```

- 52** Create a new **void OnTriggerEnter2D** function and add the following:

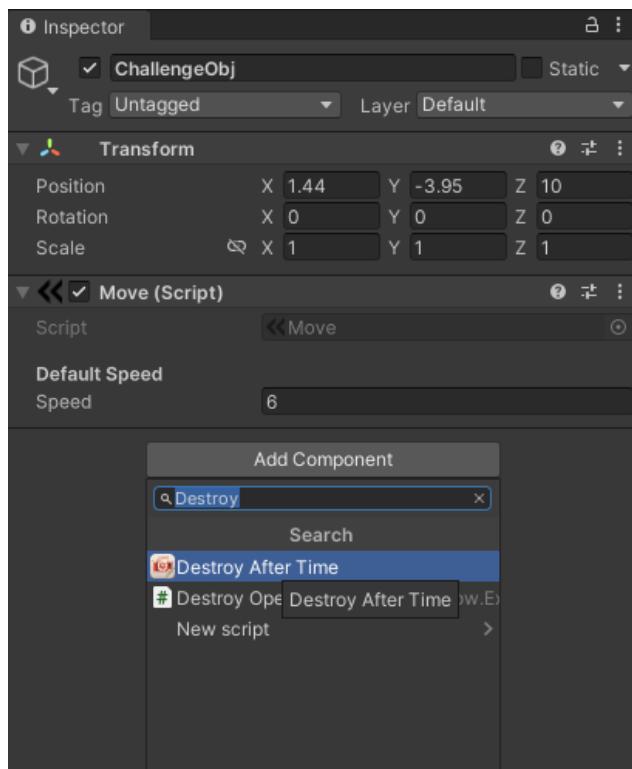
```
94
95     private void OnTriggerEnter2D(Collider2D collision)
96     {
97         if(collision.tag == "Coin")
98         {
99             Destroy(collision.gameObject);
100        }
101    }
```

Save the script and return to Unity.

53 Once back in Unity, play the scene. Collect the coins and try not to touch the enemies!

54 Stop the game.

55 At this point, you'll want to destroy the challenge objects that are still active off screen. You can destroy these objects by using a Destroyer script. Select the **ChallengeObj** in the prefab folder, select **Add Component**. Type **Destroy** in the search box and select the **Destroy After Time** script.



56 Open the **Destroyer** script and add the following variable:

```
 5  Unity Script (1 asset reference) | 0 references
 6  public class DestroyAfterTime : MonoBehaviour
 7  {
 8      [Header("Destruction Time")]
 9      public float timeToDestruction = 6;
```

57 Inside the **void Start** function, add the following:

```
10 // Start is called before the first frame update
11 Unity Message | 0 references
12 void Start()
13 {
14     Invoke("DestroyObject", timeToDestruction);
15 }
```

58 Create a new **void DestroyObject** function and add this to the script:

```
15
16 void DestroyObject()
17 {
18     Destroy(gameObject);
19 }
```

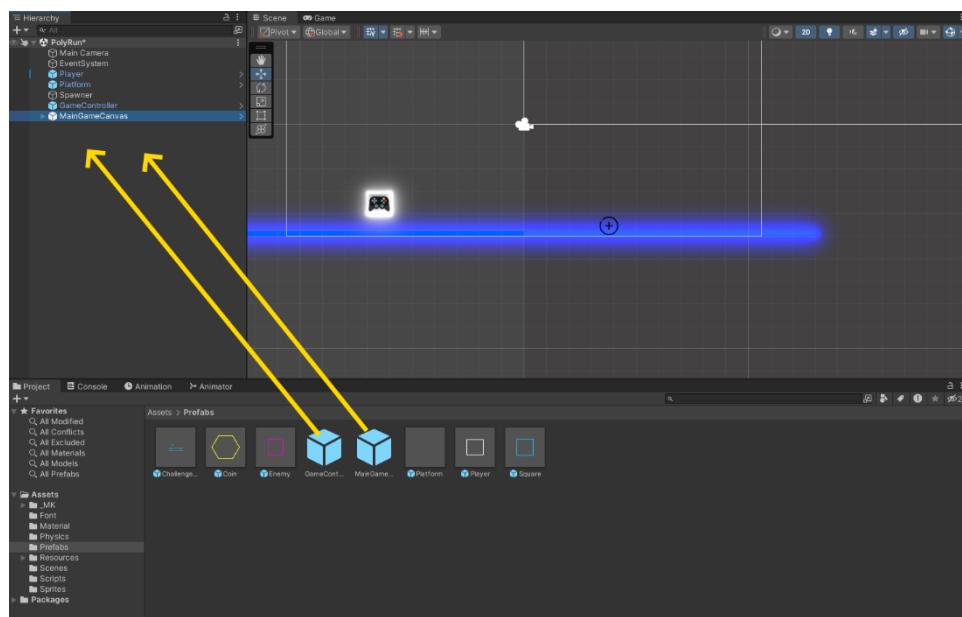
Save the script and return to Unity.

59 Play the game. Look at the Hierarchy, you should see the objects being destroyed.

60 Stop the game.

61

Like the other games, it's time to add scoring and a Game Over panel. Drag the **GameController** and **MainGameCanvas** to the Hierarchy.



62

Now, open the **PlayerControls** script. Inside the **OnTriggerEnter2D** function, add the following:

```
94
95     Unity Message | 0 references
96     private void OnTriggerEnter2D(Collider2D collision)
97     {
98         if(collision.tag == "Coin")
99         {
100             //Find the game controller and add to our score
101             GameObject.Find("GameController").GetComponent<GameController>().IncrementScore();
102
103             Destroy(collision.gameObject);
104
105 }
```

63

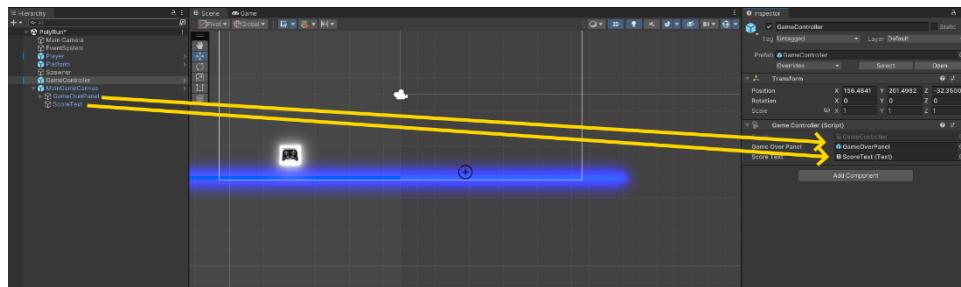
Inside the **void Game Over** function, delete the old **GameOver** function call. Then, add the following:

```
32
33     //The function to call when we GameOver
34     2 references
35     void GameOver()
36     {
37         //GameObject.Find("GameController").GetComponent<GameController>().GameOver();
38
39     }
```

Save the script and return to Unity.

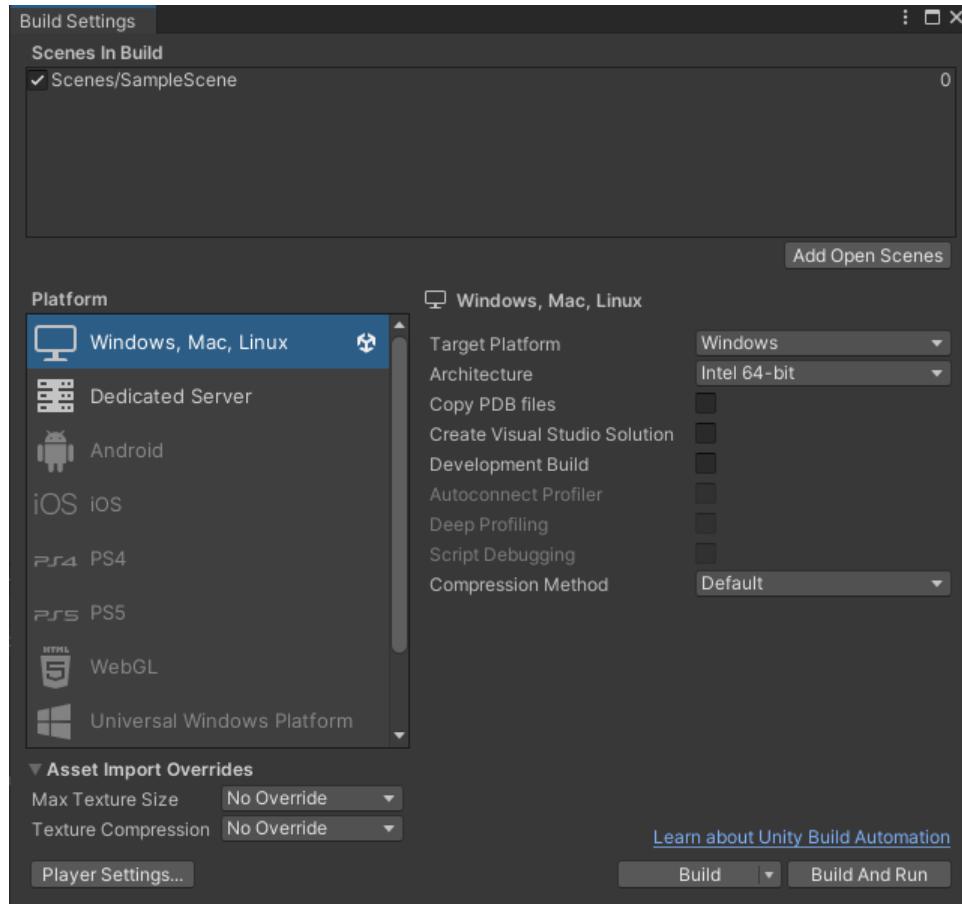
64 Select the **GameController** object. In the script components, in the **GameOverPanel** input field, search and select the **GameOverPanel** object in your Hierarchy. (You can also drag it into the slot from the Hierarchy).

In the same script component, we need to do the same for the **Score Text** input field, search and select **ScoreText** from the scene panel. (Or drag from the Hierarchy).

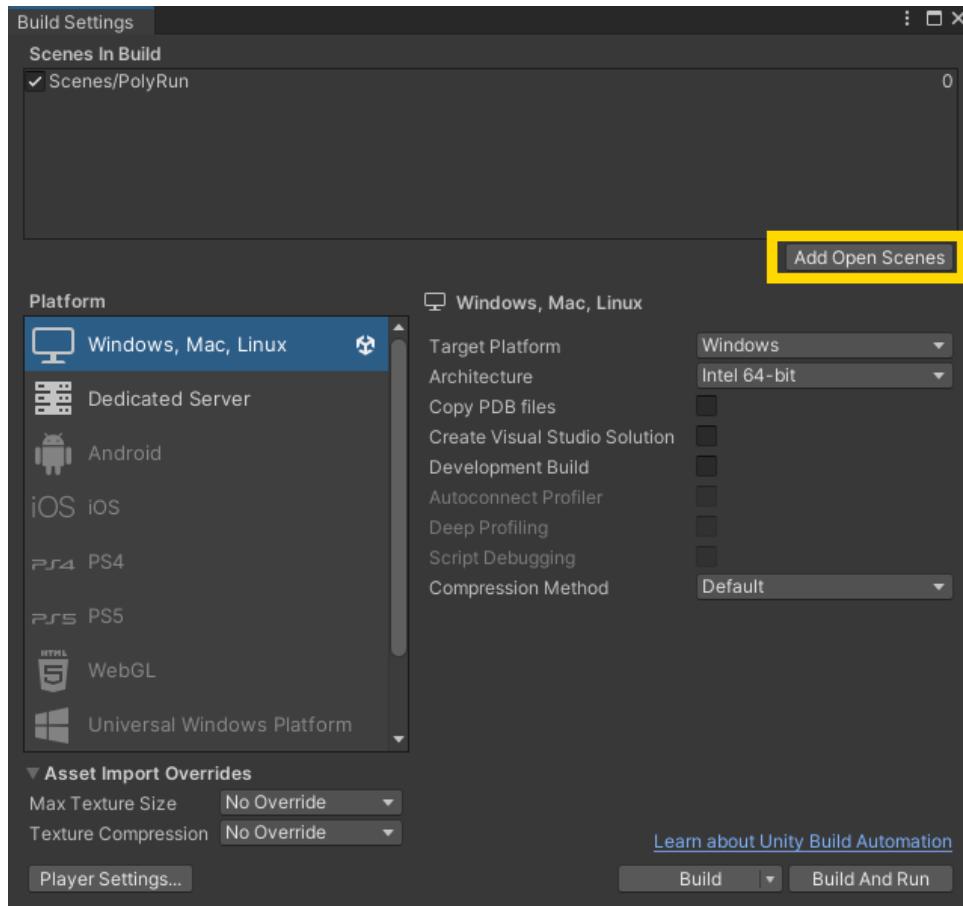


65 Play the scene. You'll see that if the player touches the coin the score counter shown above will add 1 to the score. If the player moves from its position on the X-axis or touches the enemy, the **GameOverCanvas** will appear to show the final score. There will also be a button to restart the game.

66 In order to get the play again button to work, we will have to adjust the build settings. Click the **File** tab then **Build Settings**. Select the Scenes/SampleScene and delete it.



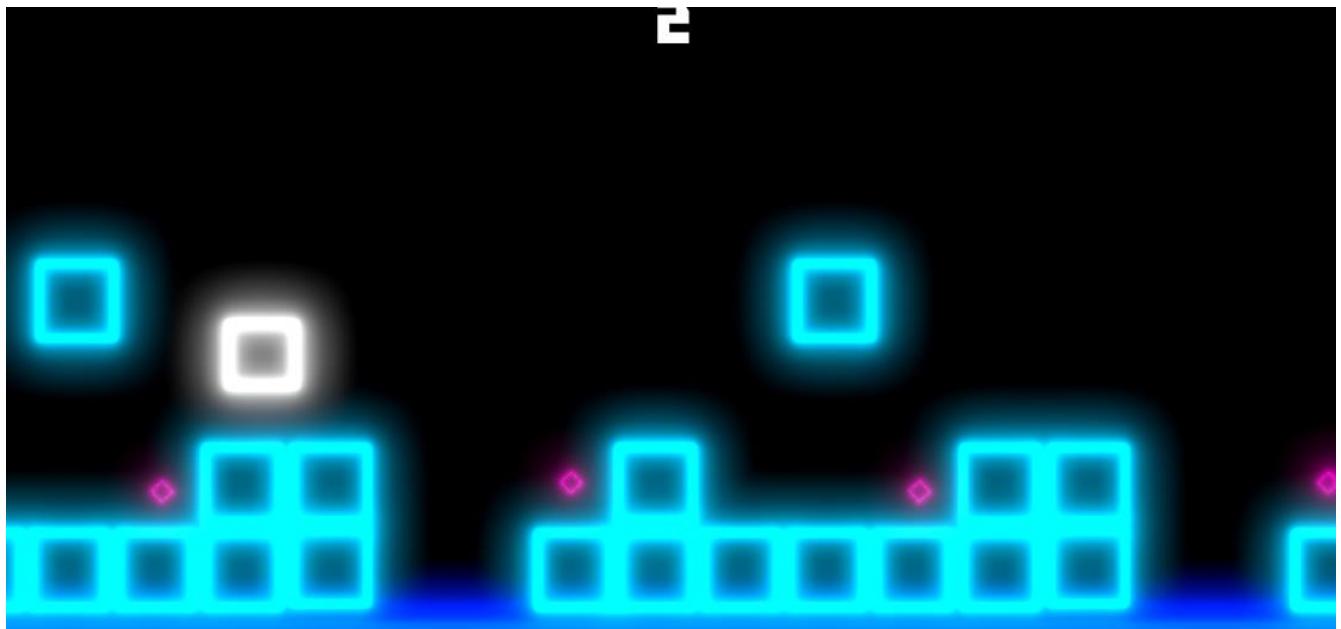
67 Once deleted, click **Add Open Scenes**. This will allow for the game to restart.



68 Play your game and see what you get. Now is the time to change the settings. Maybe you need a more powerful jump? Or maybe the spawning is too quick?

When you are ready, **save** your game and **export** it. Then, **Submit** your game.

Prove Yourself: PolyRun v2



You've built one obstacle out of blocks in PolyRun, but can you build another on your own?

In the PolyRun v2 Prove Yourself, you will do the following:

- 1. Using the same method that you did to build the original obstacle, add another obstacle before or after the original as shown above.**
- 2. You will need to give this object its own new spawn delay and timer or adjust the coordinates of the new obstacle. There are a few ways to do this one, which way will you choose?**

Lighting

Everything that we see in the world is from light that is either produced by or reflected by the objects around us. Without light, there is nothing to see. The same is true with Unity. There needs to be at least one light in the scene, or the camera has nothing to show you. Every new scene that you create has exactly one camera and one light by default. If you disable or delete that light (go ahead, try it), then there is blackness. Needless to say, Unity provides many different options when it comes to lighting, but for now, we will just concentrate on what can be done with the default light.

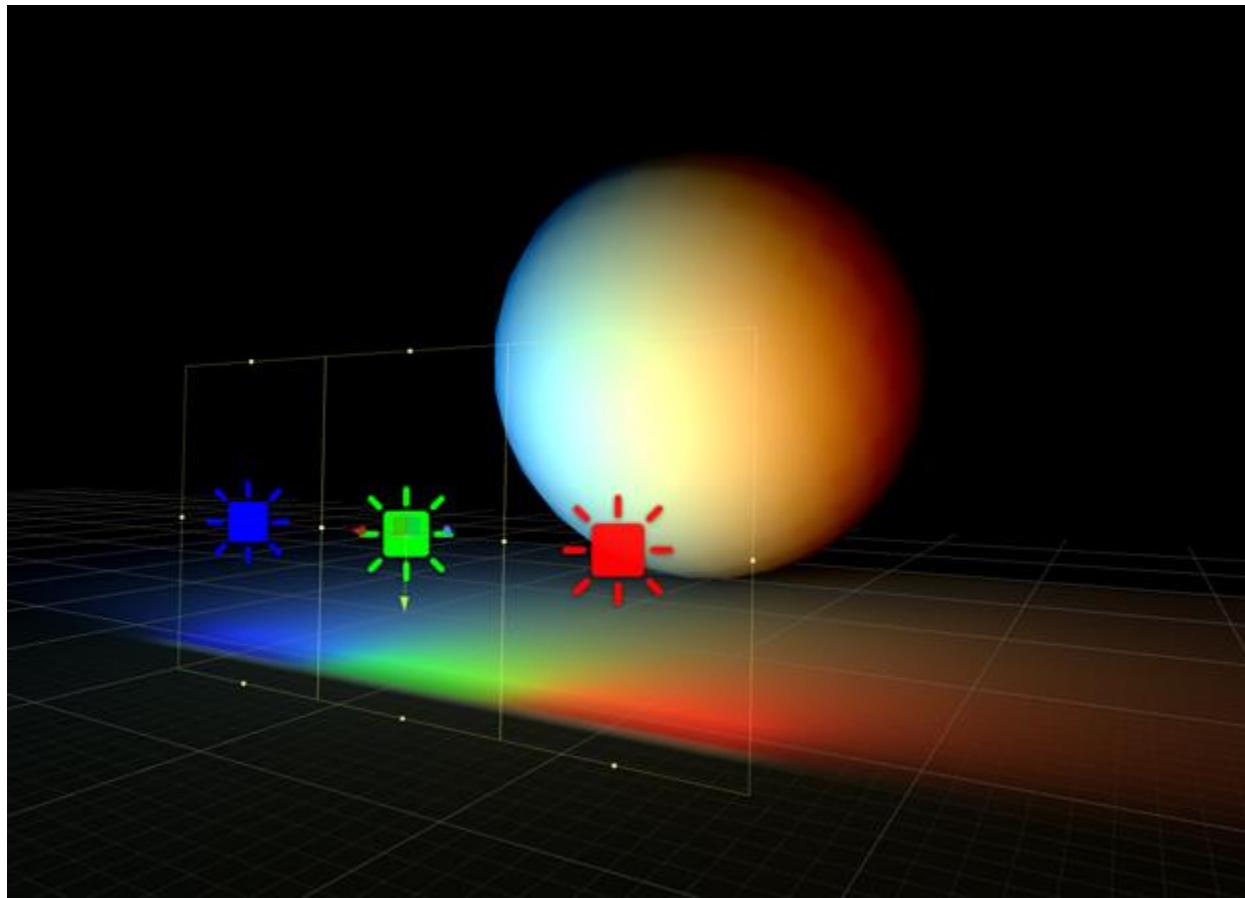
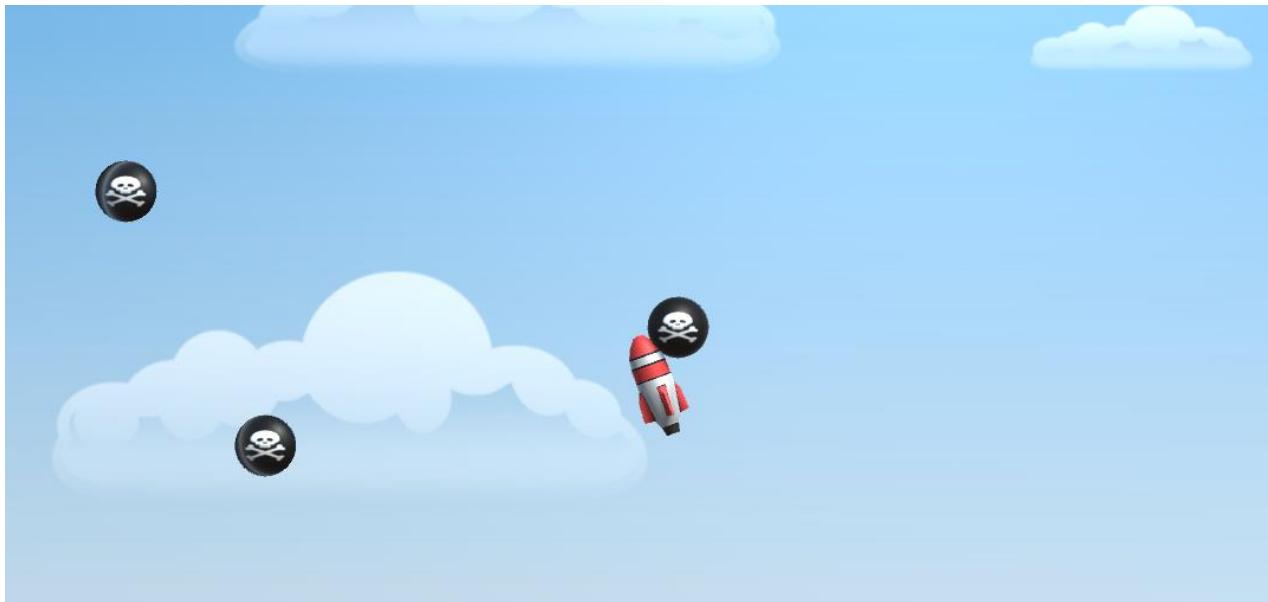


Image Source: <https://learn.unity.com/tutorial/introduction-to-lighting-and-rendering-2019-3>

Advanced Animation and User Interface

Thus far, we have been covering the essential elements of Unity, basically how to get things to do what you want them to do, when you want them to. But that's only part of making your own game. These next few activities will be covering small steps that add style to a game.

A good game needs to be easy to use (even if it's hard to win!). The User Interface is perhaps the most important part of a game. It shows the user what's going on in the game and lets them know what they need to do next.

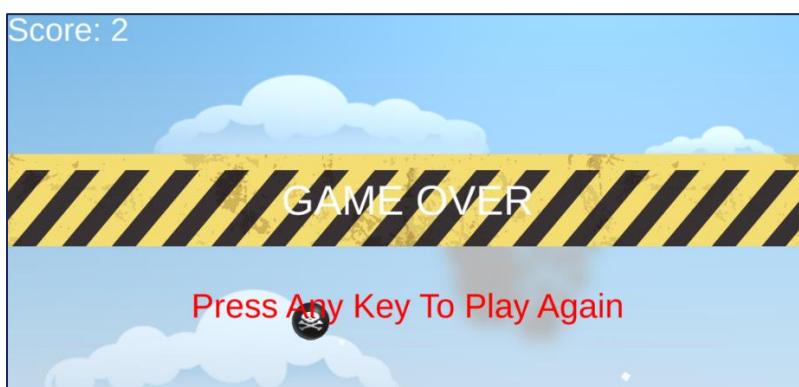
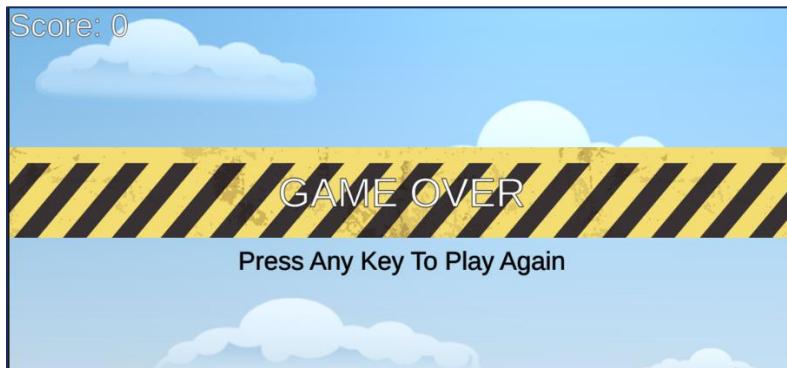


Communicating Through the User Interface

Many of the previous activities have objects that only serve the purpose of giving information to the player. The most basic information is feedback that tells the player how well they're doing, usually with some sort of score text. Other useful information is what to do next - such as "press any key to continue." Playing a game without instructions can be confusing, which isn't very fun.

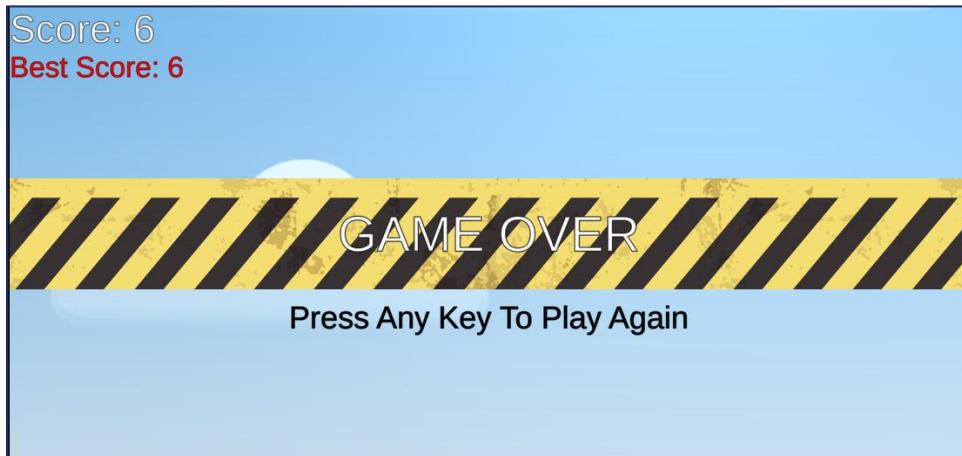
Make It Clear

Elements of the **User Interface (UI)** need to be extremely easy to understand. Information that is difficult to read is not very useful. Use bold text and contrasting colors so that the interface stands out.



In this example, the color is important. Bright colors on bright backgrounds create a clash. We want colors that stand out and don't try to hide within the other colors of our background.

Keep It Simple



Complicated graphics and decorative fonts may give the user interface a special look, but if the user must spend too much time looking for important information, they may spend less time focusing on the game. Make sure the information is specific and timely throughout a game.

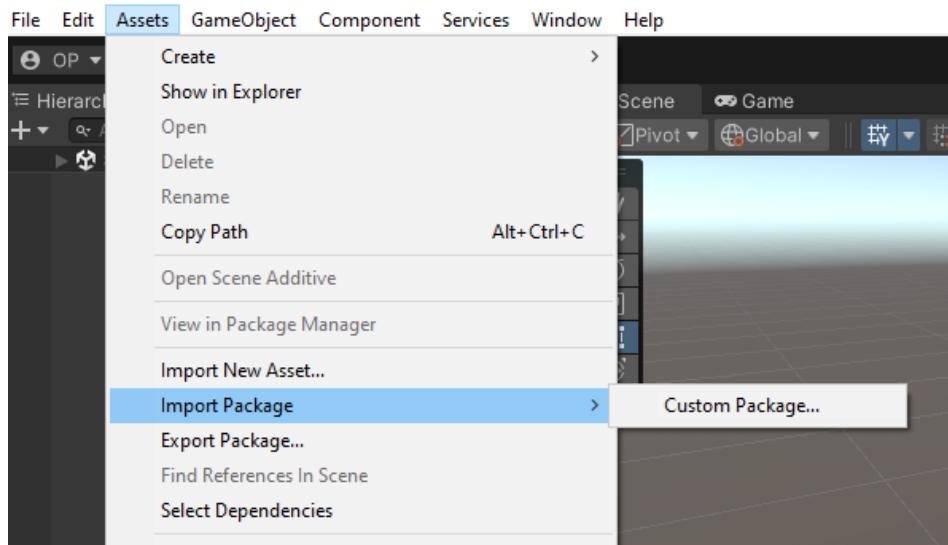
Make It Attractive

Along with UI, you will often hear about **User Experience (UX)**. The **User Experience** is how the user feels about using the product. Your goal is to give them a good experience. UI and UX go hand in hand. A good user interface is the cornerstone of a good user experience. As with the UI, keep it simple, clean, and easy to understand.

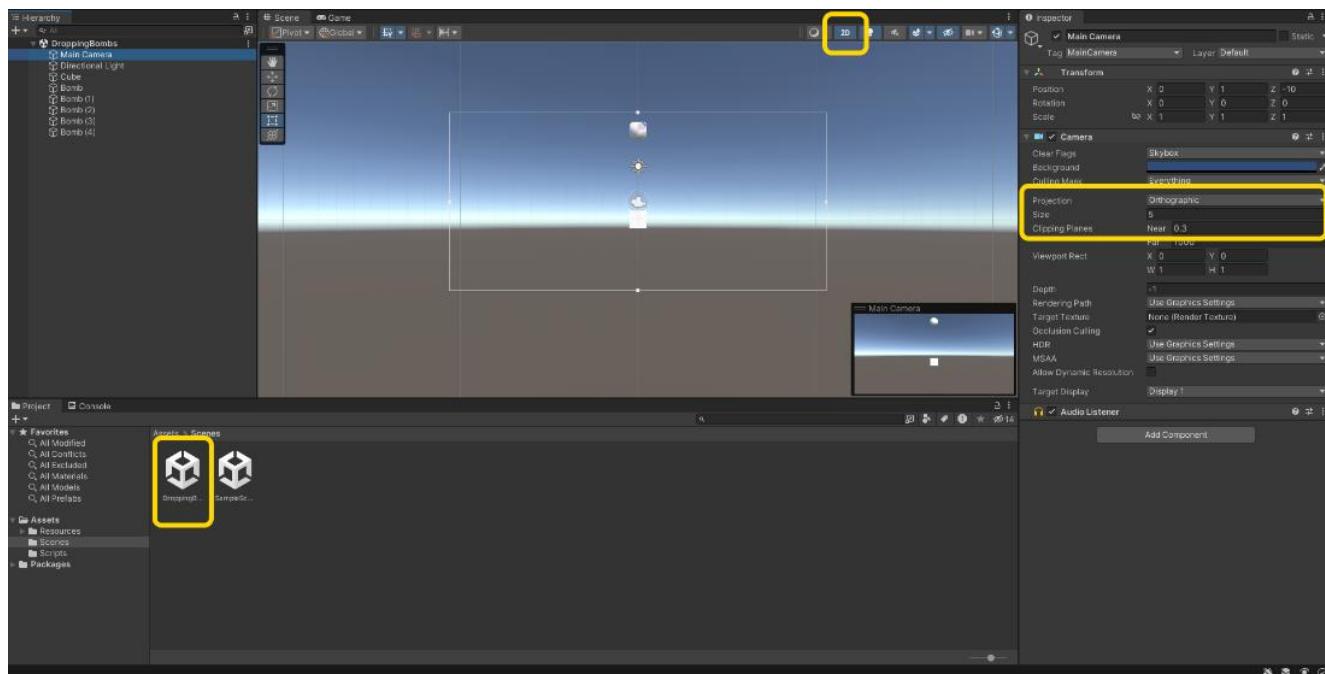
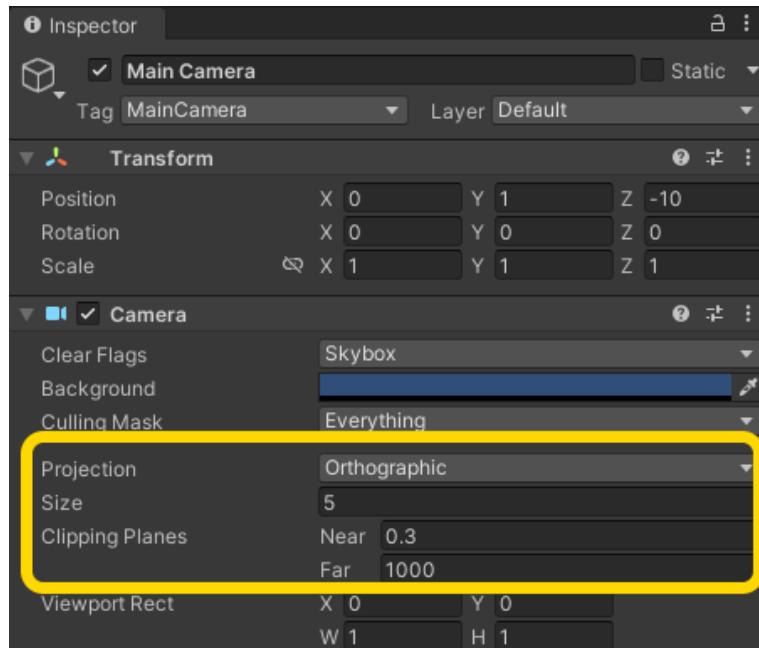
Activity 8: Dropping Bombs Part 2

As promised, this section and the next are devoted to taking the rather simple Dropping Bombs game from the beginning of the book and giving it a good update to make it look like a professional game. You may either start with your files from that activity or open up Unity and create a new 3D project, giving it a name like **JS-DroppingBombs2**.

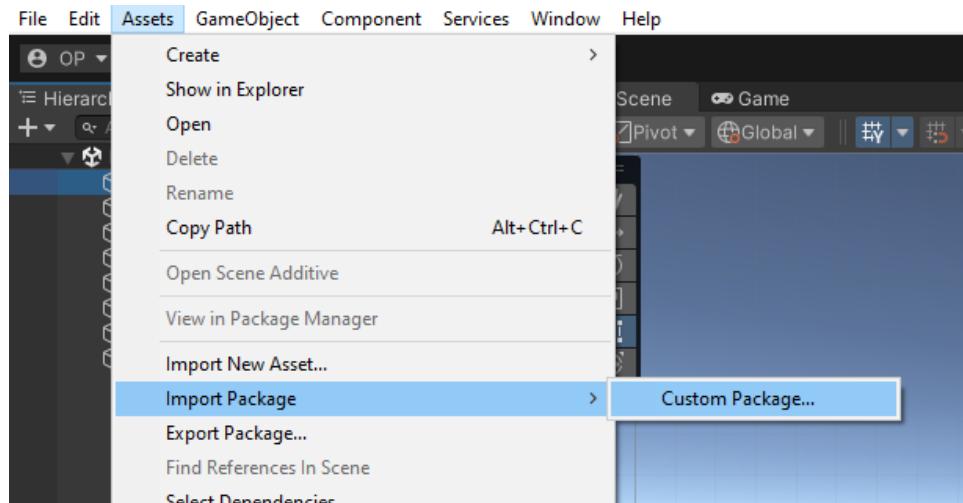
-
- 1 If you are starting with a new project, go to the **Assets** tab, select **Import Package**, and click on **Custom Package**. Go to the folder with all your Purple Belt files and select **Activity 01 – MyDroppingBombsPart1.unitypackage**.



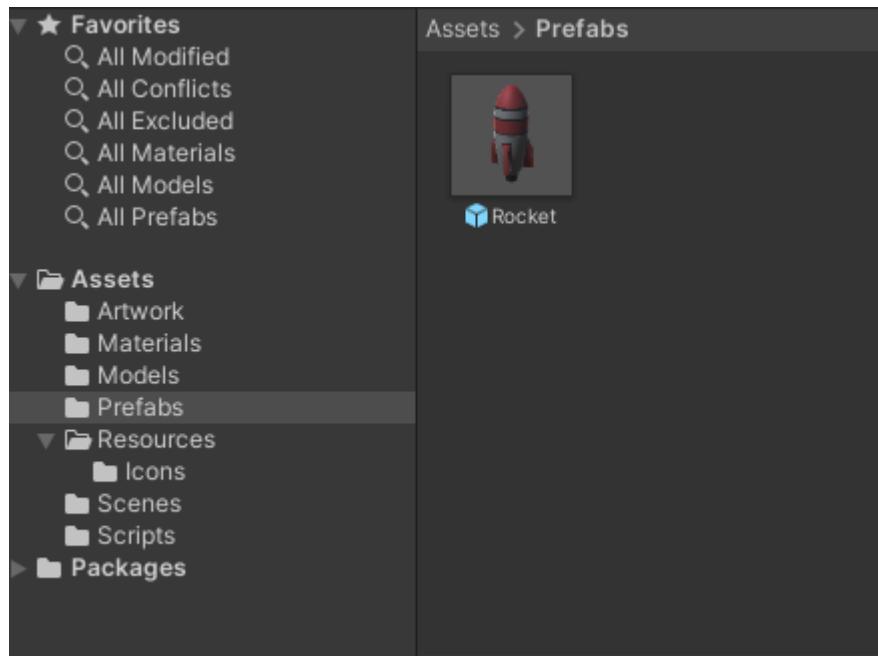
2 If the Hierarchy seems empty, go over to the **Scenes** folder, and select **DroppingBombs** (or whatever your Scene was named). Just as with the original Dropping Bombs game, we will be using the **2D** camera to edit our scene. Select the **Main Camera** GameObject in the **Hierarchy** panel and change the **Projection** from **Perspective** to **Orthographic** as shown below.



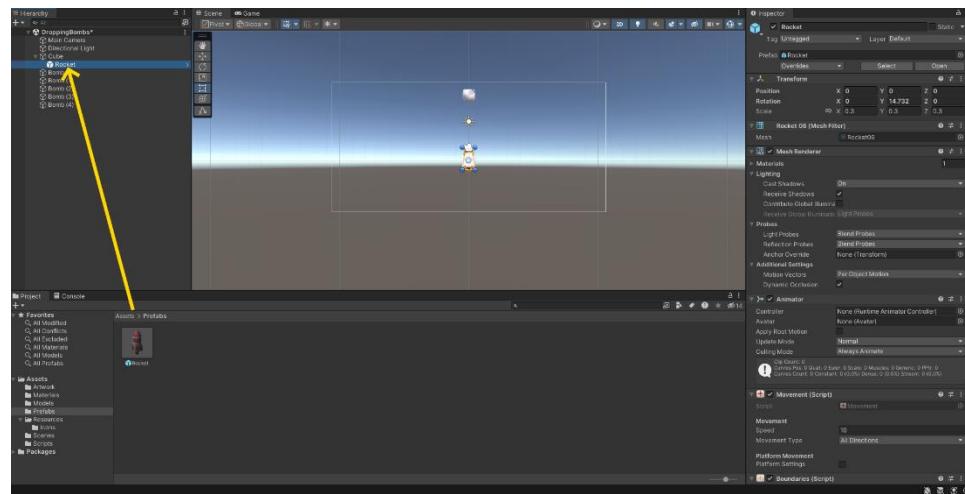
- 3** The first thing you'll want to do is replace the cube and sphere in the game. Go back to the **Assets** tab and select **Import Package**, then select **Custom Package**. This time, select and load **Activity 08 - RedRocket.unitypackage**.



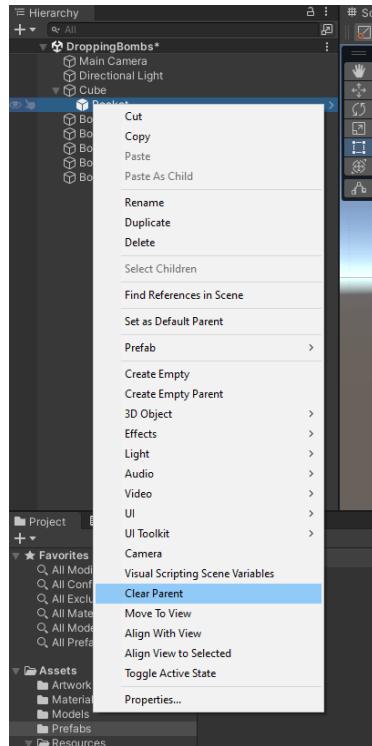
- 4** To verify that the new assets are loaded, go to the **Project** panel and select the **Prefabs** folder to see the **Rocket** assets.



5 To replace the cube with the **Rocket**, hold down the **Alt** key while you're dragging the **Rocket** prefab over to the cube object in the Hierarchy. The rocket will copy all the scripts on the cube. However, we still see the cube, so we will need to remove it.

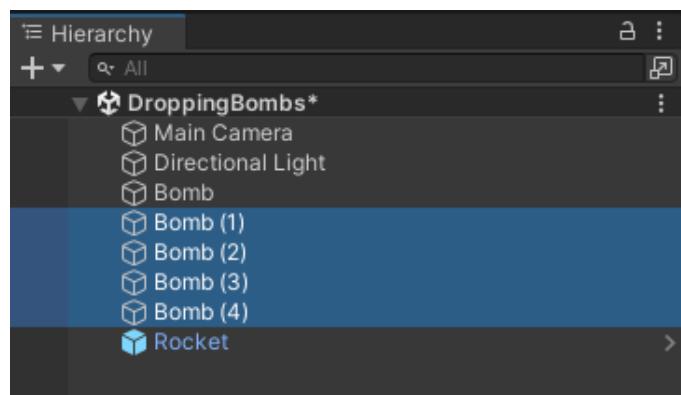


- 6** To remove the old cube, right click on the **Rocket** and select **Clear Parent**. This will remove our rocket from the parent but will still leave the cube. After that, select the **cube** and delete it.



- 7** Just like with the rocket, we'll need to do the same for the bombs.

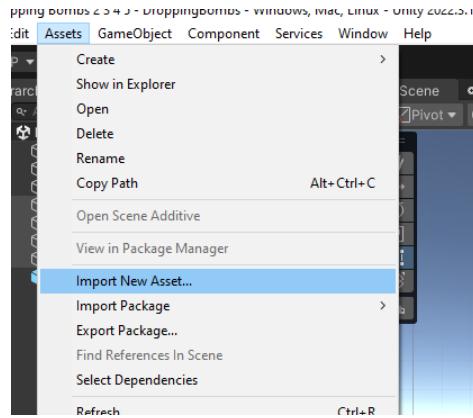
Remove the extra bombs in the **Hierarchy** by highlighting them and deleting them. You should be left with one bomb.



8

Before we can work on our bomb, we'll need to change how they look. We'll start by importing some assets. Instead of importing a Package, we want to import **Assets**.

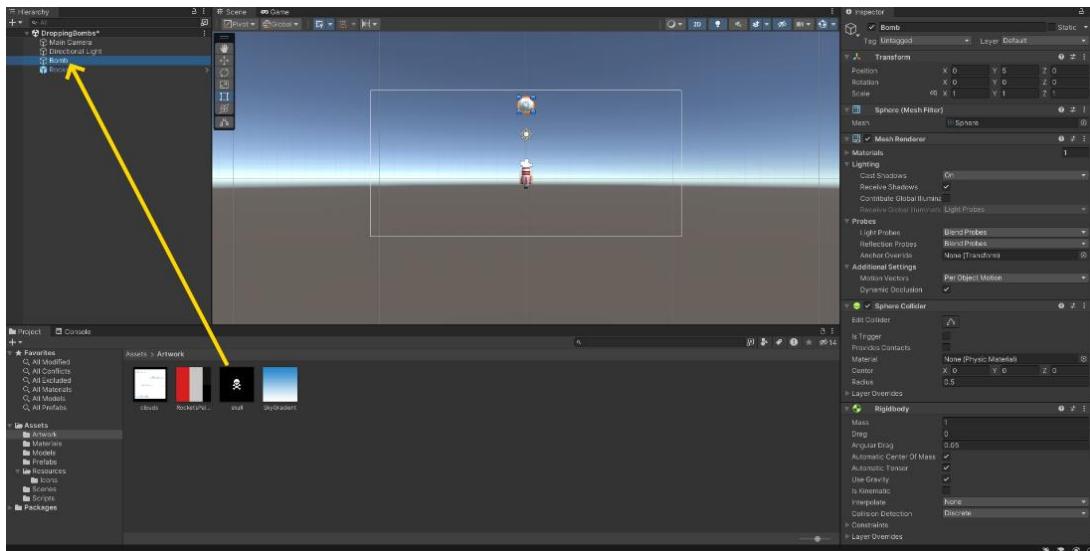
Import the **Activity 08 - Skull.png**, **Activity 08 - SkyGradient.png**, and **Activity 08 - clouds.png** files from the purple belt assets folder into your **Artwork** folder.



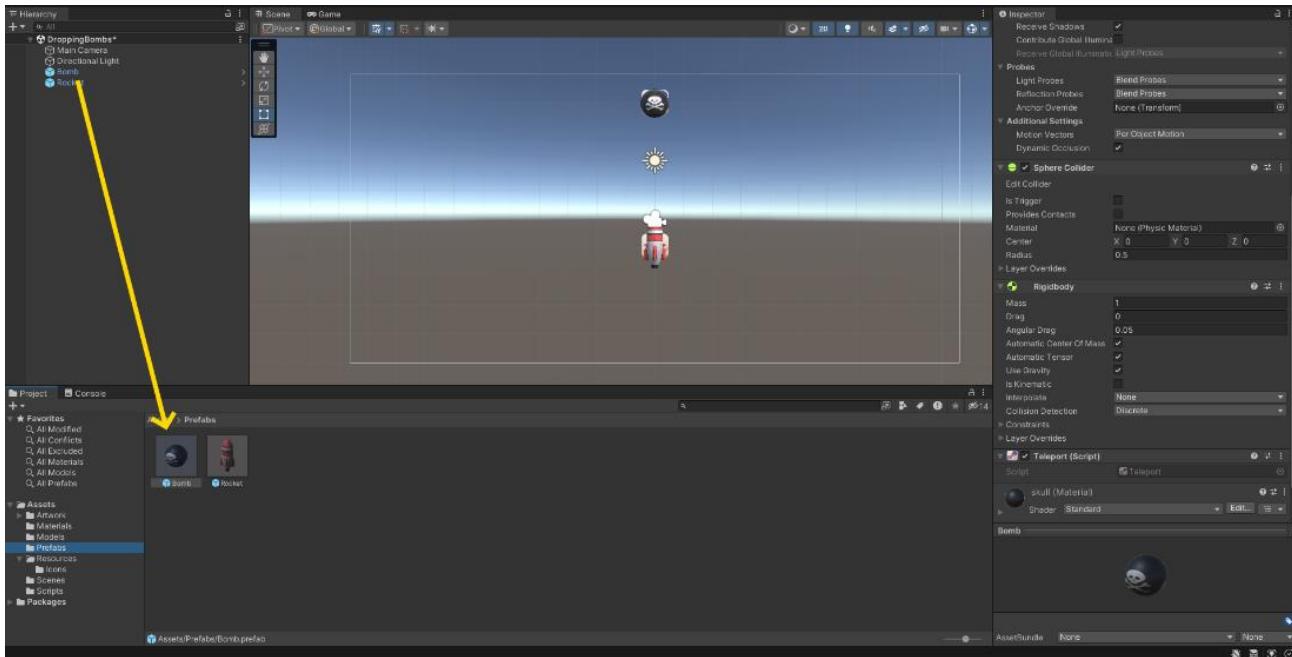
9

With the new imported assets, drag the **Skull** texture from the **Artwork** folder onto the **Bomb** GameObject in the **Hierarchy** to give it a new look. This will automatically create a new **Material** and give it to your bomb.

To make sure you can see the skull, rotate the **GameObject** by 90 on the Y axis.



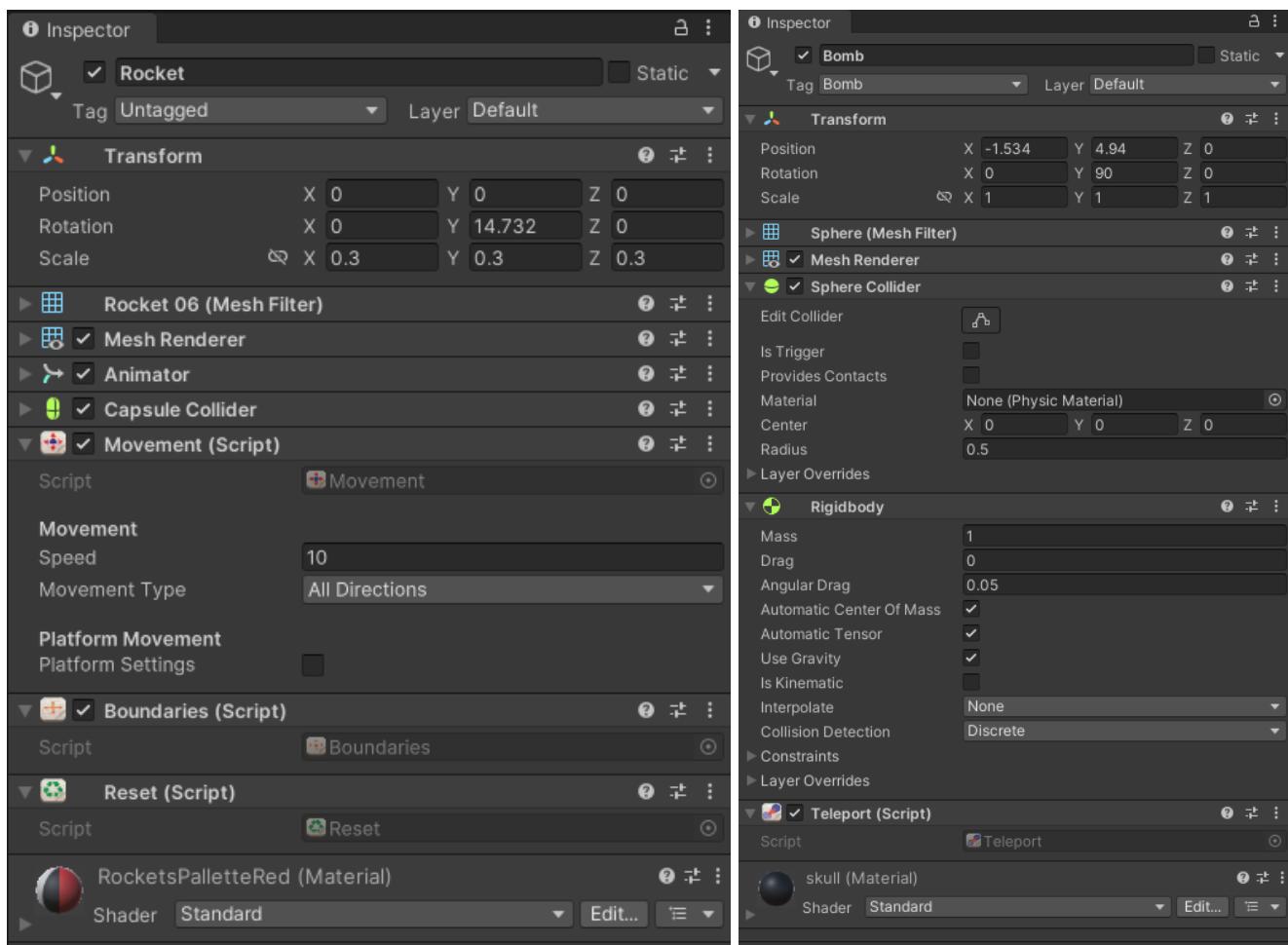
10 Drag the **Bomb Gameobject** from the **Hierarchy** to the **Prefabs** folder, making it into a **Prefab**. Now any changes you make to the **Bomb** prefab will automatically get added to any bombs in the Hierarchy



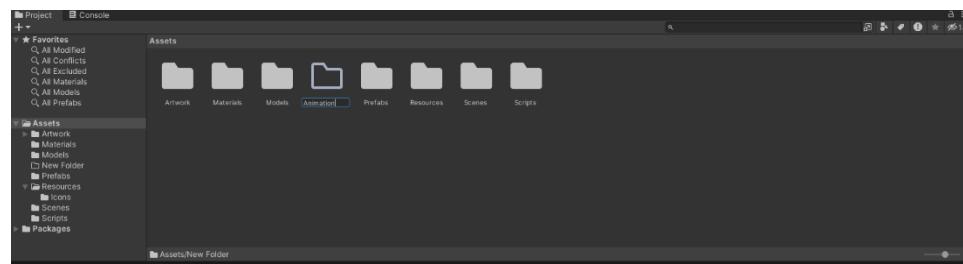
Don't panic if some of the scripts attached to your GameObjects have vanished! We'll put them back in the next step.

11

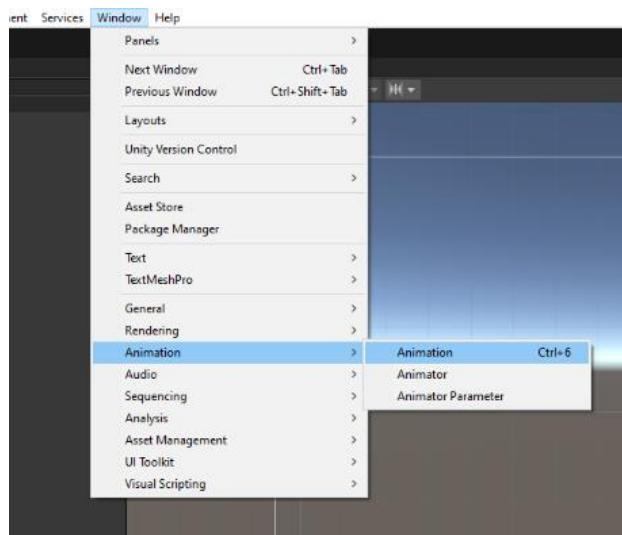
If any scripts get lost while replacing the objects, just drag them back into the **GameObject** Prefabs and use the settings below (The Teleport script goes on the Bomb while the other three go on the Rocket). Once complete, try playing the game!



12 To make this game more interesting, we are going to add some animations to make the rocket more realistic. First, create a folder just for animations. In the **Projects** panel, select the **Assets** folder, right-click to create a new folder and call it "**Animation**".

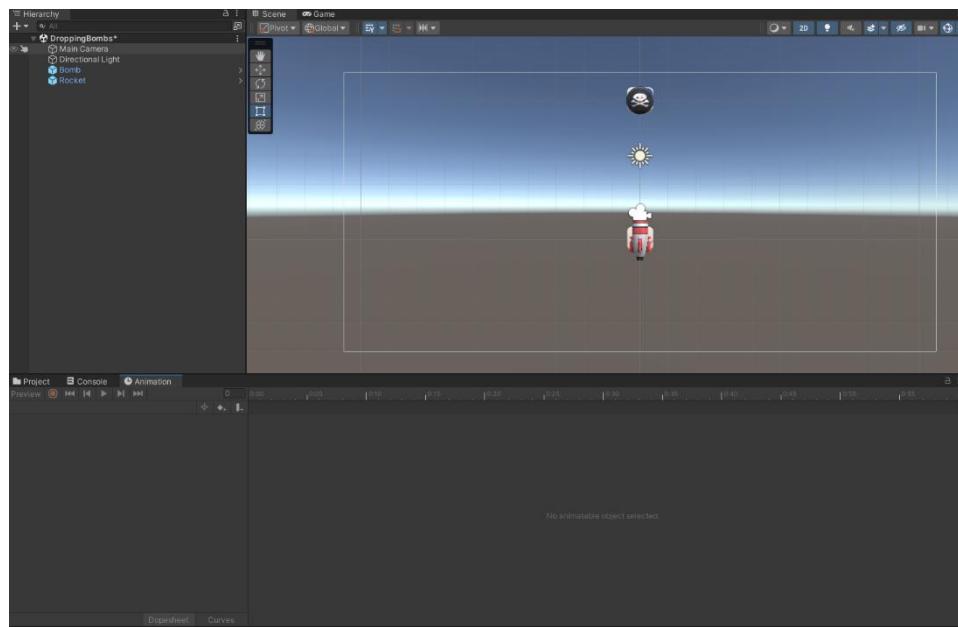
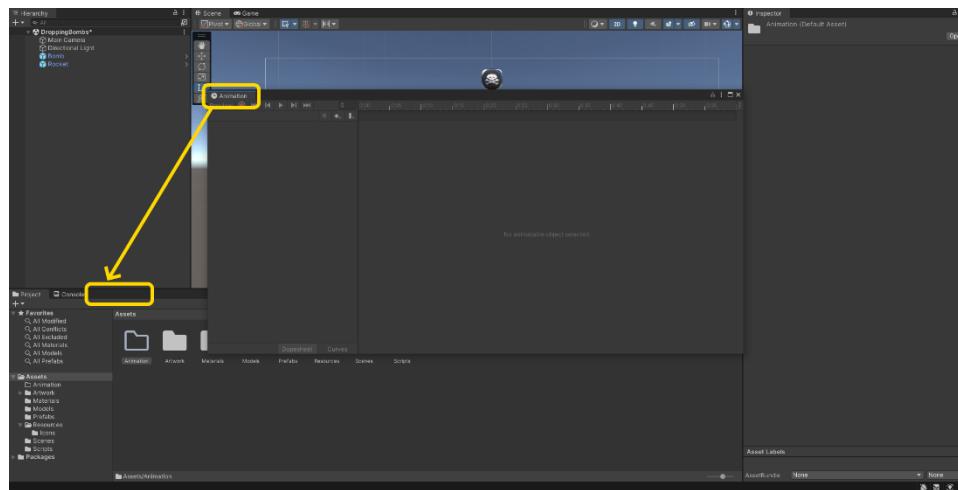


13 To edit animations, we need to open the Animation Window. There are two windows, but for now, we want the one called Animation. Click the **Windows** tab and select **Animation**, then select **Animation** again.

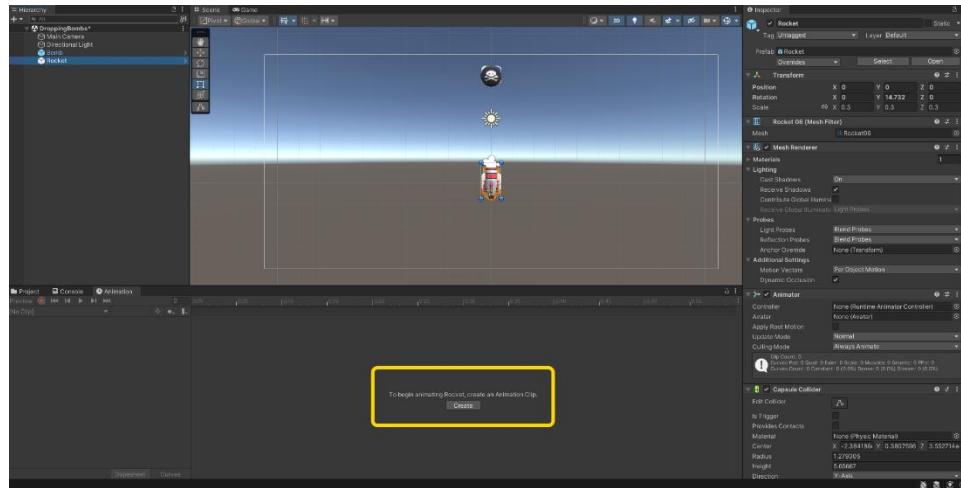


14 This has opened the **Animation** panel. This panel should appear in the middle of everything! If you need some more space, you can move it around or add it to another panel.

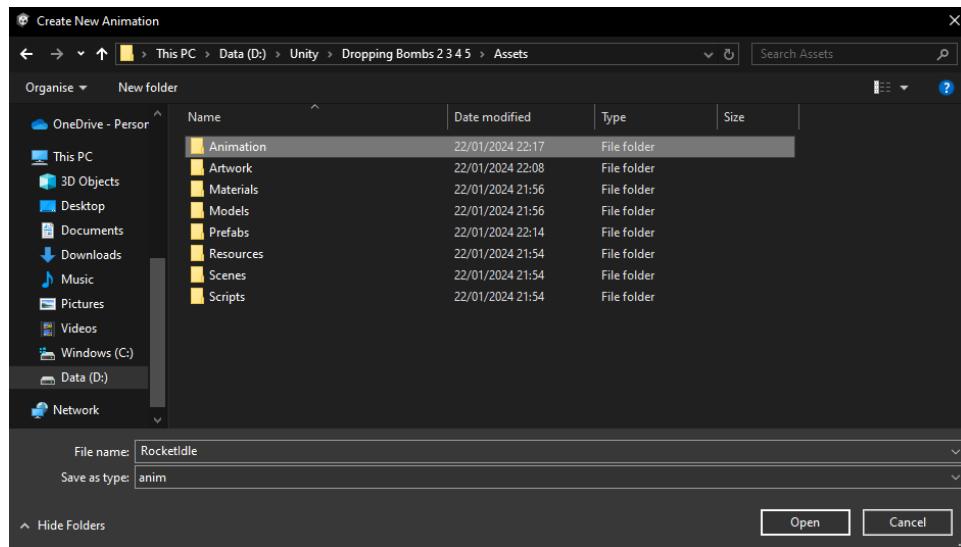
By dragging the **Animation** tab in the **Animation** panel and dragging it around, you can see how it interacts and fits into various places. Find the one that is most comfortable for you. Click the Windows tab and select Animation, then select Animation again.panel since we won't be needing that.



15 Make sure you select the **Rocket** in the **Hierarchy** and click the **Animation** tab. Since you don't have any animations attached to the Rocket, you will be asked to create one. Click the **Create** button.

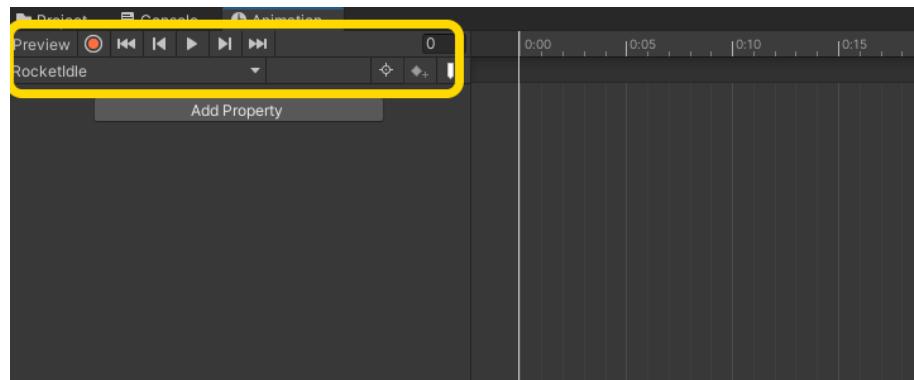


16 This opens the **Create New Animation** window. Open the **Animation** folder that you just made – we'll be putting all the animations in there. Give it a name like **RocketIdle** and click save.

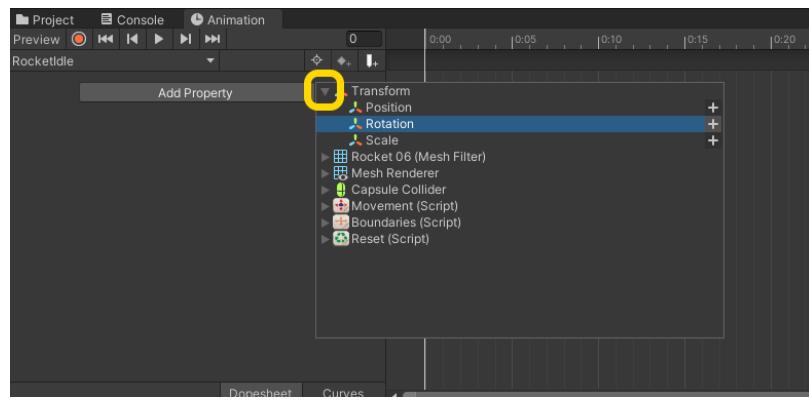


17 Now the **Animation** panel is ready for you to begin creating an animation for your rocket. Look at the control panel in the upper left. These let you record and play your animation to make sure it looks right. It also lets you know what animation you're currently working on and gives you the means to add additional animation to the object.

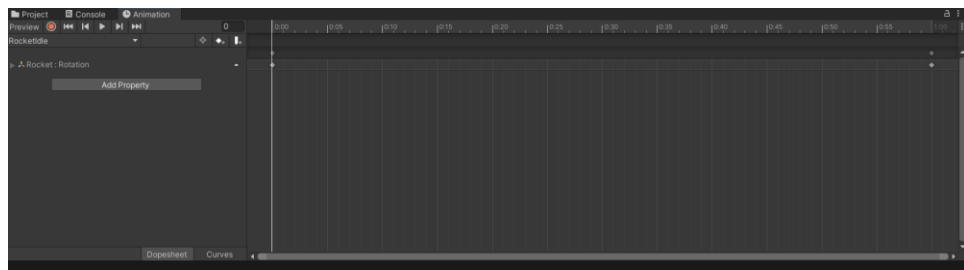
Always make sure you are on the correct animation when editing!



18 In the **Animation** panel, click **Add Property**. This opens a window of all the possible properties in the **GameObject** that you can modify to create an animation. Most Often, animations are applied to an object's **Transform** component (Position, Rotation and/or scale). In this case, we want to edit the **Rotation**, so select that and click the plus symbol (+) on the right to add that property to the panel.



19 This will add two diamonds, these are **Keyframes** to our animation Timeline. One at **0:00** and one at **1:00**. This refers to the amount of time in seconds. So, our Keyframes are happening over 1 second. The Timeline refers to the area in which we will be placing the **Keyframes** and creating the animation.

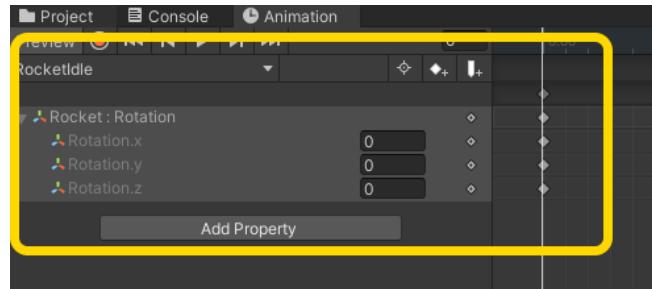


20 Currently, we see one major keyframe for the whole component, but by clicking the little arrow on the left, next to the rotation, we can expand it and see 3 new keyframes one for the X, Y and Z rotation.

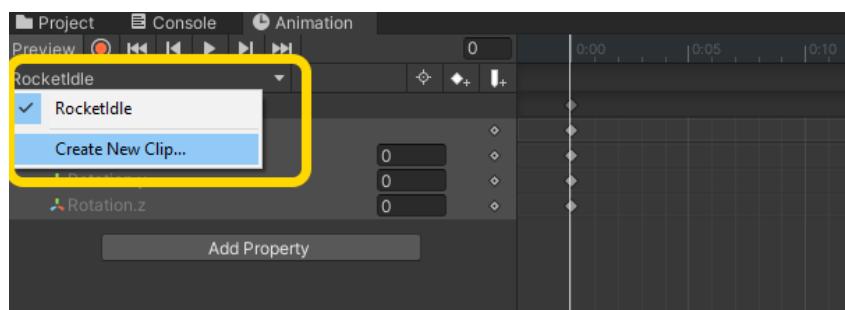
For our first animation, we don't need the keyframes at the 1:00 mark. Select the **top parent keyframe** and press the **delete** key. You will be left with only the keyframes at the 0:00 second mark, meaning this animation only happens for 1 frame.



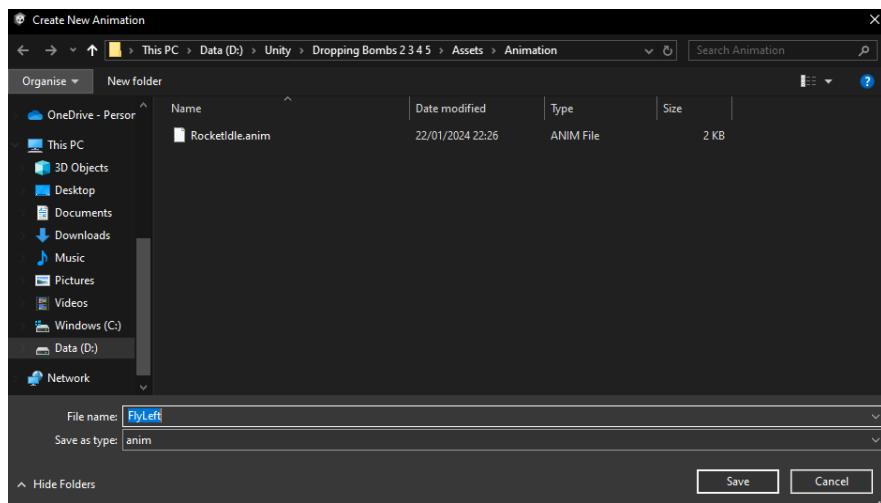
21 That's it for our first animation! The final thing is to make sure the numbers in your keyframes are set to 0, 0, 0. Since this is our idle rocket, it will be set to a rotation of 0, 0, 0.



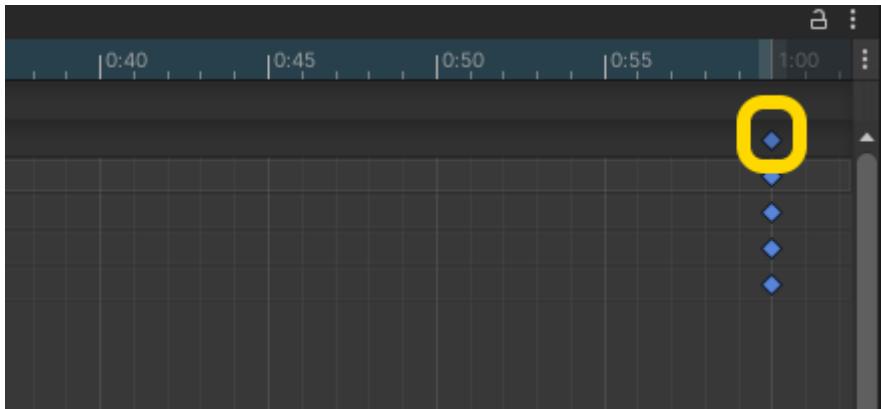
22 Now to create animations for the rocket moving left and right. In the upper left corner of the animation panel, click on the name of the current selected animation. This will open a tiny menu that shows all animations of the current selected GameObject. Select **Create a New Clip**.



23 Make sure that you're in the **Animation** folder and give your new animations a descriptive name, like **FlyLeft**.



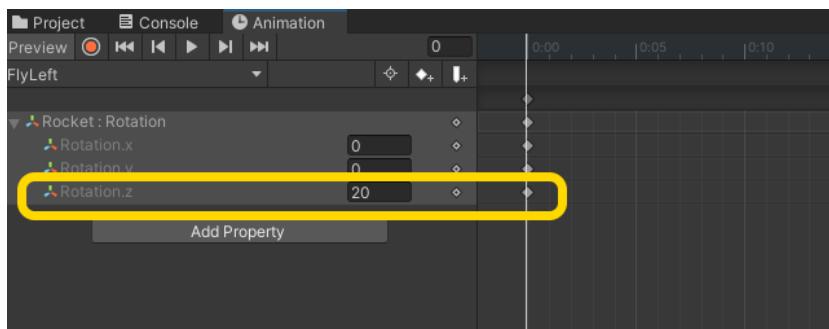
24 Repeat Steps 18 and 19 above to add in the **Rocket's Rotation** component and **delete** the extra Keyframes at the end, as you did with the first animation.



25

Once you've set up your single keyframe for the **FlyLeft** animation, we can start making the rocket fly to the left. To do so, select the **rotation Z** keyframe and type in the number **20**, giving it a rotation of that value.

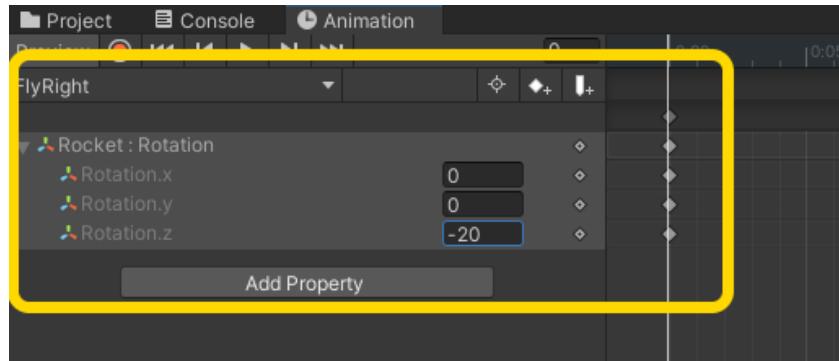
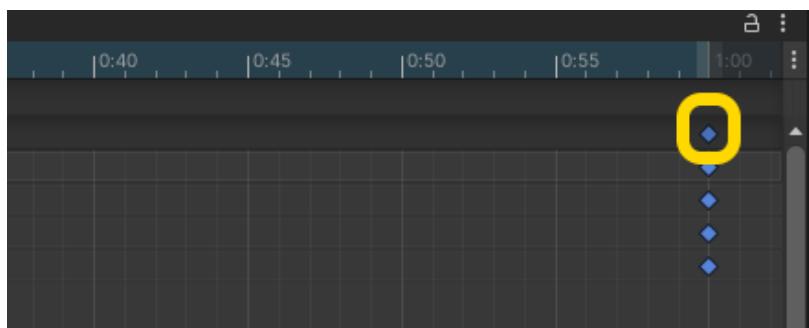
If you don't see the rocket change right away, make sure the Preview button in the top left is selected.



26

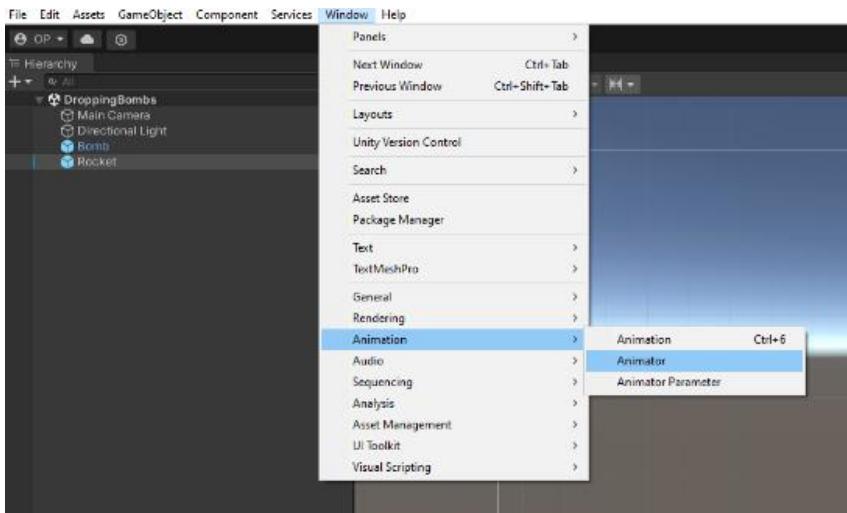
When you are done, we can repeat the above steps but for a third animation called **FlyRight**.

You'll need to **Add a New Clip**, **add the rotation value**, and **remove the extra keyframes** before changing the Z value to **-20**.



27

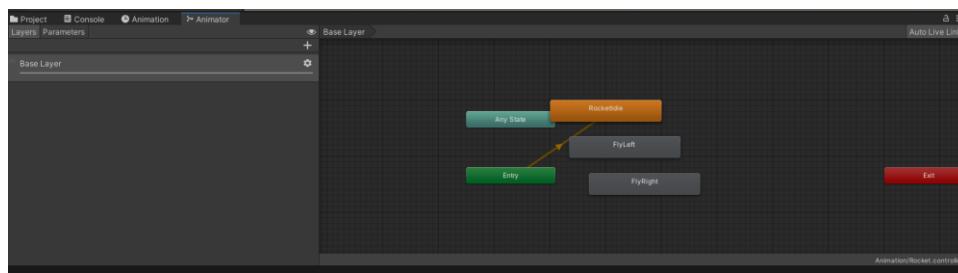
Now we have Three Animations: one for leaning left, one for leaning right, and one for the middle. The next step is to let **Unity** know when each animation needs to be used. For that, we need the **Animator** panel (not to be confused with the **Animation** panel we just used). To do this, click **Window**, select **Animation**, then select **Animator** to load it.



28

Just like with the other panels, you can move the **Animator** panel around until you find a place that feels comfortable. Just like before, placing it down by the project and the animation tab works well.

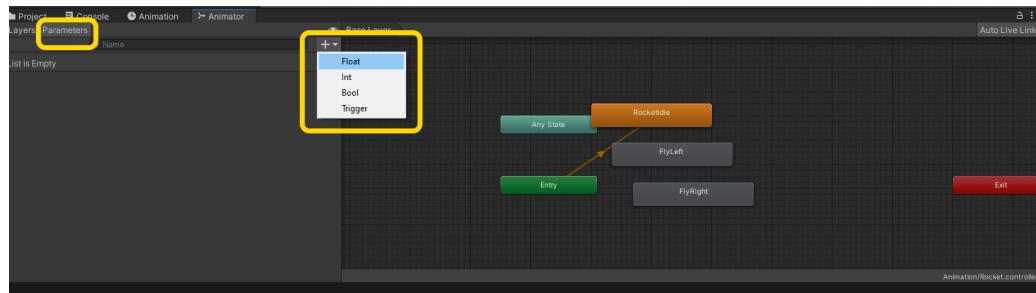
This tab may look confusing, but it's actually very simple. The rectangles are the states of animation that we just made. The green one is the **Entry** state for when the game starts, and the orange one is the default state. Currently, our default state is the one we made first, RocketIdle. This means our first animation will be RocketIdle, when the game loads.



There can only be one default animation state for each object, but any animation in an object can be designated as the default animation state.

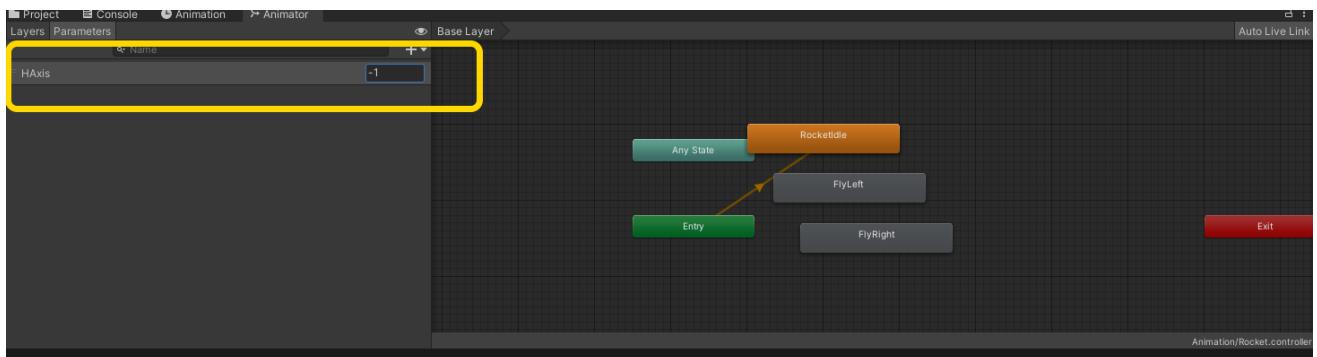
29

We can connect the **Animator** to a script, so the script can tell the animator which animation it should play. For our rocket prefab, all we need is a single parameter. Click the **Parameters** tab and add a new parameter by clicking the plus (+) symbol then **Float**. This Parameter will be a floating-point number (meaning it can be a decimal number).



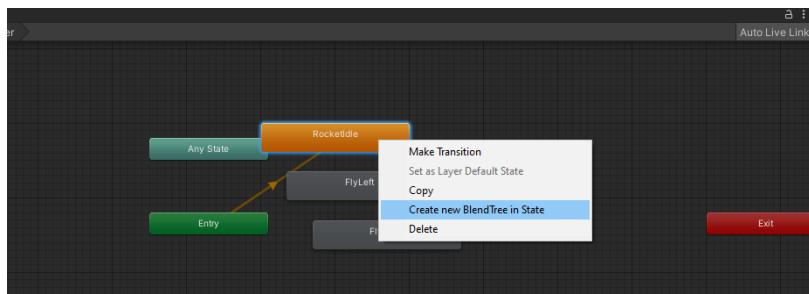
30

After the float appears, rename it **HAxis** (meaning Horizontal Axis). This parameter will tell the **Animator** whether the player is moving to the left or right. Change the value of HAxis to **-1**.

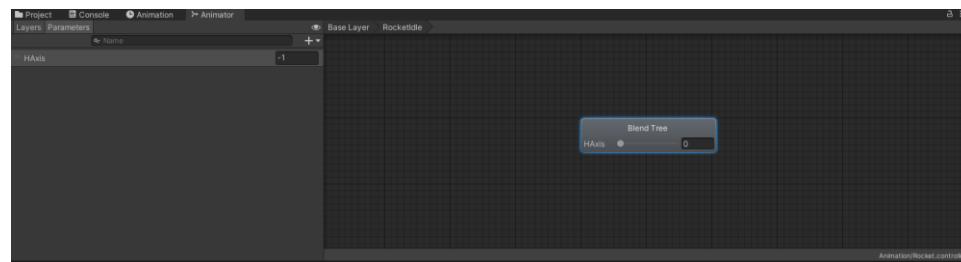


31

You can create links between animation states (like going from Entry to RocketIdle), but there is an easier way. Right-click on the orange default animation (RocketIdle) and select **Create New BlendTree In State**.

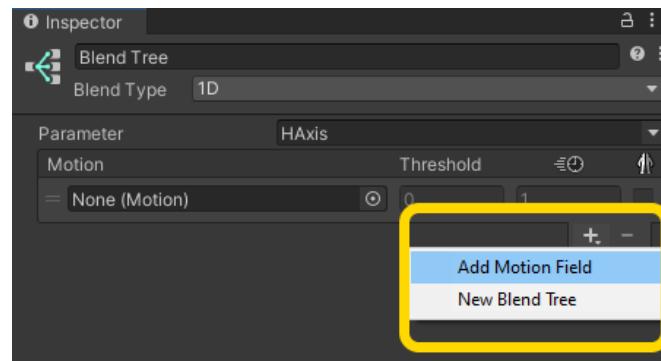
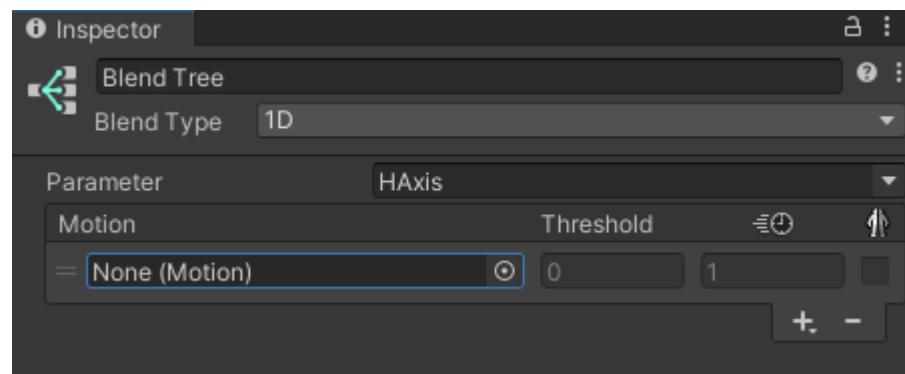
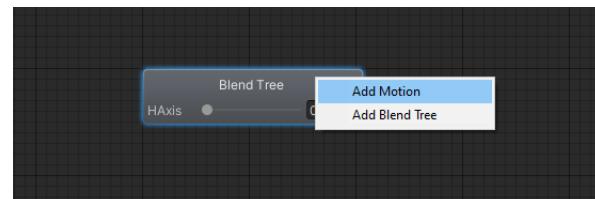


32 It will look as if nothing has happened, but it has! Double-click the default animation (Rocket Idle) to open the **Blend Tree** that you just created.



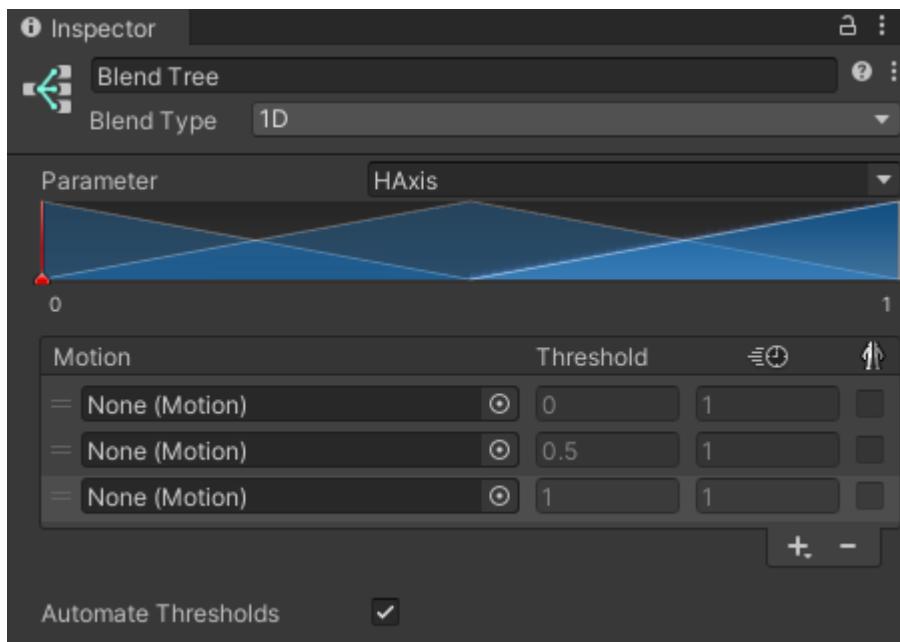
33 Once in the **Blend Tree**, right click the center node and click **Add Motion**.

This will create an option in the top right that will allow us to select our animations, but before that, let's create 2 more.



34

Once we have our 3 Motions, we can click the little dots to select what animation will go in them. The top one should be **FlyLeft**, the middle should be **RocketIdle**, and the bottom should be **FlyRight**.



Using a Blend Tree

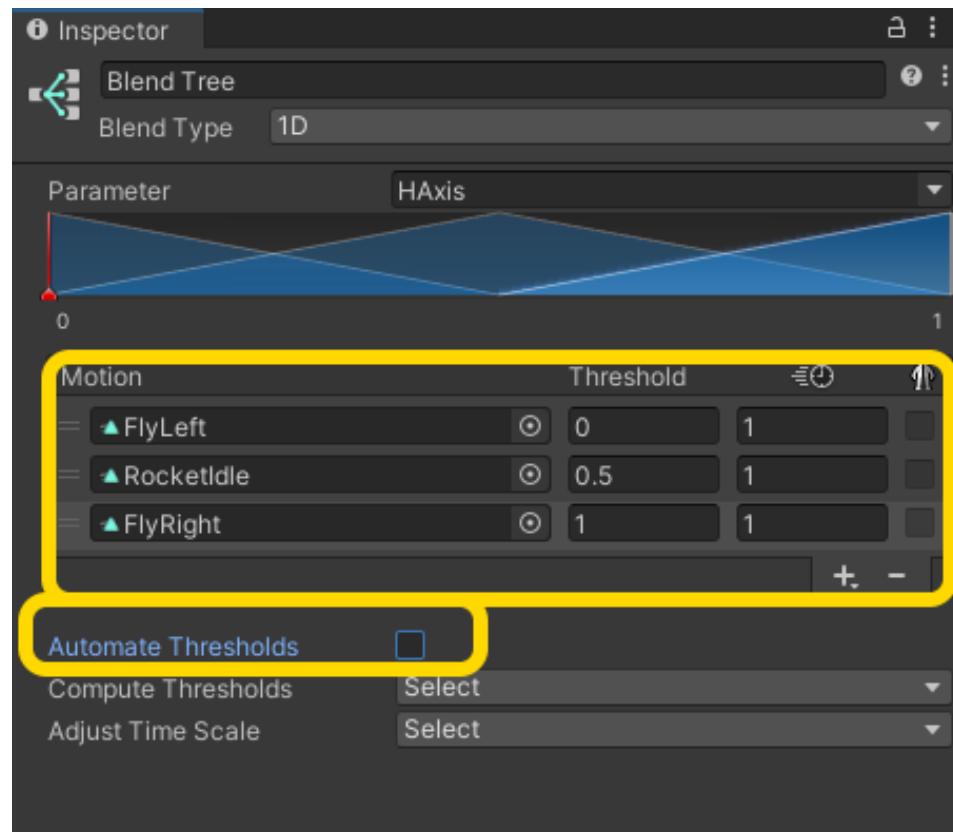
Most animation in Unity can be handled with a simple transition from one animation to the next. For instance, the avatar in Scavenger Hunt has separate falling and landing animations. We don't know how long the avatar will be falling, but when it lands, Unity can smoothly transition from the falling to the landing animation.

When there are more than two animations, handling the transitions can become a bit more complicated. That's where Blend Trees come in handy. With the Rocket, there were three animations. The Blend Tree determines which animation to use based on the data it's receiving – in this case, the data is the player input and the ship animation plays based on Left or Right input.

While our animations may be single frame animations, we can use a Blend Tree to easily transition between them. It can also handle transitions from multiple inputs, such as from moving to the right to moving straight up.

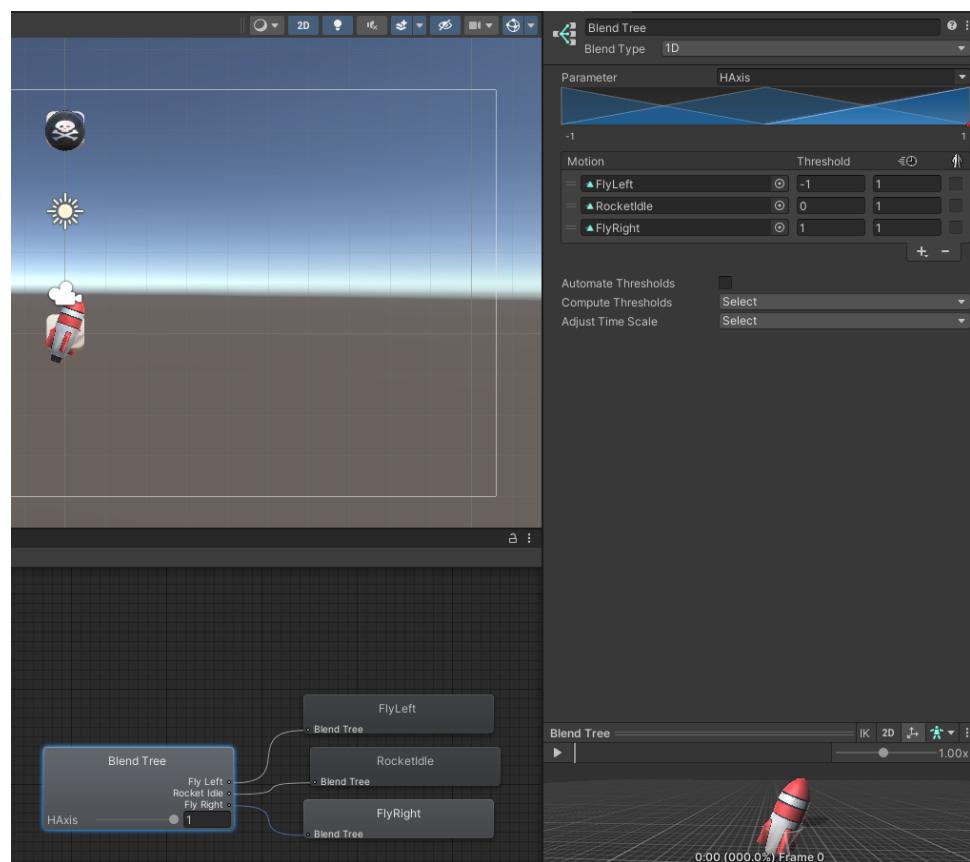
35

Once you've added the three animations, make sure to uncheck **Automate Thresholds** so that we can change the values. The **Thresholds** tell Unity what animation to play based on what number **HAxis** is equal to. FlyLeft should have a **Threshold** of -1, RocketIdle 0.5, and FlyRight 1.



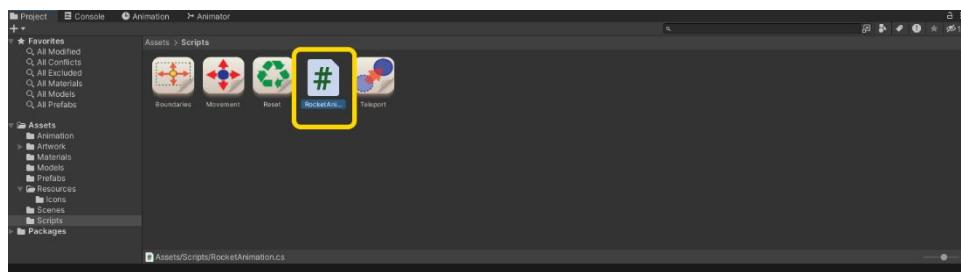
36

Once you've done that. You can preview the animation. Drag the slider in the Blend Tree and see the animation move! Just like that! The blend tree smoothly transitions between animations.



37

With the animations done, we can create a script to allow the **Rocket** and **Animator** to communicate with each other. In the **Project** panel, select the **Scripts** folder and create a new **C#** script. Call it **RocketAnimation**. Double-click on the new script to open the **C#** Editor.



38

Each new **C#** script has some basic code already. We need to add into it to give instructions on what we want the script to do. For these next steps, try to follow along on your own, but if you get stuck, there will be an image to check to see if you got it correct.

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class RocketAnimation : MonoBehaviour
6  {
7      // Start is called before the first frame update
8      void Start()
9      {
10
11
12
13      // Update is called once per frame
14      void Update()
15      {
16
17
18  }
```

39

We want our **variables** to be put before the **void Start** function. In this case, we are creating a **reference** to the animator component. This variable will be **public** so that you can see it in the **inspector**. For this variable, we have given it the type of **Animator**, and the name animator.

40

When the script starts, assign a **component** to the animator variable. In **C#** we use **GetComponent<>()** with **Animator** as the component to retrieve.

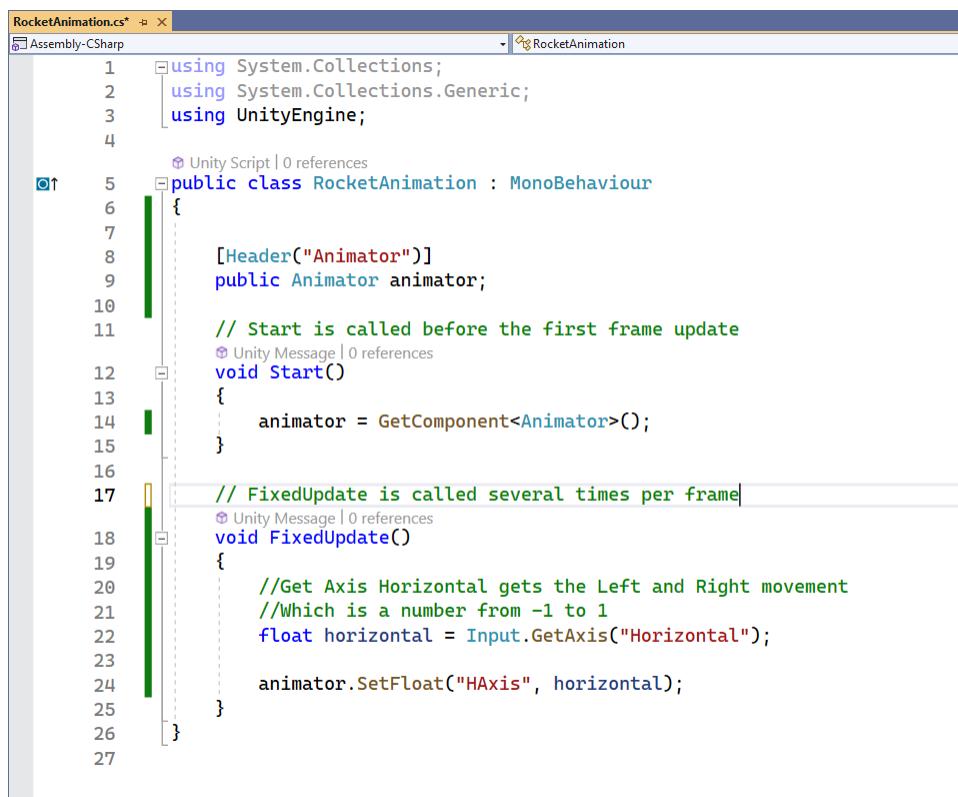
41

We need to change the **Update** function to **FixedUpdate**. While Update is called once per frame, FixedUpdate can be called several times per frame, making it ideal for situations where force or other physics-related functions are applied. In a game like this, it's not necessary, but it can help smooth things out.

Next, we declare a **float** variable named **horizontal** and set it to be equal to **Input.GetAxis("Horizontal")** - in other words, when the user moves left or right, that becomes our floating variable, horizontal.

Finally, tell the **animator** (which we set up above) that the floating variable **HAxis** should be equal to **horizontal**.

Then save the script by hitting **Ctrl+S** and return to Unity.



```
RocketAnimation.cs*  x
Assembly-CSharp
public class RocketAnimation : MonoBehaviour
{
    [Header("Animator")]
    public Animator animator;

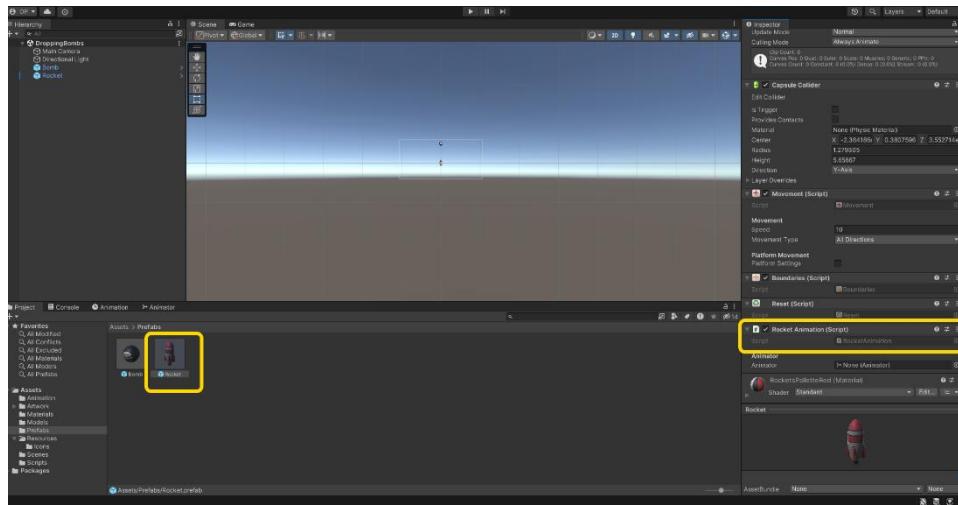
    // Start is called before the first frame update
    void Start()
    {
        animator = GetComponent<Animator>();
    }

    // FixedUpdate is called several times per frame
    void FixedUpdate()
    {
        //Get Axis Horizontal gets the Left and Right movement
        //Which is a number from -1 to 1
        float horizontal = Input.GetAxis("Horizontal");

        animator.SetFloat("HAxis", horizontal);
    }
}
```

42

Once back in Unity, attach the **RocketAnimation** script to the rocket **Prefab**. Since we are using **GetComponent<>()** in our code, we don't need to do anything else. You can enter play mode and see the animation work. You can also see the **Animator** variable we made in the inspector find the animation on the Rocket.



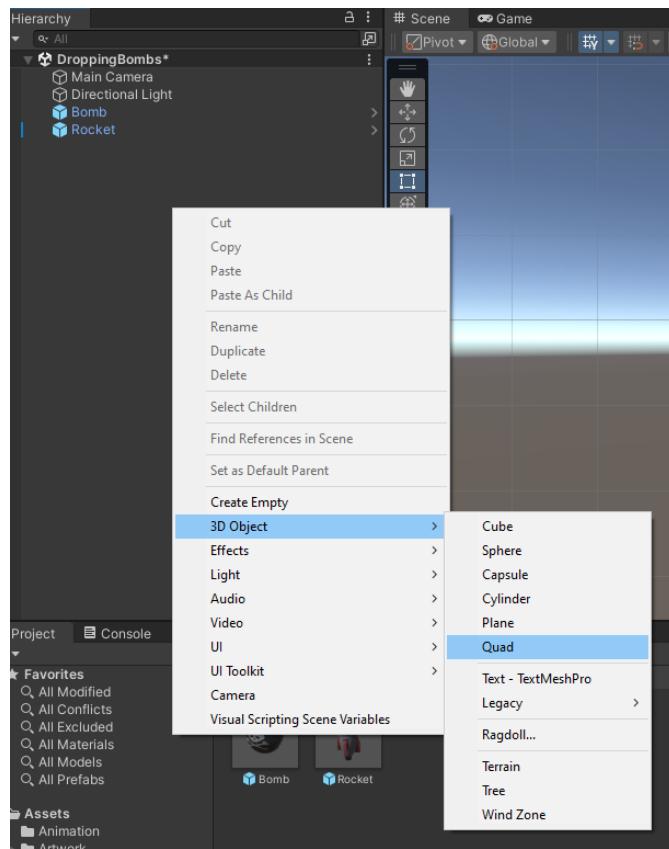
43

Once you've playtested your game, you should notice the rocket leans left and right as it flies. There's still more we can do to make it feel like it's flying.

When you are finished testing, stop the game by clicking the **Play** arrow again.

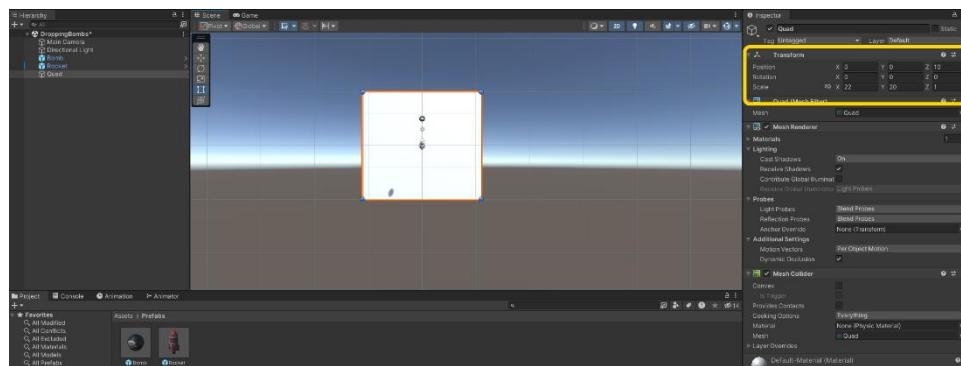
44

Where do rockets fly? In the sky, of course! Before we can add a sky, we need a place to put it. Right-click on the **Hierarchy** and select **3D Object**, then select **Quad**.



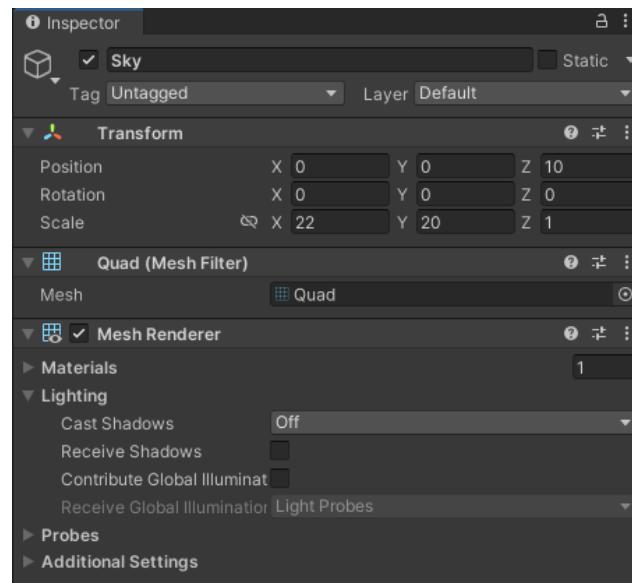
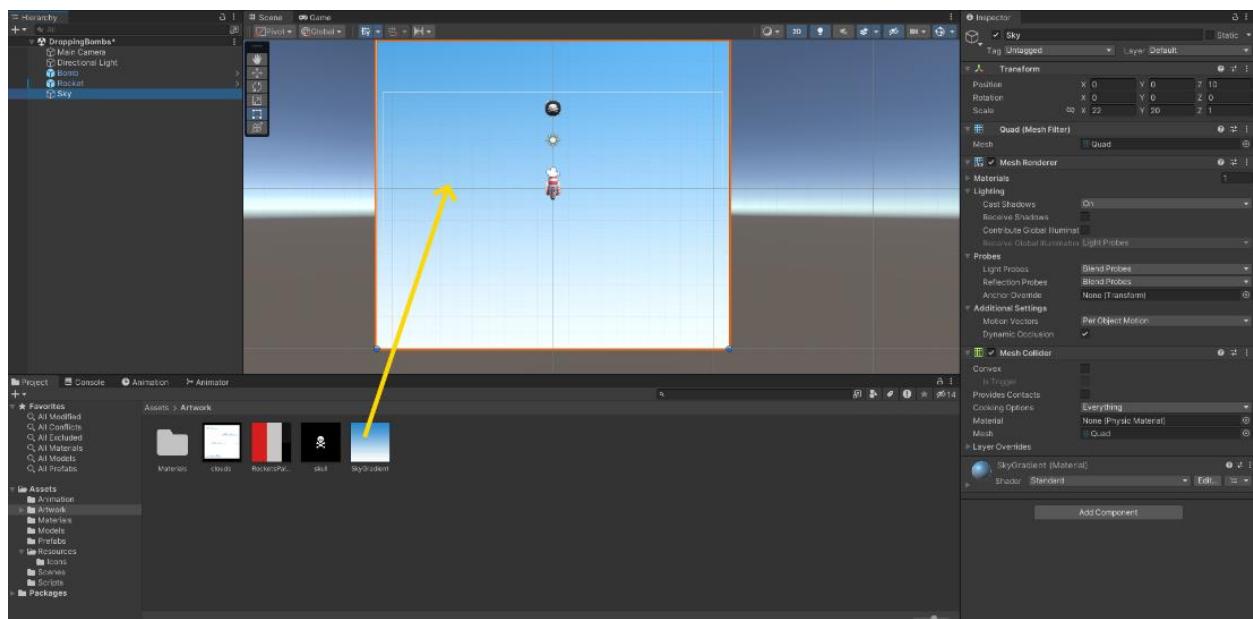
45

Make sure the new **Quad** is in the middle of the scene at X:0, Y:0, and set the Z:10. Resize the Quad so it fits the white outline of the camera from as seen in the image below.



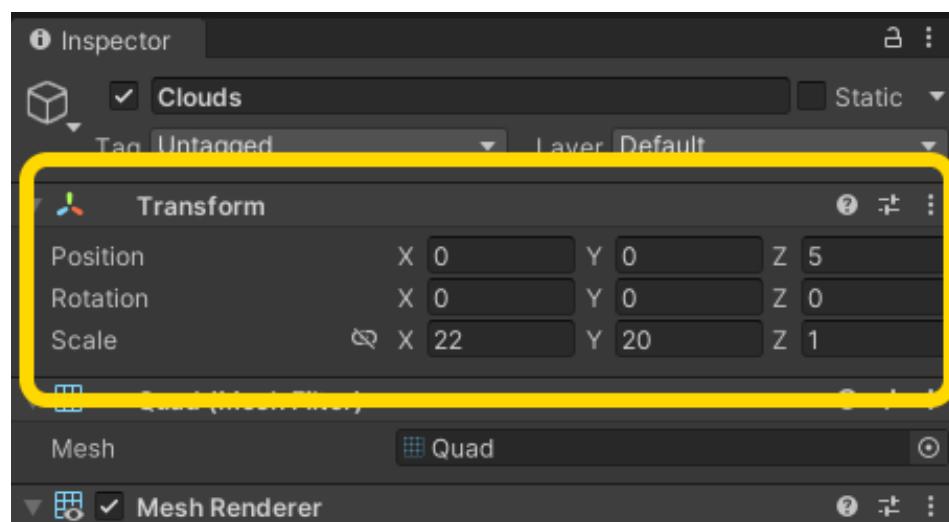
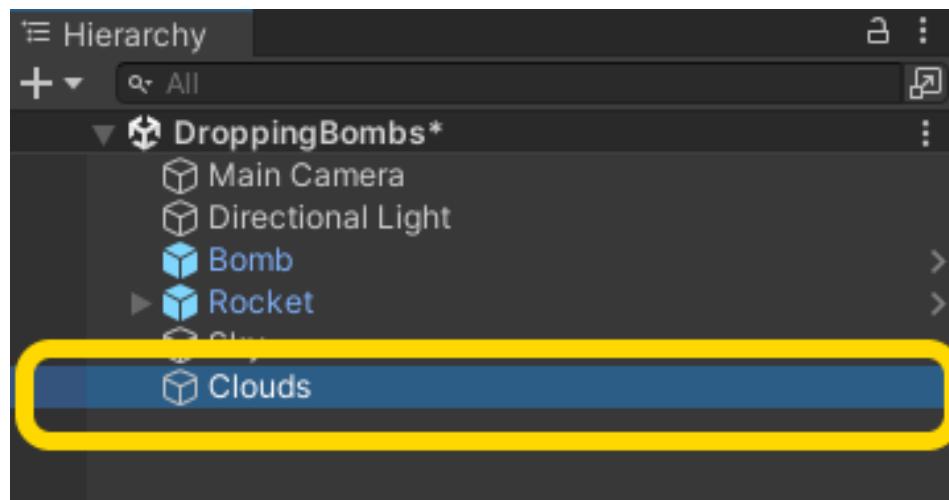
46

Rename the **Quad** to **Sky**. Drag the **SkyGradient** texture from the **Artwork** folder onto the **Sky GameObject**. In the **Mesh Renderer** component, change **Cast Shadows** to **Off** and make sure that **Receive Shadows** in the Mesh Renderer is unchecked.



47

Press **Ctrl+D** to duplicate the **Sky GameObject** and rename the new object **Clouds**. We want the clouds in front of the sky, so change the Z value of **Clouds** to 5.



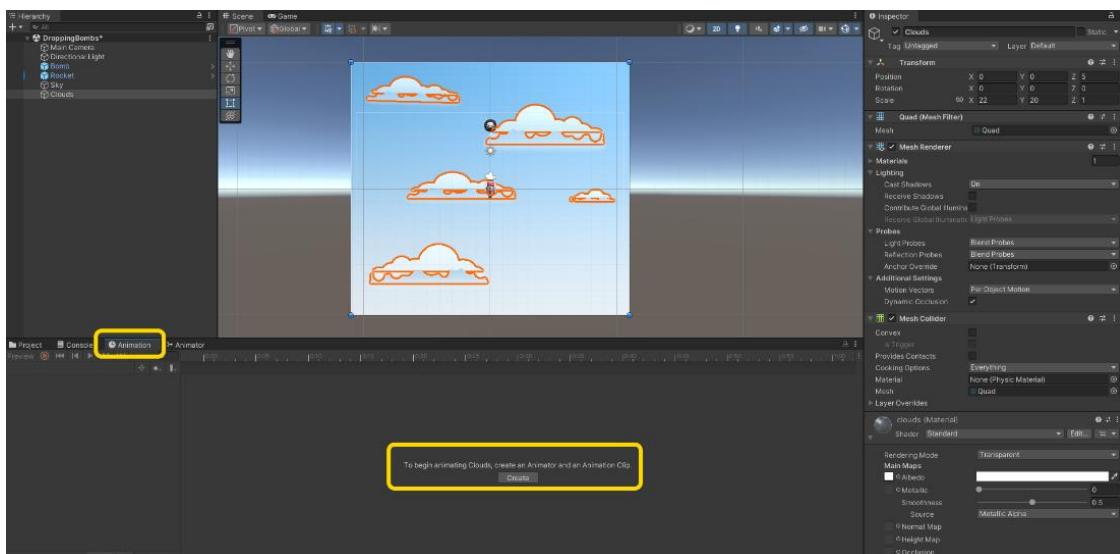
48

Drag the **clouds** texture from the **Artwork** folder onto the **Clouds GameObject**. In the **Inspector** panel, find the **Mesh Renderer** component and set **Cast Shadows** to **Off**. Then, go to the **clouds Shader** and change the **Rendering Mode** to **Transparent**. If you can't see this option, click the arrow on the material to reveal all the material options.



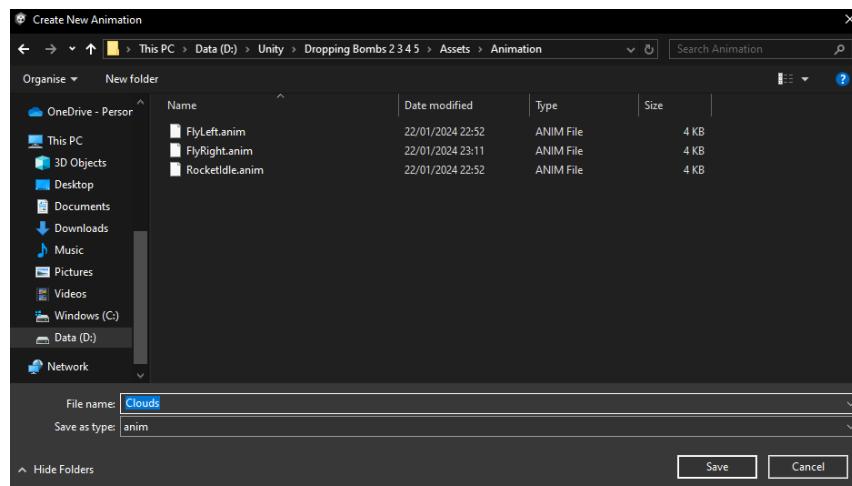
49

With the **Clouds GameObject** still selected, click the **Animation** tab and select **Create** to make an animation for the clouds.



50

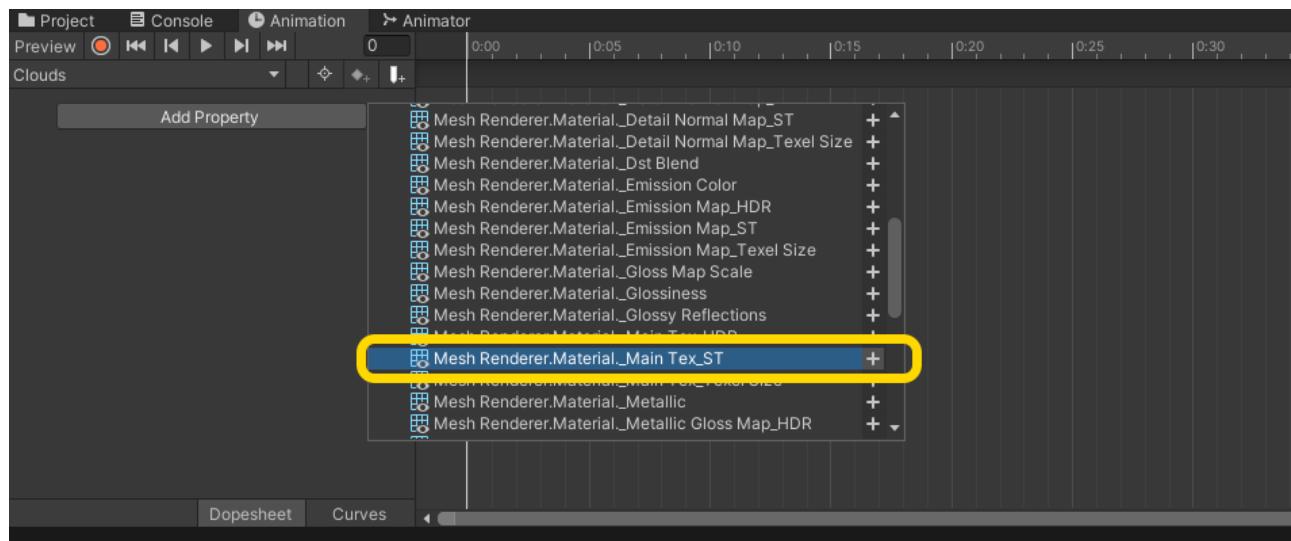
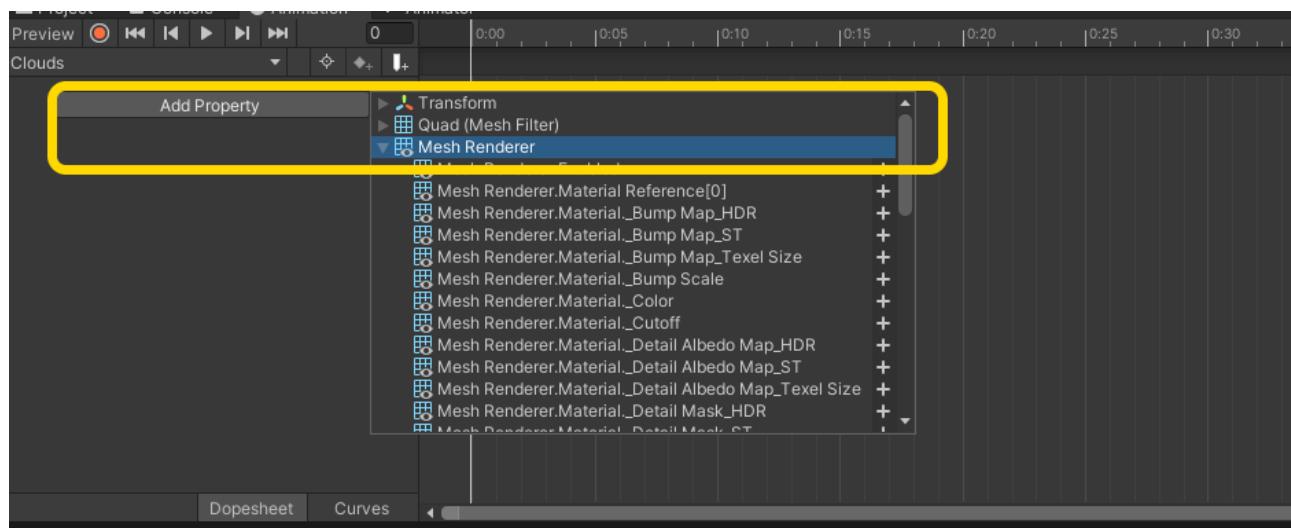
Make sure that the new animation is in the **Animation** folder and name it **Clouds**.



51

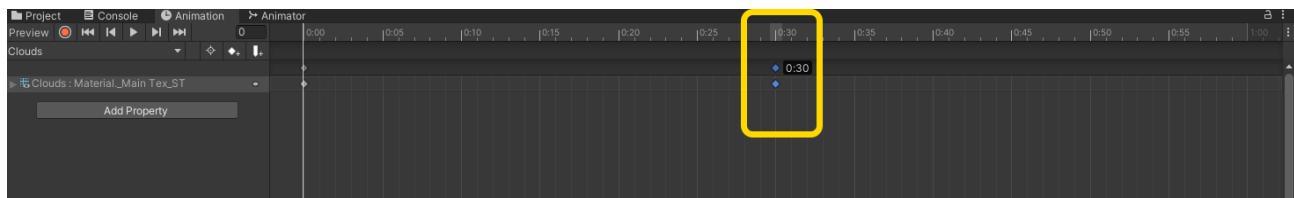
Just as you did with the **Rocket** animation, click on **Add Property**. This time, you'll animate the cloud texture which is part of the **Mesh Renderer**.

Expanding the Mesh Renderer shows other properties that can be animated. We want **Material_Main_Tex_ST**. Click the plus (+) symbol to add it to the **Animation** panel.



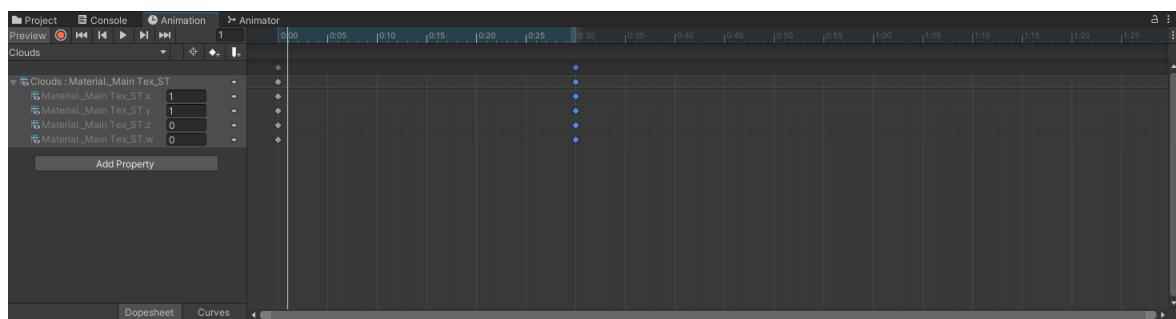
52

As with the **Rocket** animations, the length of the animation is 1 second long by default. Let's make it half that length. Click the top key frame diamond at the 1 second position and drag it over to the 0:30 second position as shown.

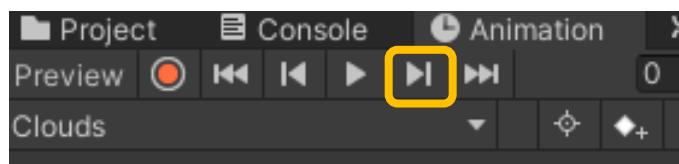


53

The animation timeline has a vertical white line (known as the **Playback Head**) at the 0 position. This line indicates where in the timeline that you're making your changes. For the clouds, we want to make our change at the very end of the timeline, at the 0:30 second position. Drag the white line over to that end of the timeline.

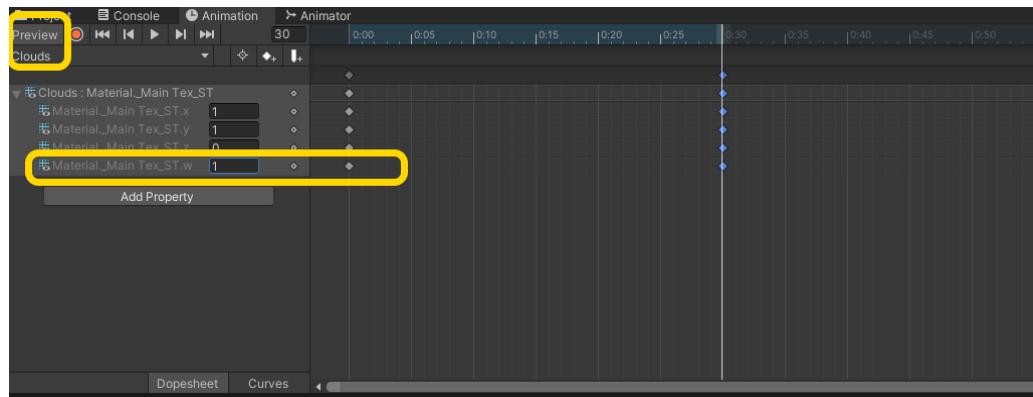


You can also use the preview controls to send the playback head to the previous or next keyframe.



54

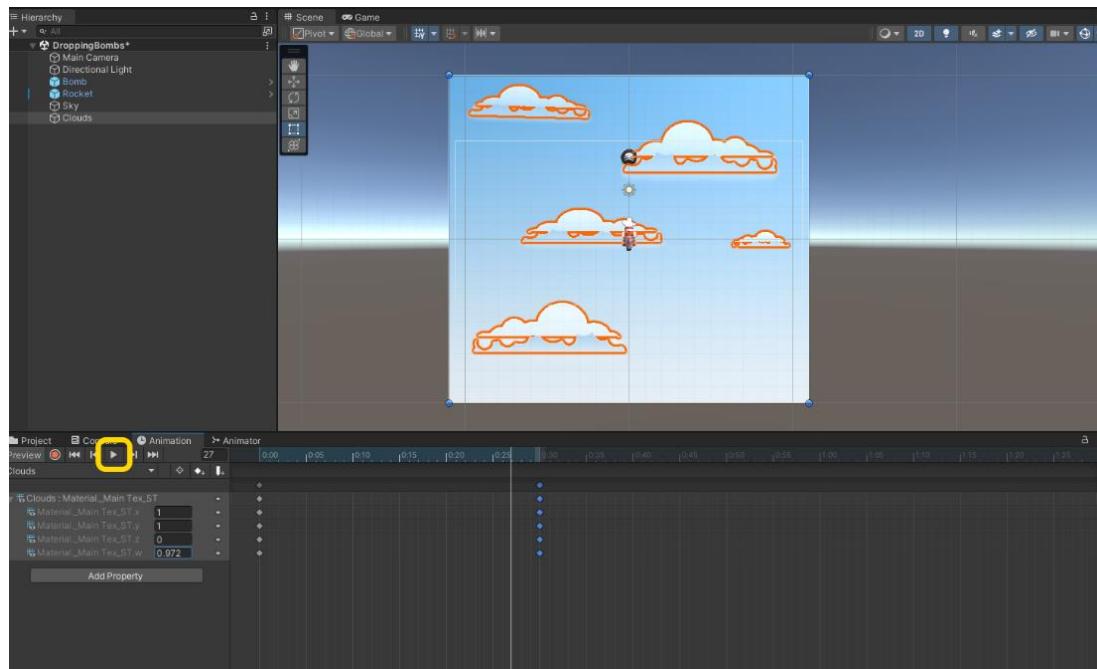
Make sure that you're at 0:30 in the timeline as shown below. Of the four properties, the only one that's being changed is **Material_Main_Tex_ST.w**. This controls the vertical offset property of the texture. Change it to **1**.



55

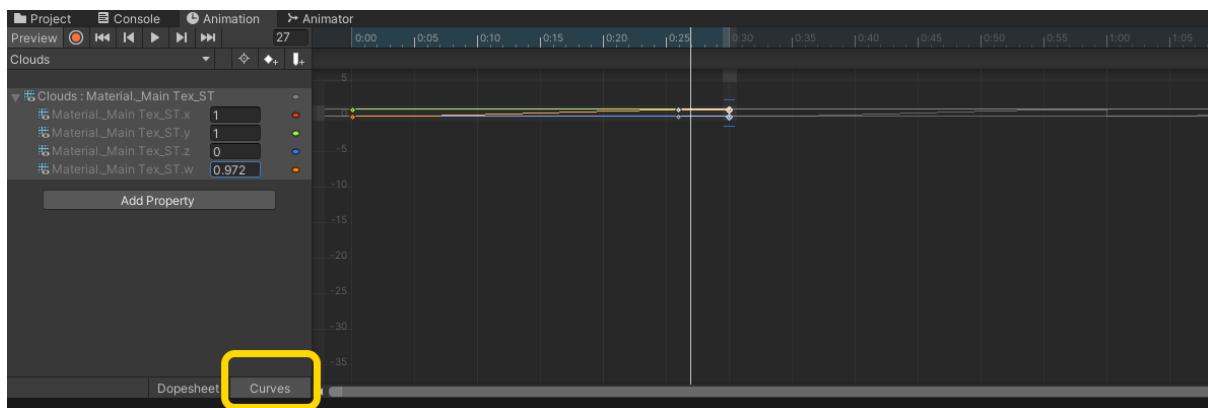
Click the black "play" arrow in the **Animation Preview** panel to check your animation. From 0:00 to 0:30, Unity is gradually changing the offset value from 0 to 1 and making it look like the clouds are moving.

If it looks like the clouds aren't moving smoothly, there's a fix for that in the next few steps.



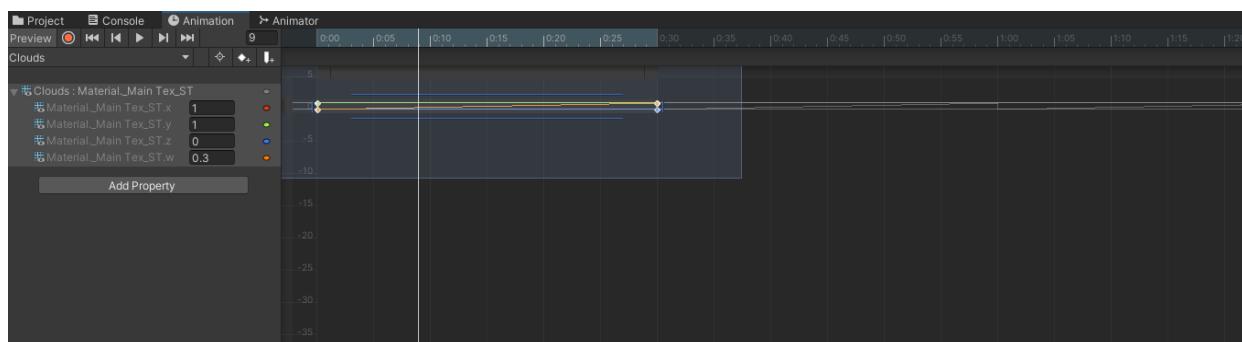
56

If the clouds look like they're lurching (speeding up and slowing down over and over again), it's because Unity **Animation** has automatically added "easing." Instead of a straight line from the beginning to end, the animation is a curve. To fix this, click **Curves** in the **Animation** panel.

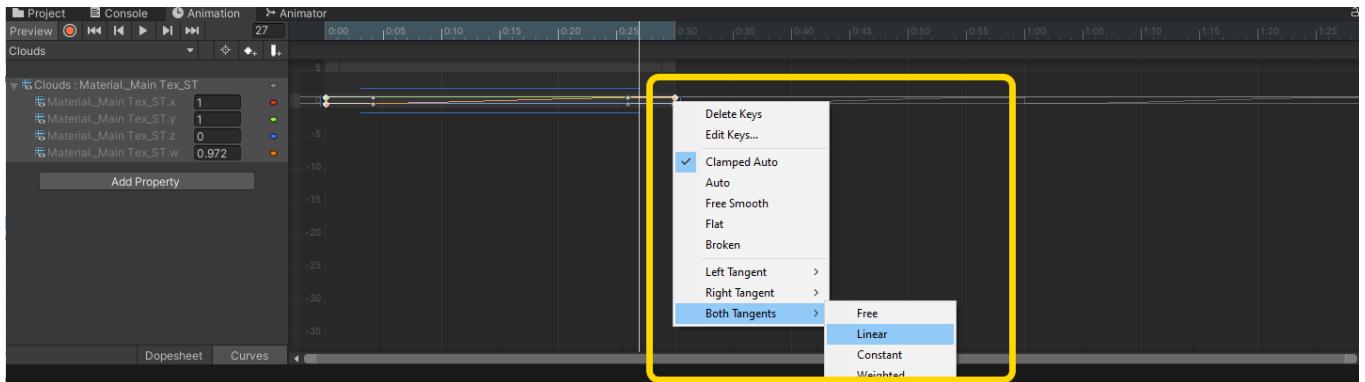


57

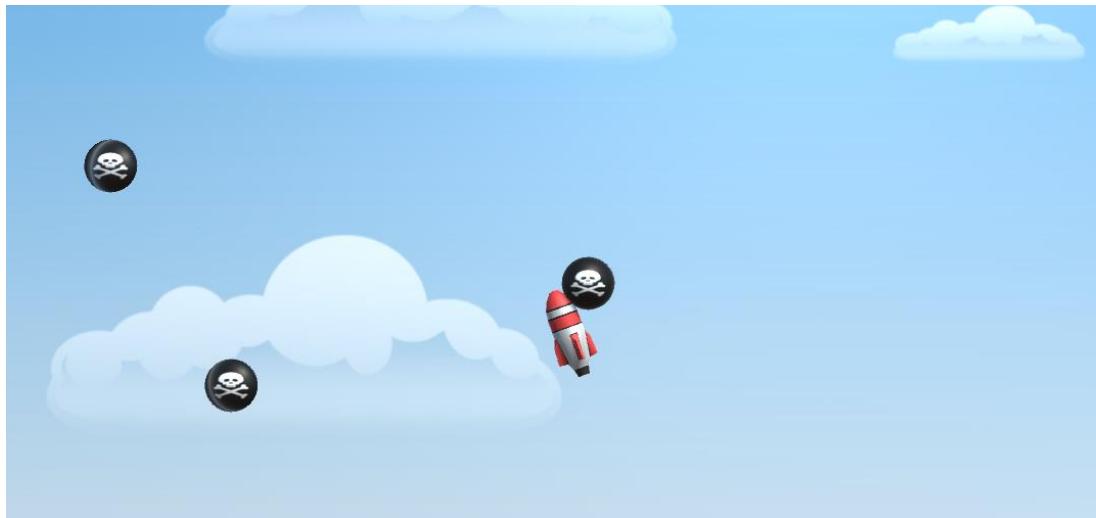
This view is a diagram showing how the properties change as the animation plays. Select all of the points in the diagram by dragging a rectangle around them as shown.



- 58** With all of the points selected, right-click on one of the points on the right side to open the menu shown below. Click **Both Tangents** and select **Linear**. This changes the animation from a curve to a straight line.



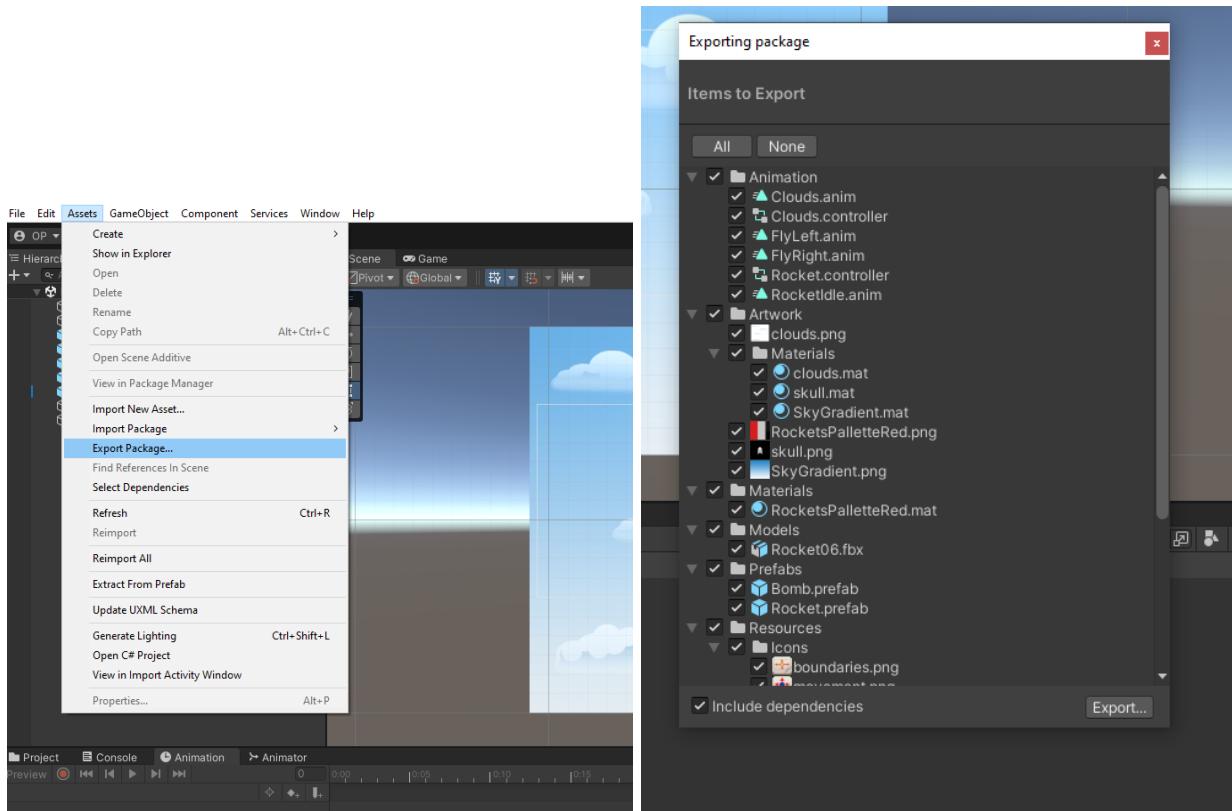
- 59** Now play your game. It should look like the rocket is flying higher and higher. However, the game doesn't have scoring and the way it resets is rather abrupt. We will tackle that in the next activity.



Activity 9: Dropping Bombs Part 3

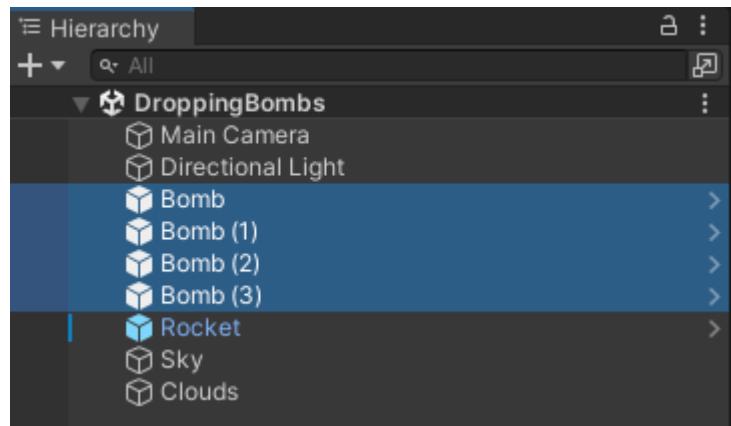
Now that you've updated the objects in the game, you can add scoring and other elements of the User Interface. To do that, you'll need to make a few changes to the game itself so that it doesn't start until the player is ready for it to start.

- 1 Before making additional changes to your project, let's make a backup of the entire game. In the **Projects** panel, make sure that the **Assets** folder is selected. Then click the **Assets** tab at the top of the screen and select **Export Package**. Make sure all assets are selected before clicking on the **Export** button. Give the package a name like **JS-DroppingBombsPart2**.



2 To set the game to start only when the player wants it to, we have to be able to control the objects in the game itself. That means having the game create and destroy the objects. Let's start with the bombs.

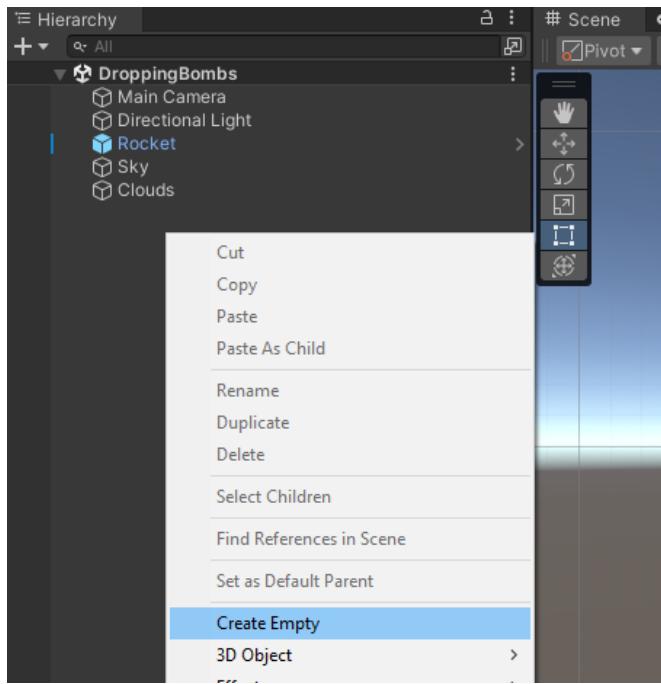
In the previous activity, the **Bomb GameObject** was made into a **Prefab**. We can delete any extra **Bomb** objects in the **Hierarchy** without affecting the object in the **Prefabs** folder. If you have any Bombs in the Hierarchy from testing, select all of them and delete them.



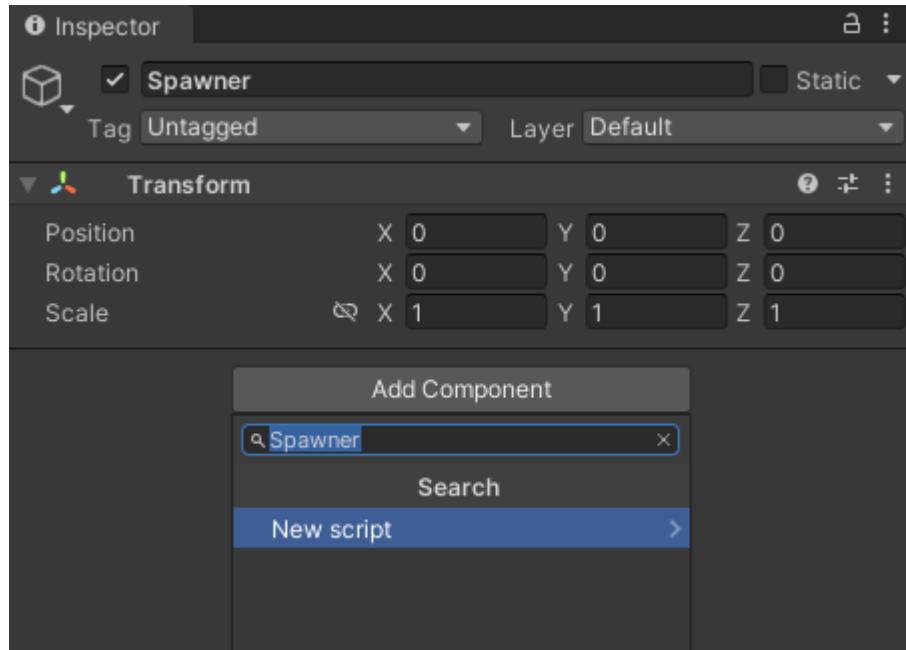
3 The dropping bombs no longer need to teleport back to the top of the scene. Open the **Prefabs** folder and select the **Bomb Prefab**. In the Inspector, find the Teleport script and remove it by clicking the three dots in the right corner and selecting **Remove Component**.



- 4** Now a **GameObject** can be created to automatically spawn the Bombs in the game. Right-click in the **Hierarchy** and select **Create Empty**. Rename this new empty object as **Spawner**.



- 5** All the **Spawner GameObject** will do is hold the script that spawns the bombs. Click **Add Component** in the inspector panel and search for **Spawner**. The script currently doesn't exist, so click **New Script**, **Create**, and **Add**.



Unity might place your new script in the **Assets** folder instead of the **Scripts** folder. If it does, just drag the script into the **Scripts** folder.

- 6** Double-click the **Spawner** script to open it up in the script editor. The next few steps will guide you through how to create this script and the image on step 10 will show you how it is meant to look. Try to follow the steps before looking at the image below.
- 7** First, we need a variable. Create a **Public** variable with the type **GameObject** and name it **bombPrefab**.
- 8** The next variable is the default setting for how often the spawner makes a new object. This also needs to be **public**. The type is **float** and the name should be **delay**. Set it to equal to **2.0f**.
- 9** We need a variable to let us know if the Spawner is active or not. Again, make this a **public** variable. The type is **bool** since it will only be true or false. Name it **active** and set it equal to **false**.

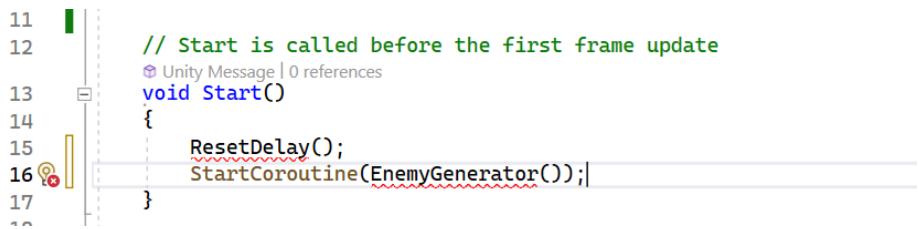
10 The last variable will be to let us set the range for a random number for the spawn delay. We could use two different variables, but it's easier to use a **public Vector2** variable. Call it **delayRange** and set it equal to a new **Vector2(1,2)** (The x will be the first part of our range and the y will be the second part).

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class Spawner : MonoBehaviour
6  {
7      public GameObject bombPrefab;
8      public float delay = 2.0f;
9      public bool active = true;
10     public Vector2 delayRange = new Vector2(1, 2);
11
12     // Start is called before the first frame update
13     void Start()
14     {
15 }
```

-
- 11** When the script is started, choose a random number for the delay and have it run the function that spawns the objects. For the first, we'll call a function named **ResetDelay()** (with no parameters).

Then, call the second function using **StartCoroutine**. A **Coroutine** is like a function that can pause execution and return control to Unity and then continue where it left off. This is very useful since we'll be constantly pausing the spawn process until the random delay has expired.

The **StartCoroutine** calls the **EnemyGenerator** function.



```
11 // Start is called before the first frame update
12
13 void Start()
14 {
15     ResetDelay();
16     StartCoroutine(EnemyGenerator());
17 }
```

The screenshot shows a portion of a C# script in a code editor. Lines 11 through 17 are visible. Lines 15, 16, and 17 contain underlined errors: 'ResetDelay()', 'StartCoroutine()', and 'EnemyGenerator()'. The code editor has a vertical ruler on the left and a status bar at the bottom.

At this point, you will have errors in your code, this is because we haven't made the functions yet

12 Unlike other functions, a coroutine *must* start with the **IEnumerator** data type. So instead of void, our function is **IEnumerator EnemyGenerator()**

(See if you can try to figure out what code you need from the text before looking at the below image)

The first line in the function is **yield return new WaitForSeconds(delay);**. As you might have guessed, this is the part of the coroutine that lets Unity do what it needs to do while waiting for the required length of time.

The next line is a conditional that checks if the **Spawner** is active or not. Remember, we don't want to start the game until the player is ready and this keeps anything from spawning until it is active.

Inside the conditional, we do two things. We use **Instantiate** to spawn our **bombPrefab** at a position of 0,0,0 (right in the middle) with the current rotation of the prefab. Once that is done, we call the **ResetDelay** function to choose a new random number for the delay.

After all of that, regardless of whether anything was spawned, we start the coroutine all over again.

```
15     RESETDELAY();
16     StartCoroutine(EnemyGenerator());
17 }
18
19 2 references
20 IEnumerator EnemyGenerator()
21 {
22     yield return new WaitForSeconds(delay);
23     if(active) {
24         Instantiate(bombPrefab, new Vector3(randomX, spawnY, 0), bombPrefab.transform.rotation);
25         ResetDelay();
26     }
27
28     StartCoroutine(EnemyGenerator());
29 }
30
31 }
```

Again, at this time, you will have an error, because we still need to make the **ResetDelay** Function

13 The **ResetDelay** function is just a single line setting **delay** to equal a **Random.Range** between the values of **delayRange.x** and **delayRange.y**.

```
31 2 references
32 void ResetDelay()
33 {
34     delay = Random.Range(delayRange.x, delayRange.y);
35 }
36
```

14 There are still some errors to fix, but before that, we are going to fix a problem that may occur. Currently, if the game were to be run, the bombs would always spawn at (0,0,0). We are going to fix this first by using a private variable.

This variable is a **Vector2** that will be called **screenBounds** and hold the dimension of the screen.

```
7 public GameObject bombPrefab;
8 public float delay = 2.0f;
9 public bool active = true;
10 public Vector2 delayRange = new Vector2(1, 2);
11
12 private Vector2 screenBounds;
13
14 // Start is called before the first frame update
```

15 When you start the script, you will get the value for this new variable. **screenBounds** is from the **Camera.main.ScreenToWorldPoint** as a new **Vector3** with x as the **Screen.width**, y as the **Screen.height** and z as the **Camera.main.transform.position.z**.

```
14 // Start is called before the first frame update
15 // Unity Message | 0 references
16 void Start()
17 {
18     ResetDelay();
19     StartCoroutine(EnemyGenerator());
20
21     screenBounds = Camera.main.ScreenToWorldPoint(new Vector3(Screen.width, Screen.height, Camera.main.transform.position.z));
22 }
```

16

The center of the screen is 0,0 so the left side is negative x (-x) and the right side is positive x (+x). We want to spawn the bomb somewhere between those two points.

To do this, before we Instantiate our bomb, make a **float** variable called **randomX** that is a **Random.Range** between **-screenBounds.x** and **screenBounds.x**. This will give us a random number between the left and right side of our screen.

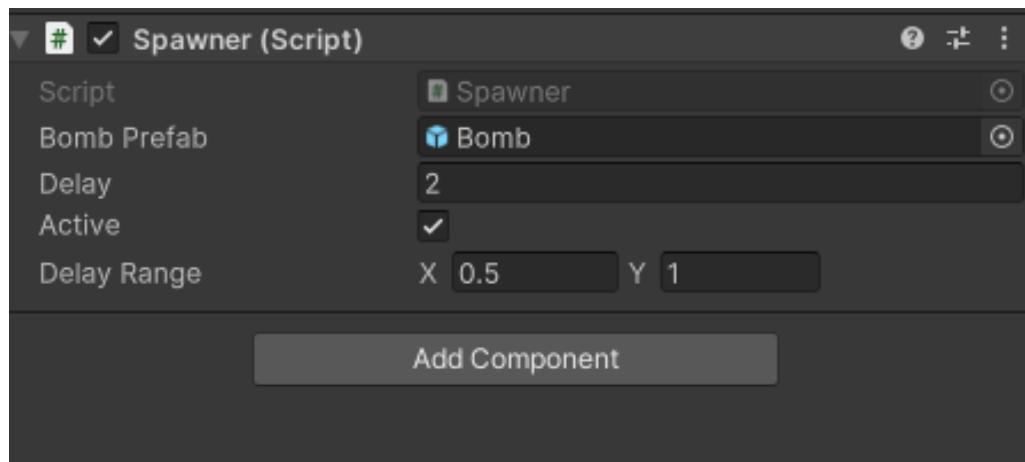
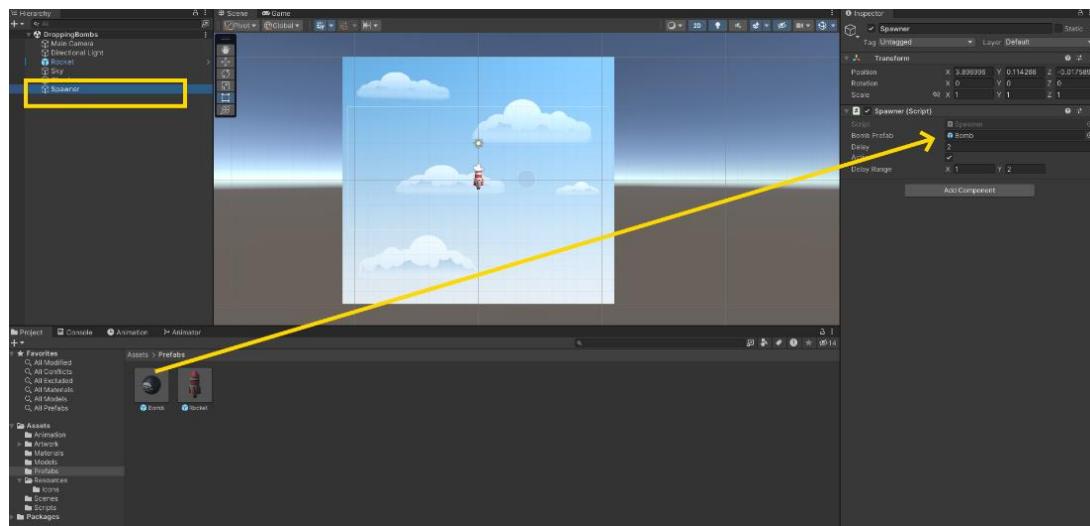
The Y position will always be the same, based on the **screenBounds.y + 1**

The **+1** is there to give a bit of space; you can try changing this number if you like.

These two numbers are passed to our **Instantiate** from earlier. If all is correct, the errors from earlier will be resolved.

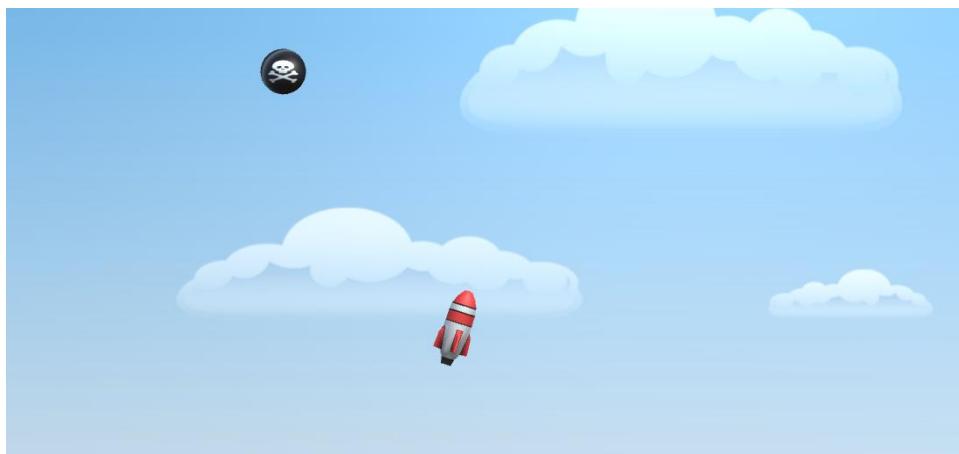
```
23     2 references
24     IEnumerator EnemyGenerator()
25     {
26         yield return new WaitForSeconds(delay);
27         if(active)
28         {
29             float randomX = Random.Range(-screenBounds.x, screenBounds.x);
30             float spawnY = screenBounds.y + 1;
31             Instantiate(bombPrefab, new Vector3(randomX, spawnY, 0), bombPrefab.transform.rotation);
32             ResetDelay();
33         }
34     }
35     StartCoroutine(EnemyGenerator());
36 }
37
```

17 Save the **Spawner** script and return to Unity. Select the **Spawner Gameobject** in the **Hierarchy**. Search for the **Spawner** script in the inspector panel. The script needs to know what object to use for the **Bomb Prefab**. Open the **Prefabs** folder in the **Project** panel and drag the **Bomb** from the folder into the script as shown.



18 Now test the game by clicking the **Play** arrow above the Scene. You should see a new bomb between one and two seconds.

When you are done, stop the game by pressing the **Play** arrow again.



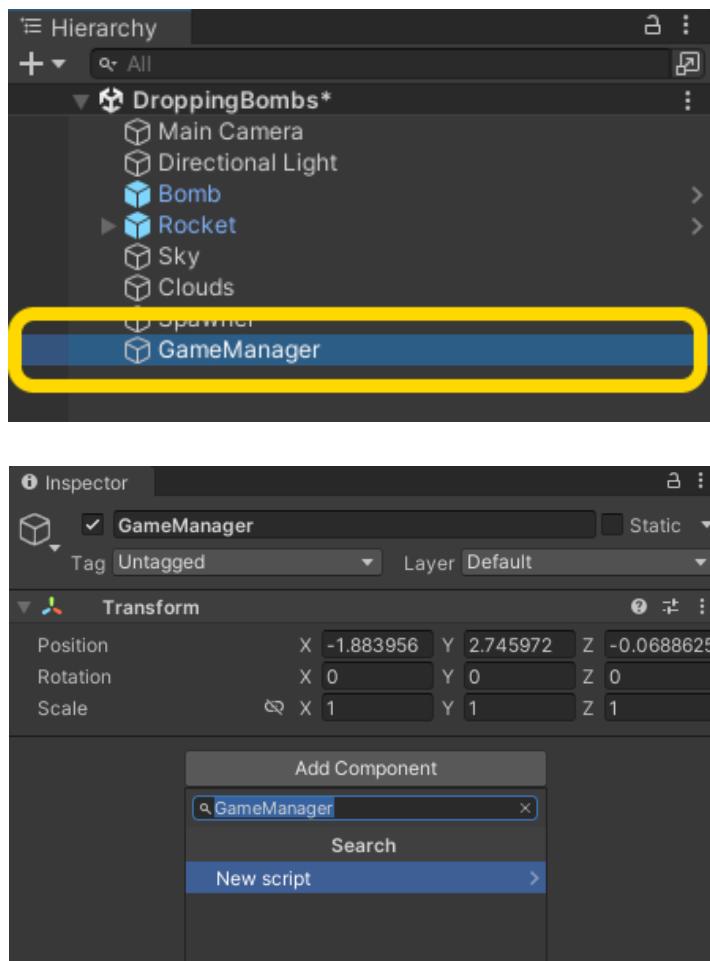
Not Enough Bombs?

If you want to see more bombs in your scene, change the Delay Range in the Spawner script. Lower numbers mean a shorter delay, creating more bombs!

19

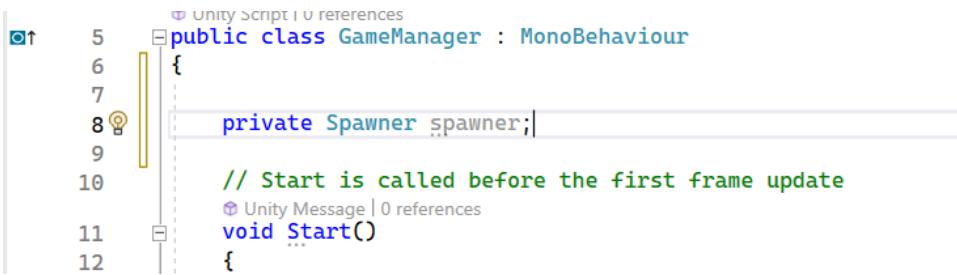
The **Spawner** allows you to make (and stop) bombs whenever you need to. Now there needs to be something to tell the **Spawner** when to turn it off and on. To do this, you will make a **GameManager**.

Just as you did with the **Spawner**, create an empty **Object** in the **Hierarchy** and name it **GameManager**. In the **Scripts** folder, create a new **C#** script and also give it the name of **GameManager**. Drag the new script onto the **GameManager GameObject**.



Why does the **GameManager** script look like it does? Unity has a special symbol reserved for these types of scripts. Other than the icon, it's exactly like all of the other scripts.

20 Open the **GameManager** script by double-clicking it. To communicate with the **Spawner**, you must first create a variable for it. Make it a **private** variable of type **Spawner** and give it the name **spawner** (all lower case).



```
5  public class GameManager : MonoBehaviour
6  {
7
8    private Spawner spawner;
9
10   // Start is called before the first frame update
11   void Start()
12   {
```

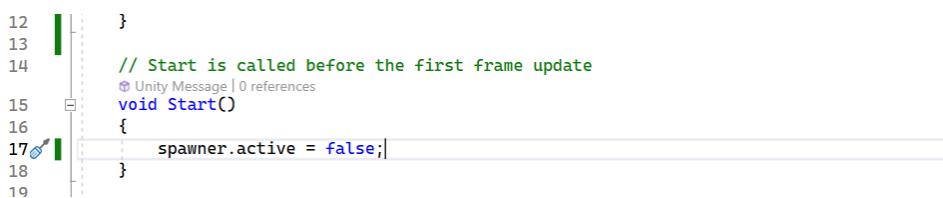
21 We have yet to identify the spawner, which we can do when the script first awakes. Create a void function for **Awake**. Similar to **void Start**, **void Awake** is called when the game loads, but Awake is called before Start.

Inside the **Awake** function, we are going to find the **GameObject** called "**Spawner**" using the **GameObject.Find** function. However, since this finds a **GameObject** we will get an error since our variable is of the type **Spawner** and not **GameObject**. To fix this, we can add **GetComponent<Spawner>()** to the end of the line, to access the component on that object.



```
4
5  public class GameManager : MonoBehaviour
6  {
7    private Spawner spawner;
8
9    private void Awake()
10   {
11     spawner = GameObject.Find("Spawner").GetComponent<Spawner>();
12   }
13 }
```

22 Now that **spawner** is defined as the script component of the **Spawner GameObject**, you can access the functions and variables within. We want to turn the **Spawner** script off, so when the **GameManager** script starts, we can have the **active** variable of the **spawner** component set to false.



```
12
13
14   // Start is called before the first frame update
15   void Start()
16   {
17     spawner.active = false;
18   }
19 }
```

23 Lastly, there will need to be some way to turn the spawner script component back on. We need to listen for player input, which means we will use the **Update** function. Inside the function, you'll set up a conditional for input. If the user hits any key (or button), then you will set **spawner.active** to **true**. This can be accomplished with **Input.anyKeyDown**. So, when the user does anything, the spawner is activated.

```
19 // Update is called once per frame
20
21 void Update()
22 {
23     if (Input.anyKeyDown)
24     {
25         spawner.active = true;
26     }
27 }
28
29
```

24 Save your script and return to Unity.

If you start the game now (go ahead, we'll wait), you'll see the rocket and the animations work, but there are no bombs until you hit any key (which means that the second that you use the arrow keys to move the rocket, the spawner becomes active). It would be useful to have a message to tell the player that they're supposed to press any key to start the game. This will be the first part of our **User Interface**.

25

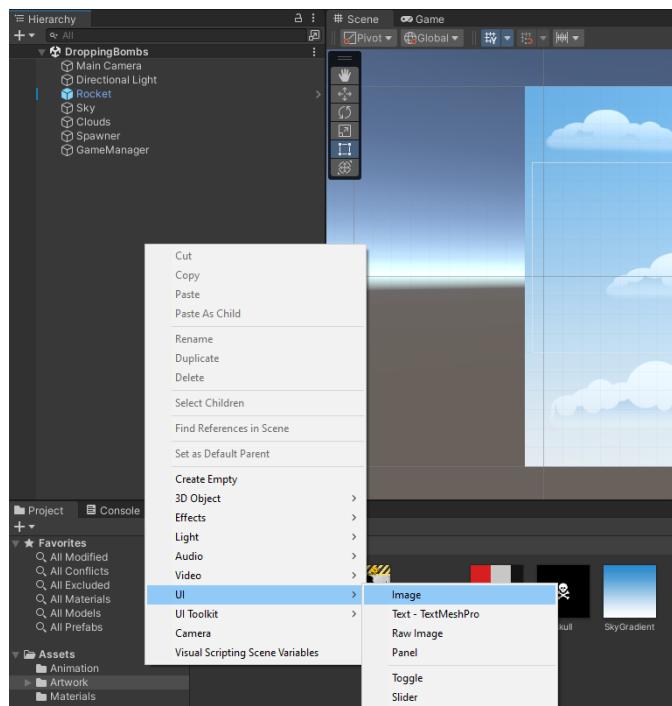
If you started the game, stop it before continuing to the next step.

Artwork for the User Interface has been prepared for you. Go to the **Artwork** folder and right-click inside the folder. Select **Import New Assets** and from the Purple Belt assets folder, find and import **Activity 09 - DBTitle.png** and **Activity 09 - grungeHazard.png**. With both images selected, go to the **Inspector** panel and change **Texture Type** to **Sprite (2D and UI)**. Scroll down to the bottom of the Inspector and click **Apply**.



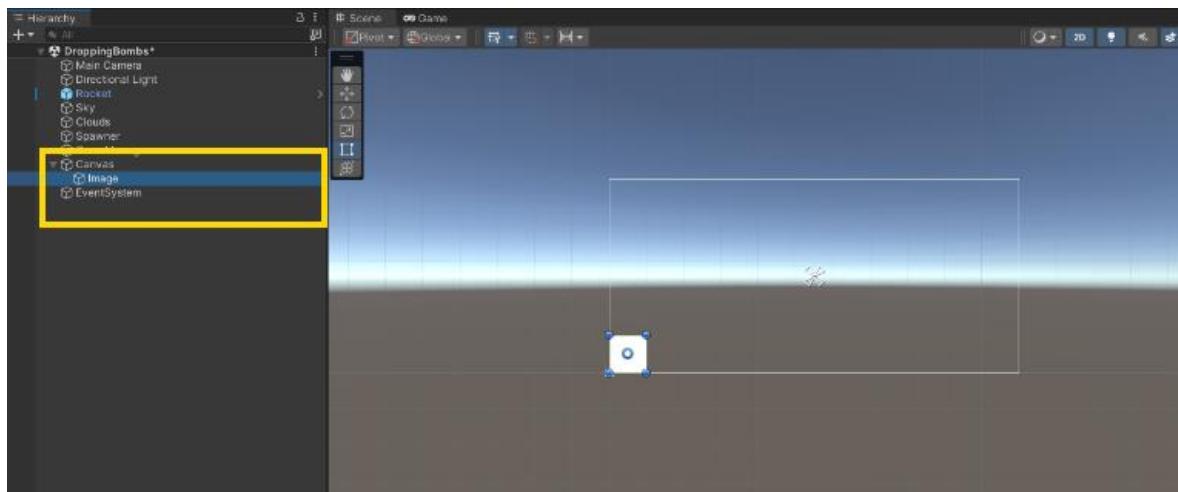
If at any time when you are making edits to the texture component and you forget to hit “Apply” a reminder pops up. Either apply the changes or revert them.

26 Right-click in the **Hierarchy** panel to bring up the **Create** menu. Select **UI**, then select **Image**.

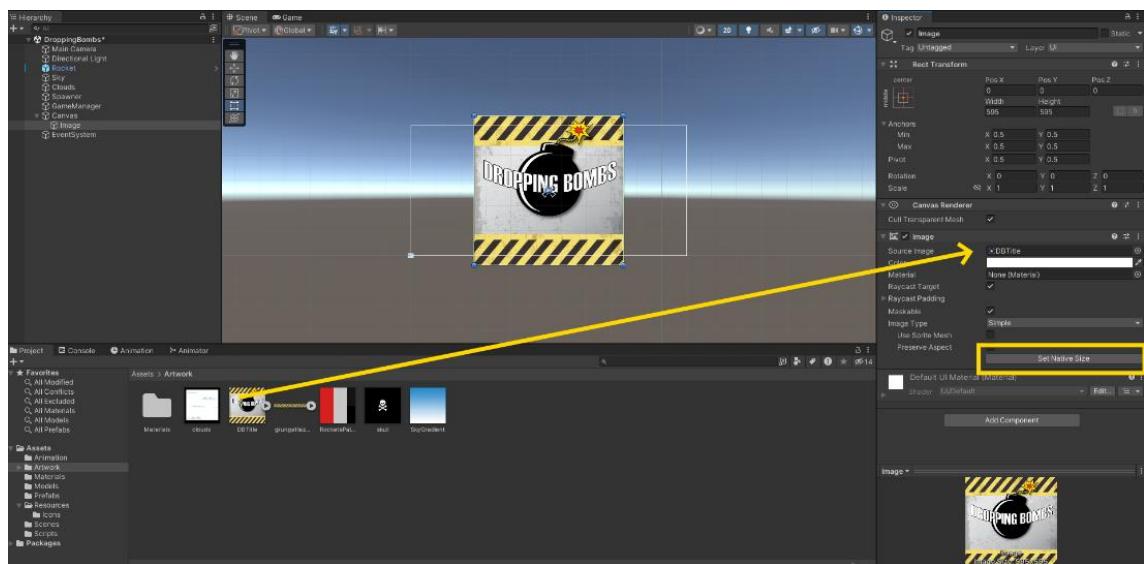


27 When you do this, a **Canvas** object is created to hold all the **UI** elements. Additionally, an **EventSystem** is created to support the **Canvas**. Without an **EventSystem**, you won't be able to interact with the **UI**.

The **Canvas** is much larger than the camera view, so double click the newly created image in the Hierarchy to focus onto it.



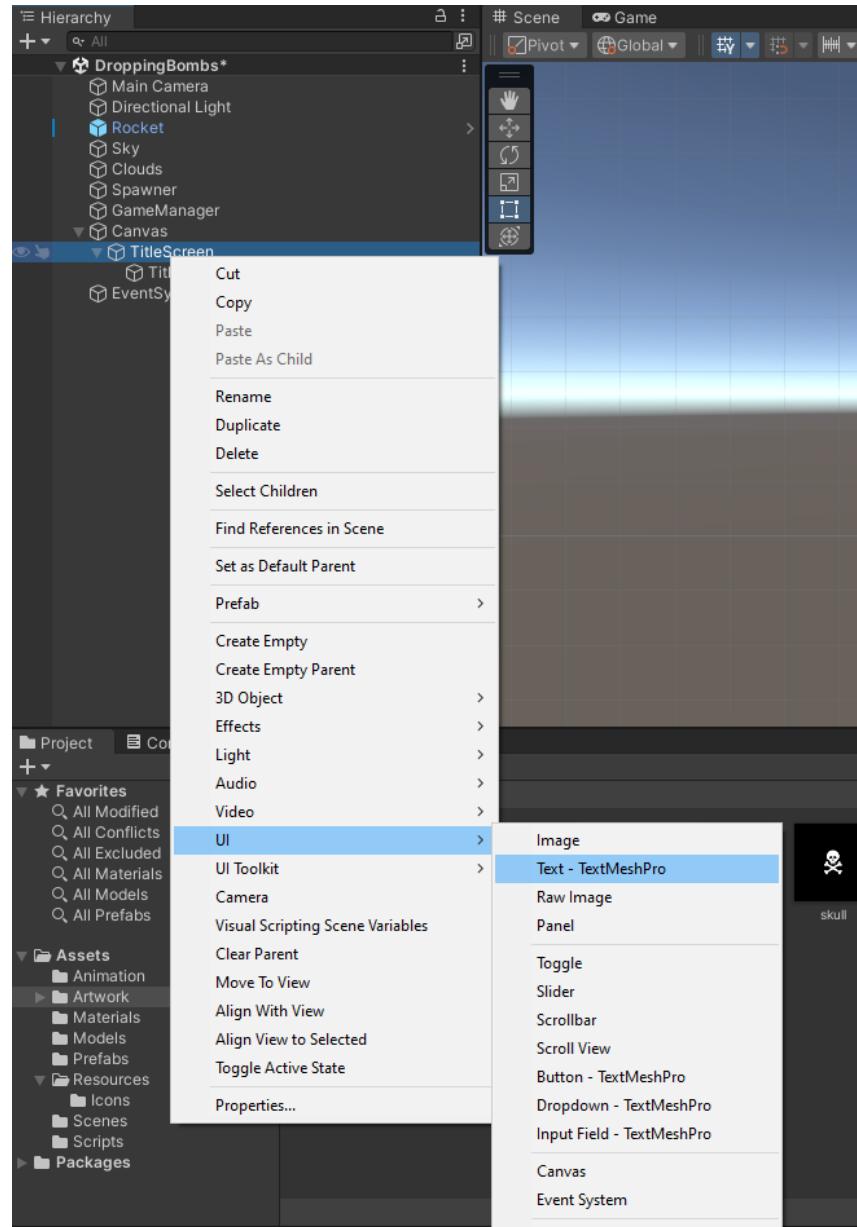
28 Change the name of the **Image** to **Title**, then click on the gear and select **Reset** to put the image to the center of the **Canvas**. Drag **DBTitle** into the slot for the **Source Image** and click **Set Native Size** to enlarge the image to the size of the graphic.



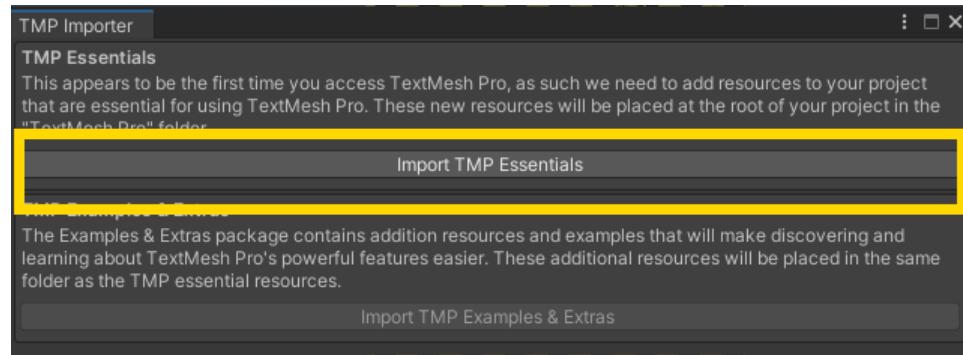
29 The title screen will have several objects, so let's create an **Empty Object** inside of the Canvas to hold everything and call it **TitleScreen**. Drag the **Title** object so that it is inside of the **TitleScreen** object.



30 Next, let's add some text that tells the player how to start the game. Right-click on the **TitleScreen** object and go back to **UI**. We are going to add **Text - Text Mesh Pro**.



31 You may get a box that tells you to import Text Mesh Pro. If so, click **Import TMP Essentials**. This will install all the required TMP assets. This may take a few minutes.



32 With that installed, our text should appear in the middle. Let's stretch it and move it somewhere sensible. Inside the text, we can write our message. You can create something similar to the image below.

You may wish to tinker with the size and the color to make it look even better.



33 Now that the player knows that they need to press any key to start the game, we need to add code to hide the **TitleScreen**. Switch back to the **GameManager** script in the script editor.

Let's start by adding a **public** variable to hold the **TitleScreen Gameobject**.

```
5  Unity Script | references
6  public class GameManager : MonoBehaviour
7  {
8      private Spawner spawner;
9      public GameObject title; // Line 9 highlighted with a yellow box
10     private void Awake()
11     {
12         spawner = GameObject.Find("Spawner").GetComponent<Spawner>();
13     }
14 }
```

34 When the **spawner** is off, the **title** should be on. When the **spawner** is on, the **title** should be off. In the **Start** function, add a line for the **GameObject title.SetActive(true)**. In the conditional for **Input** in the **Update** function, add a line for the **GameObject title.SetActive(false)**. Save the script.

```
14 // Start is called before the first frame update
15 // Unity Message | 0 references
16 void Start()
17 {
18     spawner.active = false;
19     title.SetActive(true); // Line 18 highlighted with a yellow box
20 }
21
22 // Update is called once per frame
23 // Unity Message | 0 references
24 void Update()
25 {
26     if (Input.anyKeyDown)
27     {
28         spawner.active = true;
29         title.SetActive(false); // Line 27 highlighted with a yellow box
30     }
31 }
```



PRO TIP!

You might be wondering why the boolean for the **spawner** is set to equal true or false while the boolean for **title** uses **SetActive** with a parameter of true or false. In **Unity**, a **GameObject** (like **title**) can use **SetActive**. However, **spawner** is not a **GameObject**, it is a component inside of a **GameObject**. With a component, we have to work with the functions and variables inside the

35 Switch back to **Unity** and select **GameManger** in the **Hierarchy**. Drag **TitleScreen** from the **Hierarchy** into the slot for **Title** in the **GameManager** script component.



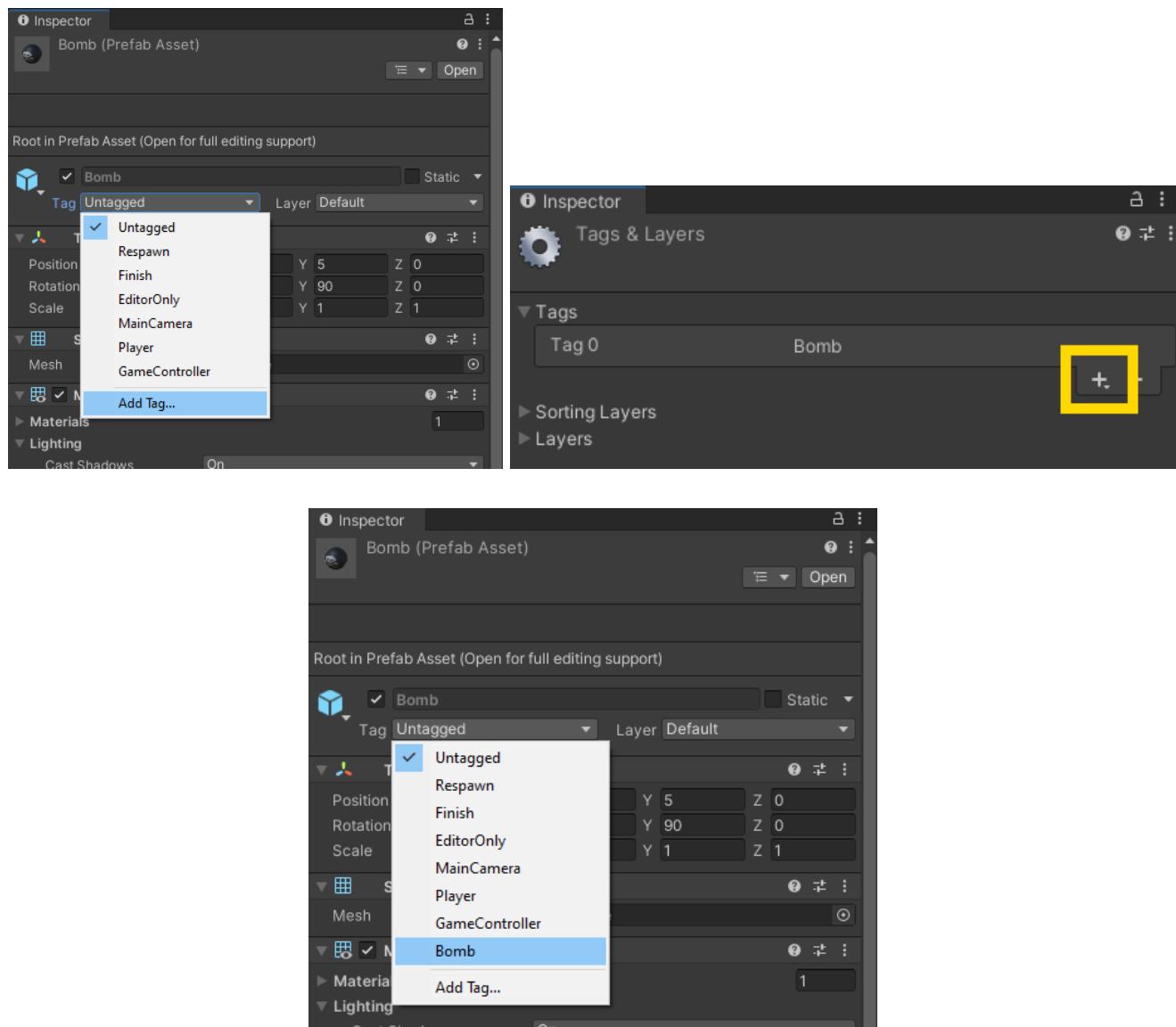
36 Now start the game by clicking the Play arrow above the Scene. The interface appears with a friendly message about how to start. When you press any key, the interface is hidden and the Bombs start spawning. So far, so good.

Stop the game by clicking the Play arrow a second time.

37

When the bombs spawn, they stay in the program's memory, with more bombs being added every second! We don't need the bombs after they've passed off the bottom of the screen, so we can get rid of them and use that for scoring, too.

Every time a new bomb is spawned, it gets a new name. We need another way to identify the spawned bombs. Select the **Bomb** prefab in the **Prefabs** folder. Click the **Tag** menu and select **Add Tag**. In the **Tags & Layers** menu, click the plus (+) symbol and name the new tag "**Bomb**". Select the **Bomb** prefab again and set the **Tag** to **Bomb**.



38

Switch back to the **GameManager** in the script editor. To know when the Bomb has gone below the screen, you can use the same **screenBounds** that we used when spawning the Bomb.

Add a **private Vector2** variable with the name of “**screenBounds**”. In the **Awake** function, add the same line that we used for the Spawner, setting **screenBounds** to equal **Camera.main.ScreenToWorldPoint** as shown below.

```
private Spawner spawner;
public GameObject title;
private Vector2 screenBounds;
// Unity Message | 0 references
private void Awake()
{
    spawner = GameObject.Find("Spawner").GetComponent<Spawner>(),
    screenBounds = Camera.main.ScreenToWorldPoint(new Vector3(Screen.width, Screen.height, Camera.main.transform.position.z));
}
```

39

Switch back to the **GameManager** in the script editor. Next, we’ll need to find any object with the **Bomb** tag and remove it when it goes below a certain point. In the **Update** function, add a line at the bottom that sets **var nextBomb = GameObject.FindGameObjectsWithTag("Bomb")**.

```
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
```

```
// Update is called once per frame
// Unity Message | 0 references
void Update()
{
    if (Input.anyKeyDown)
    {
        spawner.active = true;
        title.SetActive(false);
    }

    var nextBomb = GameObject.FindGameObjectsWithTag("Bomb");

    foreach (GameObject bombObject in nextBomb)
    {
    }
}
```

40

Now that we have all bombs, we need to loop through them. You may be familiar with a **for loop**, but instead we will use a **foreach loop** instead. A **foreach loop** is a way to loop through every element in an array or a list. Specify what type it is, then what name you’ll use to access that element (Similar to using ‘*i*’ in a **for** loop), and what array to loop through. In our

example, we are looking through all **GameObjects** (called **bombObject**) in the **array nextBomb**.

41

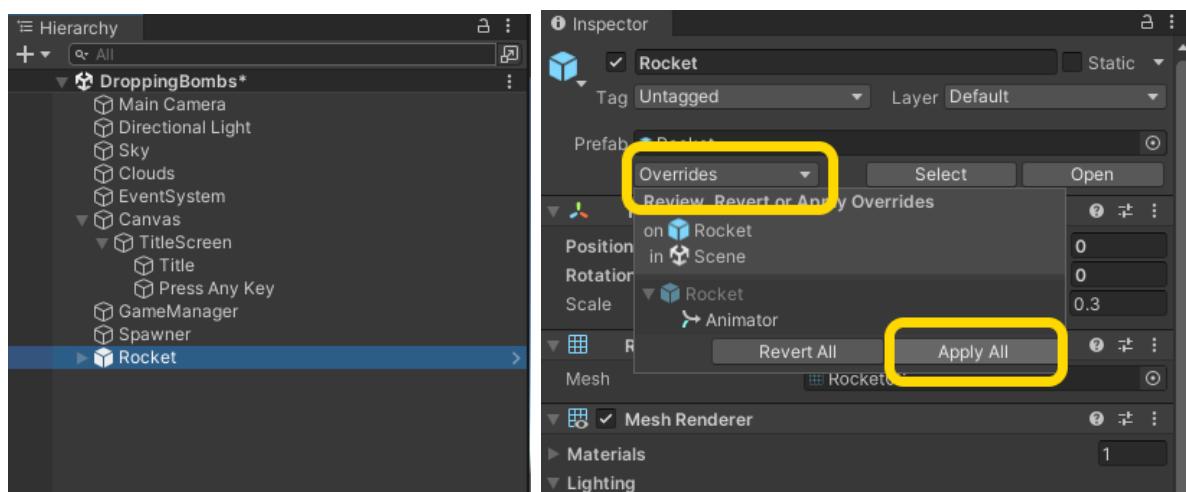
Inside the loop, we need a conditional that compares the Y value of the **bombObject** to see if it is off our screen. Any object that falls below our screen will be destroyed. Later, we'll use this for scoring as well.

```
22 // Update is called once per frame
23 // Unity Message | 0 references
24 void Update()
25 {
26     if (Input.anyKeyDown)
27     {
28         spawner.active = true;
29         title.SetActive(false);
30     }
31
32     var nextBomb = GameObject.FindGameObjectsWithTag("Bomb");
33
34     foreach (GameObject bombObject in nextBomb)
35     {
36         if (bombObject.transform.position.y < (-screenBounds.y - 12))
37         {
38             Destroy(bombObject);
39         }
40     }
41 }
42
43 }
```

Don't forget to save your script.

42

In addition to the bombs, the rocket should be spawned and destroyed as well when the game starts and stops. However, the **Rocket** prefab was changed in the previous activity by adding the **Rocket Animate** script. When changes are made to a prefab in the **Hierarchy**, the prefab itself does not automatically get those changes. To make sure that our **Rocket** prefab is up to date, select the **Rocket** in the **Hierarchy**, and in the **Inspector** panel, click **Overrides** and select **Apply All** to make sure that the prefab matches the **GameObject** in the **Hierarchy**. After that, it will be safe to delete the **Rocket** from the **Hierarchy**.



43

Let's have the **GameManager** spawn the **Rocket** from the **Prefabs**. Switch back to the **GameManager** script in the script editor and add a **public** variable with the type of **GameObject** and name it **playerPrefab**. You also need a **boolean** to know if the game has started or not. Add a **private** variable of the **bool** type and give it the name of "**gameStarted**" and set it to **false**.

```
7     private Spawner spawner;
8     public GameObject title;
9     private Vector2 screenBounds;
10
11    [Header("Player")]
12    public GameObject playerPrefab;
13    private GameObject player;
14    private bool gameStarted = false;
15
16    void Awake()
17    {
18        spawner = GameObject.Find("Spawner");
19        screenBounds = Camera.main.ScreenToWorldPoint(new Vector3(Screen.width, Screen.height, 1));
20    }
```

44

Our **Update** function is getting a bit long and complicated. We are going to create a new function below **Update** and move some of the code inside to make it easier to read. This new function will be called **ResetGame**.

45

Below the **Update** Function, add the **void ResetGame** function. Inside the newly created function, put the **spawner.active** and **title.SetActive** lines that used to be in the **Input.anyKeyDown** conditional above. Then **Instantiate playerPrefab** in the middle of the scene using the current **playerPrefab** rotation. Then set the **gameStarted** boolean to true so that new players don't keep spawning while the game is running. Save the script.

```
48
49
50    void ResetGame()
51    {
52        spawner.active = true;
53        title.SetActive(false);
54
55        player = Instantiate(playerPrefab, new Vector3(0,0,0), playerPrefab.transform.rotation);
56        gameStarted = true;
57    }
```

46

Once we have created the function, we need to call it. We are going to call it when we press any key, but only if the game hasn't started.

```
28 // Update is called once per frame
29
30 void Update()
31 {
32     if (!gameStarted)
33     {
34         if (Input.anyKeyDown)
35         {
36             ResetGame();
37         }
38     }
39 }
```

47

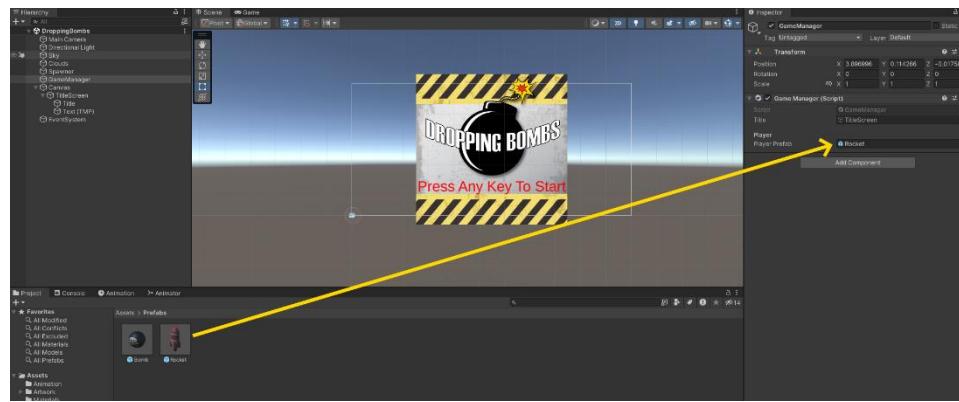
Currently, the **Rocket** has a **Reset** script that restarts everything when the **Rocket** collides with anything. In **Unity**, go into the **Scripts** folder and double-click **Reset** to edit it. There's only one line of code inside a single **OnCollision** function that has the **SceneManager.LoadScene(0)**. Replace that line with **Destroy(gameObject)** and save the script.

```
1 using System.Collections;
2 using System.Collections.Generic;
3 using UnityEngine;
4 using UnityEngine.SceneManagement;
5
6 public class Reset : MonoBehaviour
7 {
8     private void OnCollisionEnter(Collision collision)
9     {
10        Destroy(gameObject);
11    }
12 }
```

In **C#**, **gameObject** always refers to the **GameObject** that the script is attached to. It saves a bit of time.

48

Back in Unity, we need to assign the **Rocket** Prefab in the **GameManager**. We will do this now, so we don't forget later. The game won't work until we write some more code, so let's get back to coding!



49

While still in the script editor, switch back to the **GameManager** script. While the game is running, check to see if the **Rocket** (player **GameObject**) has been destroyed. We're already checking if **gameStarted** is false, so just check to see if the game is running by adding an **else** to the conditional. Inside the **else**, you'll add a conditional to check if the player object has been destroyed and if so, run the **OnPlayerKilled** function.

At this point, we will get an error, but we will fix that.

```
28
29
30 // Update is called once per frame
31
32 void Update()
33 {
34     if (!gameStarted)
35     {
36         if (Input.anyKeyDown)
37         {
38             ResetGame();
39         }
40     }
41     else
42     {
43         if (!player)
44         {
45             OnPlayerKilled();
46         }
47     }
48 }
```

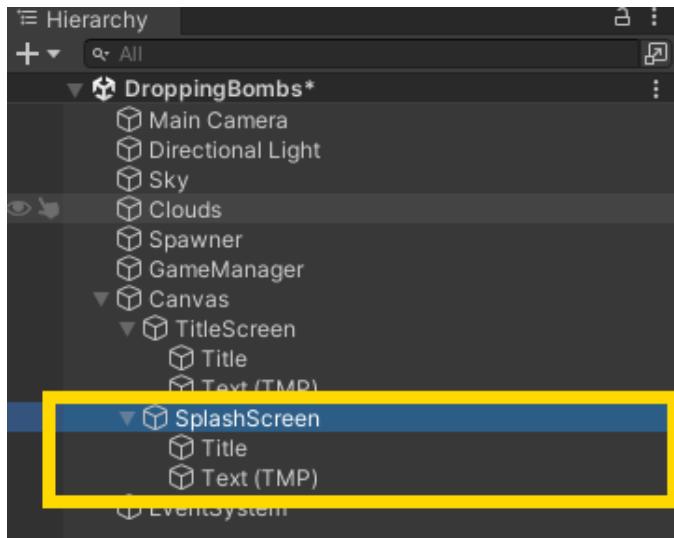
50

At the end of the script, let's add the **OnPlayerKilled** function. When the player is killed, the **spawner** should stop. Set **gameStarted** to **false**. Finally, you'll want to show the **splash** screen (which we haven't made yet, so you will get an error).

```
61
62
63
64
65
66
67 void OnPlayerKilled()
68 {
69     spawner.active = false;
70     gameStarted = false;
71
72     splash.SetActive(true);
73 }
```

51

Switch back to Unity to make a splash screen. To save time, use **Ctrl+D** to make a copy of the title screen and rename it **SplashScreen**. Then, adjust the elements within it as follows:



52

Change the **Title** image to **Background** and replace the **DBTitle** image with **grungeHazard**. Click the **Set Native Size** button.

53

The Text object gets renamed as **Play Again** and the text is altered to **Press Any Key to Play Again**. Change the color to black and move it up so that it's underneath the banner.

54

Use **Ctrl+D** to copy the Text object and make this into a “**Game Over**” object. Change the text to **GAME OVER**, increase the font size to **72** and make sure that this is white with a black outline. Use the **Rect Tool** to resize and adjust the text box. Feel free to change any of the fonts.

At this point, you can change how the Game Over screen looks. Below is an example of what it could look like.



55

Switch back to the **GameManager** script in the Script editor. Make a copy of **public GameObject title** and change the variable name to **splash**.

```
Unity Script (1 asset reference) | 0 references
6  public class GameManager : MonoBehaviour
7  {
8      private Spawner spawner;
9      public GameObject title;
10     private Vector2 screenBounds;
11
12     public GameObject splash; // Variable name changed from title
13
14     [Header("Player")]
15     public GameObject playerPrefab;
16     private GameObject player;
17     private bool gameStarted = false;
```

56

In the **Start** function, copy **title.SetActive(true)** and change it to **splash.SetActive(false)**.

```
24  
25 // Start is called before the first frame update  
26 Unity Message | 0 references  
27 void Start()  
28 {  
29     spawner.active = false;  
30     title.SetActive(true);  
31     splash.SetActive(false);  
32 }  
33  
34 // Update is called once per frame  
35 Unity Message | 0 references  
36 void Update()  
37 {
```

57

In the **ResetGame** function, copy **title.SetActive(false)** and rename it to **splash.SetActive(false)**. Save your script.

```
72  
73 Unity Message | 1 reference  
74 void ResetGame()  
75 {  
76     spawner.active = true;  
77     title.SetActive(false);  
78     splash.SetActive(false);  
79  
80     player = Instantiate(playerPrefab,new Vector3(0,0,0), playerPrefab.transform.rotation);  
81     gameStarted = true;  
82 }  
83  
84 }
```

58

Switch back to Unity. With the **GameManager** object selected, make sure that the objects for **SplashScreen** and **Rocket** are in the appropriate slots in the **GameManager** script in the **Inspector** panel. Save your project.



59

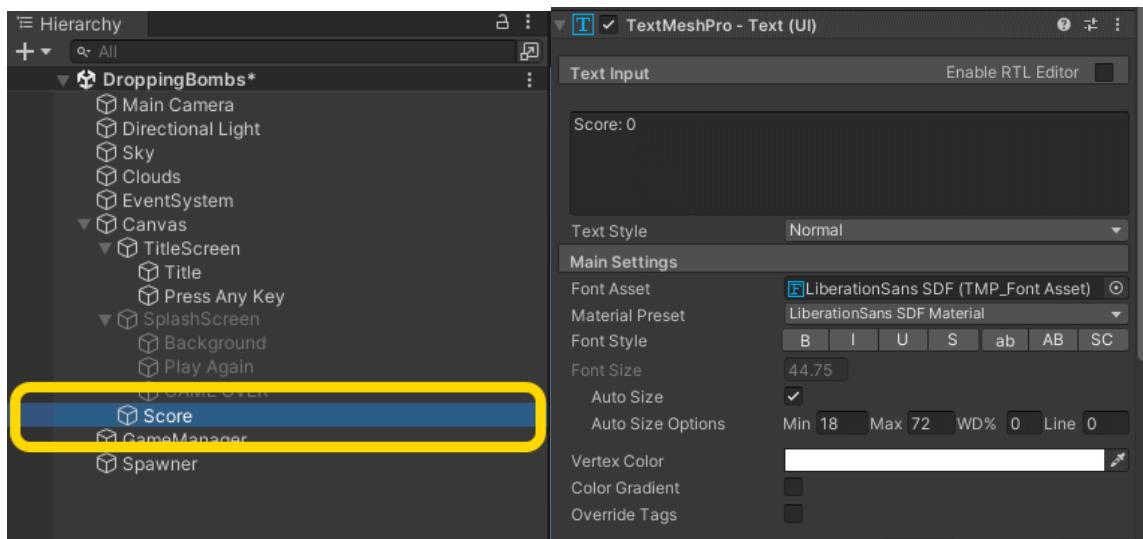
Play your game by clicking the **Play** arrow above the **Scene**. When you lose, the splash screen should appear and give you a chance to play again. All that's left to do is the score.

Stop your game by clicking the **Play** arrow again.



60

With the **Canvas** Selected, right click on it and add **Text Mesh Pro Text**. Use the **Move Tool** to move the object to the upper left of the canvas. Change the text to the upper left of the Canvas and change it to say **Score: 0**. You can resize and change the font style if needed.



61

We are going to update our score text to display the number of bombs that have been deleted. When a bomb gets removed, we get a point.

First, let's go back to our **GameManager** script and add the needed libraries at the top for **Text Mesh Pro**.



```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4  using TMPro;
5
6  public class GameManager : MonoBehaviour
7  {
8      private Spawner spawner;
9      public GameObject title;
```

62

Next, let's add some variables for our score. We want a **public TMP_Text** that is the score text component.

Add a **public** integer that says how many points we gain each time, and a **private** integer that holds our score.

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4  using TMPro;
5
6  public class GameManager : MonoBehaviour
7  {
8      private Spawner spawner;
9      public GameObject title;
10     private Vector2 screenBounds;
11
12     public GameObject splash;
13
14     [Header("Player")]
15     public GameObject playerPrefab;
16     private GameObject player;
17     private bool gameStarted = false;
18
19     [Header("Score")]
20     public TMP_Text scoreText;
21     public int pointsWorth = 1;
22     private int score;
23 }
```

63

Before our game starts, we want to **disable** the **TMPro** **Text** so it only appears when needed. Inside the **Awake** function, add a line to set **scoreText.enabled** to **false**.

```
22
23
24     private void Awake()
25     {
26         spawner = GameObject.Find("Spawner").GetComponent<Spawner>();
27         screenBounds = Camera.main.ScreenToWorldPoint(new Vector3(Screen.width, Screen.height, Camera.main.transform.position.z));
28         scoreText.enabled = false;
29     }
30
31 // Start is called before the first frame update
```

64

Inside the **restart** function, we are going to add a line that **resets** the score when the game restarts. Set **scoreText.enabled = true** and reset the score integer to **0**.

```
114     1 reference
115     void ResetGame()
116     {
117         spawner.active = true;
118         title.SetActive(false);
119
120         splash.SetActive(false);
121
122         scoreText.enabled = true;
123         score = 0;
124
125         scoreText.text = "Score: " + score.ToString();
126
127
128         player = Instantiate(playerPrefab,new Vector3(0,0,0), playerPrefab.transform.rotation);
129         gameStarted = true;
130
131     }
132
133 }
```

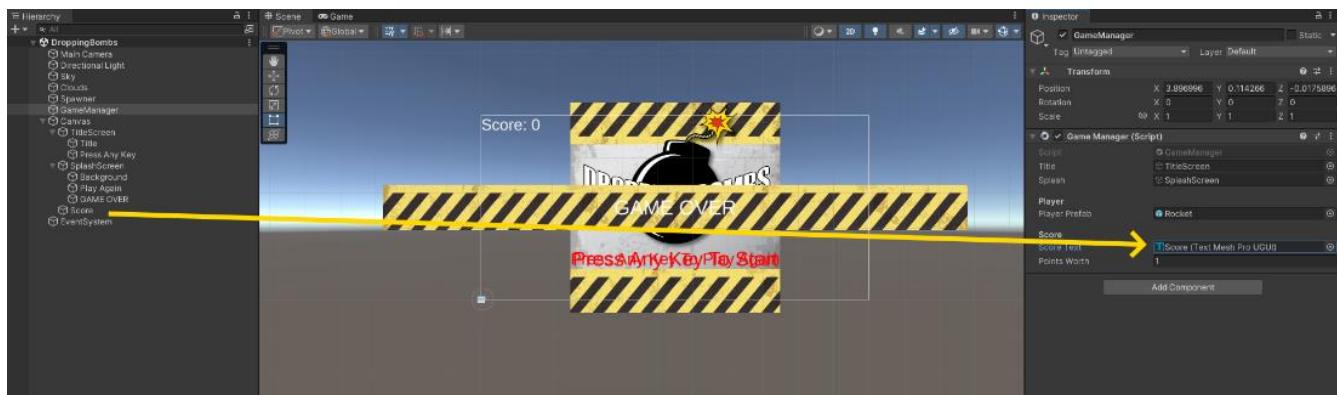
65

Finally, when our bomb gets removed for going too low, we want to gain a point. However, this will cause a problem if we lose, and the few remaining bombs get removed. To fix this, we can add a simple **if** statement to make sure that the game has **started**, and we also need to update our text to display the new value.

```
57
58
59     var nextBomb = GameObject.FindGameObjectsWithTag("Bomb");
60
61     foreach (GameObject bombObject in nextBomb)
62     {
63         if (bombObject.transform.position.y < (-screenBounds.y - 12))
64         {
65             if (gameStarted)
66             {
67                 score += pointsWorth;
68                 scoreText.text = "Score: " + score.ToString();
69             }
70
71             Destroy(bombObject);
72         }
73
74     }
75
76 }
```

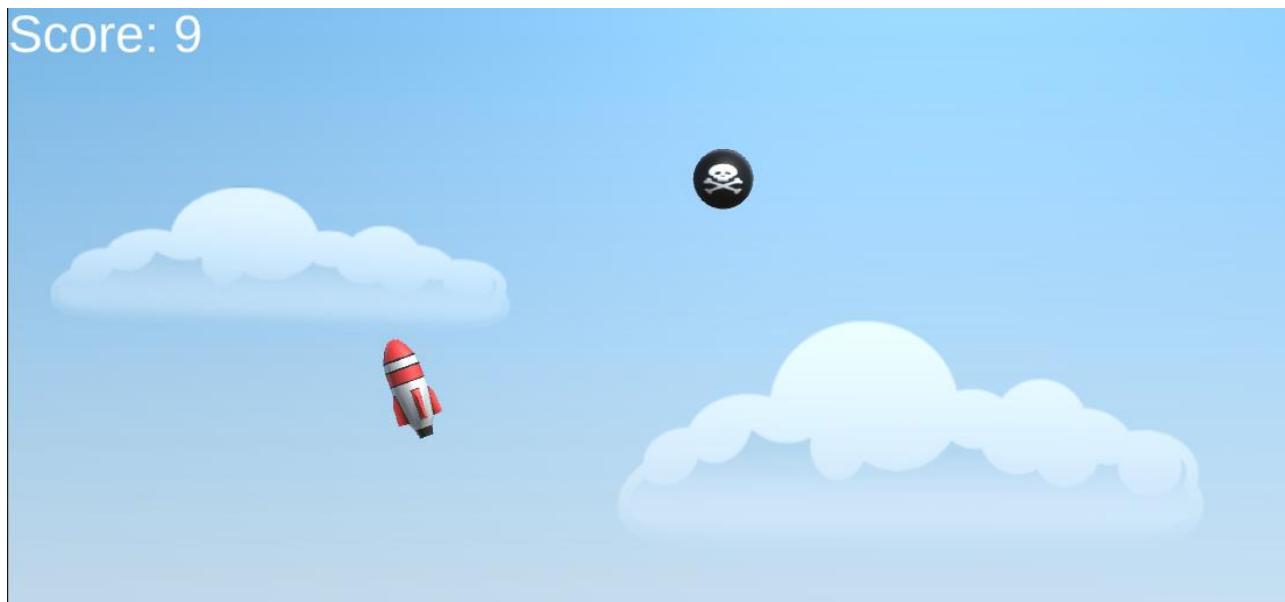
66

Almost done, let's head back into **Unity** and link our text to the **TMP** slot in the script. **Save** your project.



67

Now play your game. In the next section, there will be steps adding some more polish to finish the game!



Particle Systems and Player Preferences

Particle systems are great ways to simulate fire, smoke, water and other “fluid” visual effects. As the name implies, the system is making something out of a collection of smaller pieces and is ideal for explosions, fireworks, and even feedback for picking up power-ups in a game.

As with the other sections in this book, we’re going to cover some of the essential information that you need to know to use these systems. The systems will be covered in more detail in future books.



The Joy of Particle Systems

In Unity, most objects are represented as “solids,” rigid and well-defined. However, a solid is only one possible state of matter - liquids and gases are also commonly encountered in the world. Since liquids and gases are less rigid and well-defined than solids, a different system is required to simulate them in Unity. This section introduces the Unity particle system and some of the ways that it can be used.

What is a Particle System?

A particle system creates and displays a collection of simple images or objects as part of a whole. Each particle has its own behavior, but when viewed with other particles, it creates the illusion of smoke, fire, or many other possible effects.

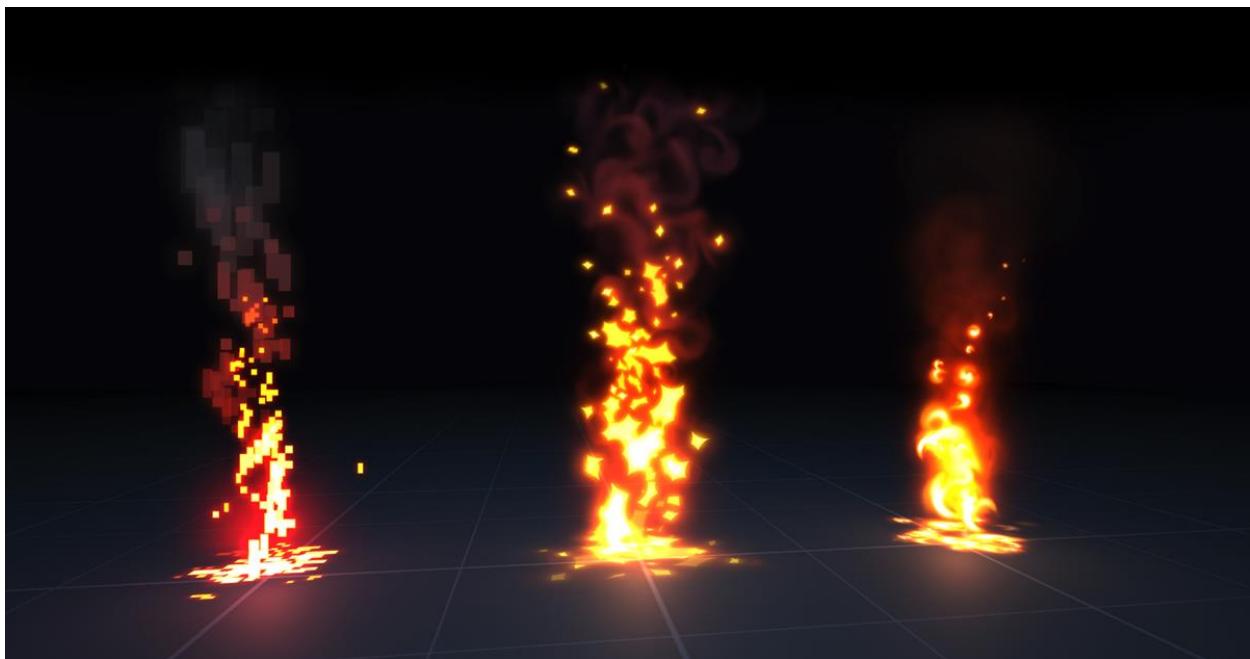


Image Source: <https://assetstore.unity.com/packages/vfx/particles/fire-explosions/stylized-fire-fx-i-67379>

Particles and the Environment

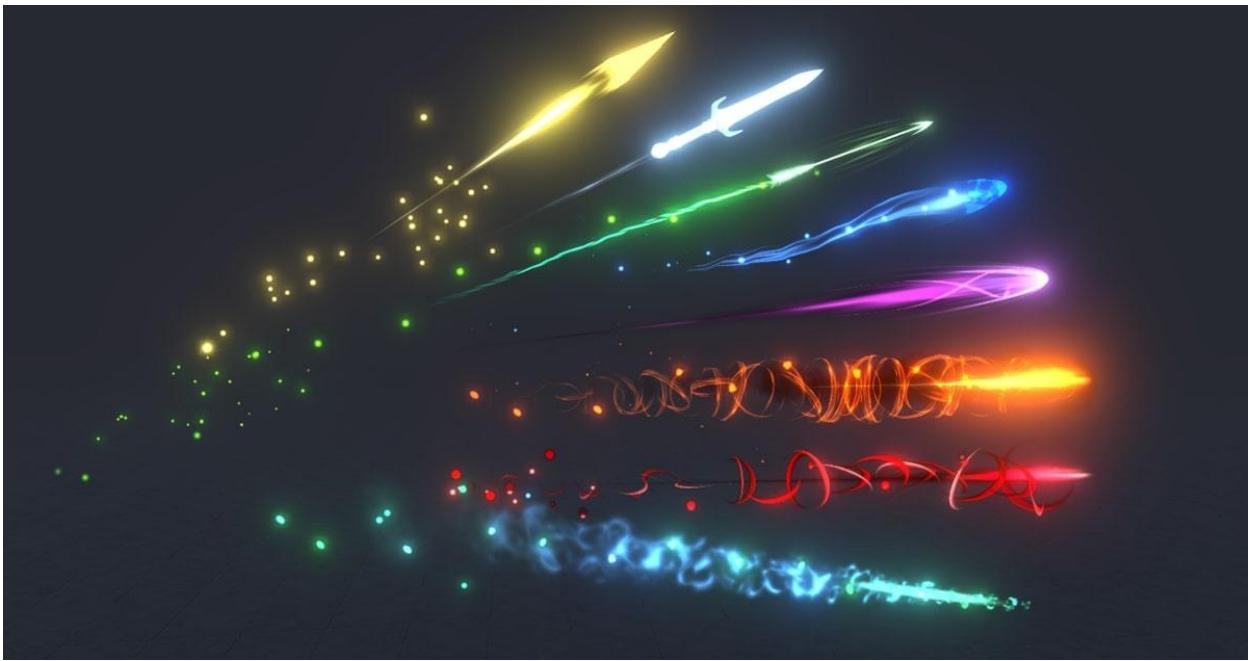


Image Source: <https://80.lv/articles/creating-projectiles-in-unity/>

Particles can be set to interact with (or being interacted by) the world in which they are set, or they can ignore it entirely. Particles can bounce off solids, be affected by wind and/or gravity, and can even act as triggers for other events.

Using Particle Systems

Since particle systems can be used to simulate so many things, both real and imagined, they have a lot of settings and sometimes you will find yourself making subtle changes to get the look just right. Even so, a particle system is just like any other GameObject in Unity and can exist on their own or be attached to other objects and can even be controlled by scripts. This section will introduce you to some of the most essential aspects of using particle systems.

Activity 10: Dropping Bombs Part 4

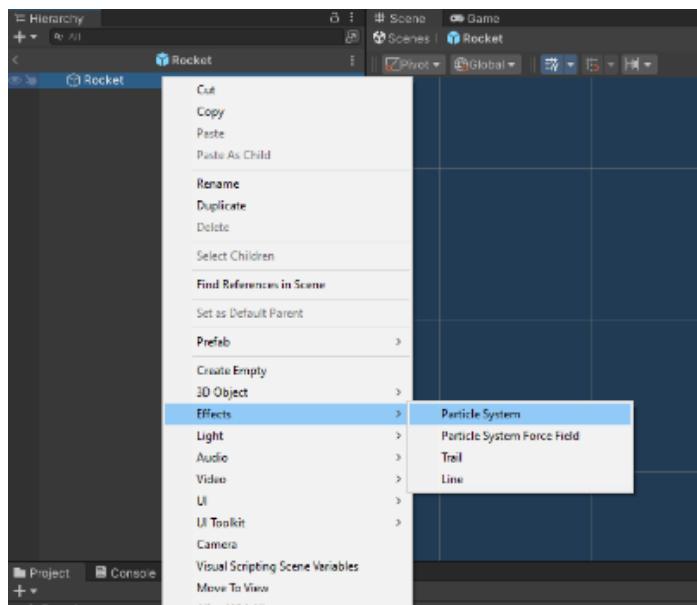
The Dropping Bombs game has come a long way since creating the simple cubes and spheres at the beginning of this book. In this activity, you'll make your game even more exciting with fire and explosions!

- 1 Before making additional changes to your project, it's important to make a backup of the entire game. In the **Projects** panel, make sure that the **Assets** folder is selected. Then click on the **Assets** tab at the top of the screen and select **Export Package**.

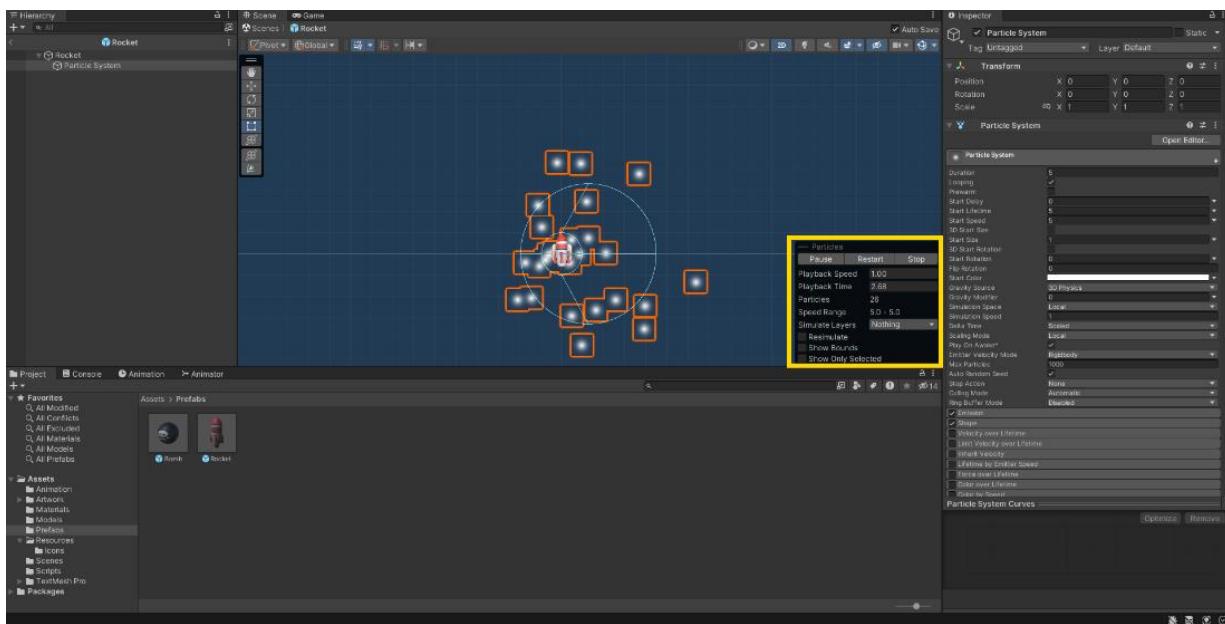
Make sure that all of the assets are selected before clicking the **Export** button. Give the package a name like **JS-DroppingBombsPart3**.

- 2 Somehow, the rocket is flying without any thrust. Which might be great for the environment, but it certainly doesn't look like a rocket should. Let's add a particle system to simulate rocket thrust.

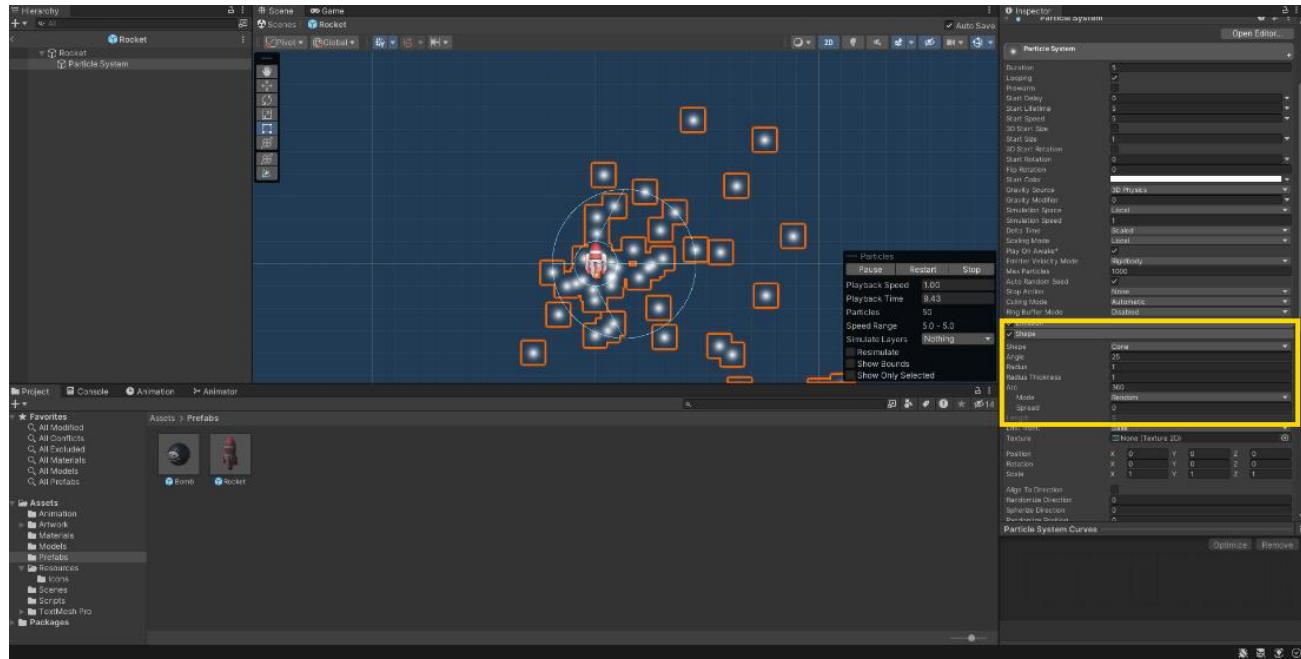
Find the **Prefabs** folder in the **Project** panel and click on the **Rocket** to edit it. In the **Hierarchy** panel, right-click on the **Rocket** and select **Effects**, then select **Particle System** to add it to the **Rocket**.



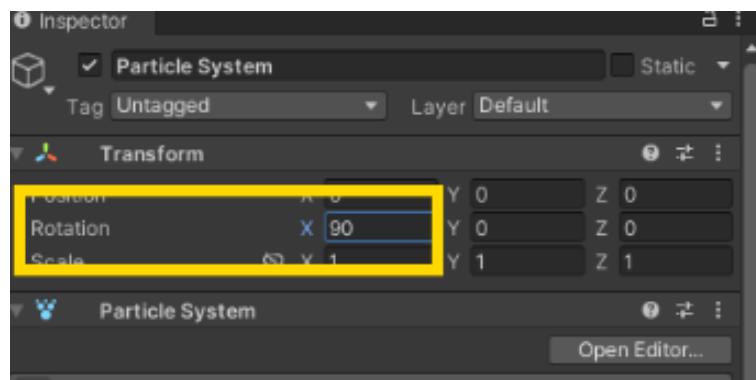
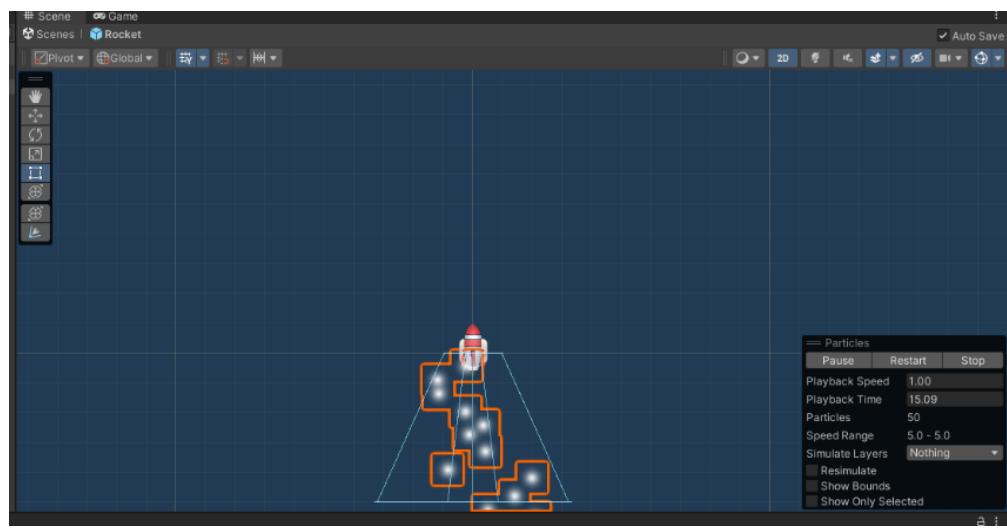
3 When added, the **Particle System** gets to work right away by showing default particles spreading away from the **Rocket**. In the corner of the scene, there's a control panel that allows you to control the particle system and provides you information on what the system is doing. Let's leave it alone for now.



4 The particles seem random, but they're actually confined to a shape which you cannot see. In the **Inspector**, scroll down until you see the **Shape** menu and click on it to make it expand. Now you can see the cone that the particles are coming from. It appears that they're coming from the back side of the rocket instead of the nozzle.

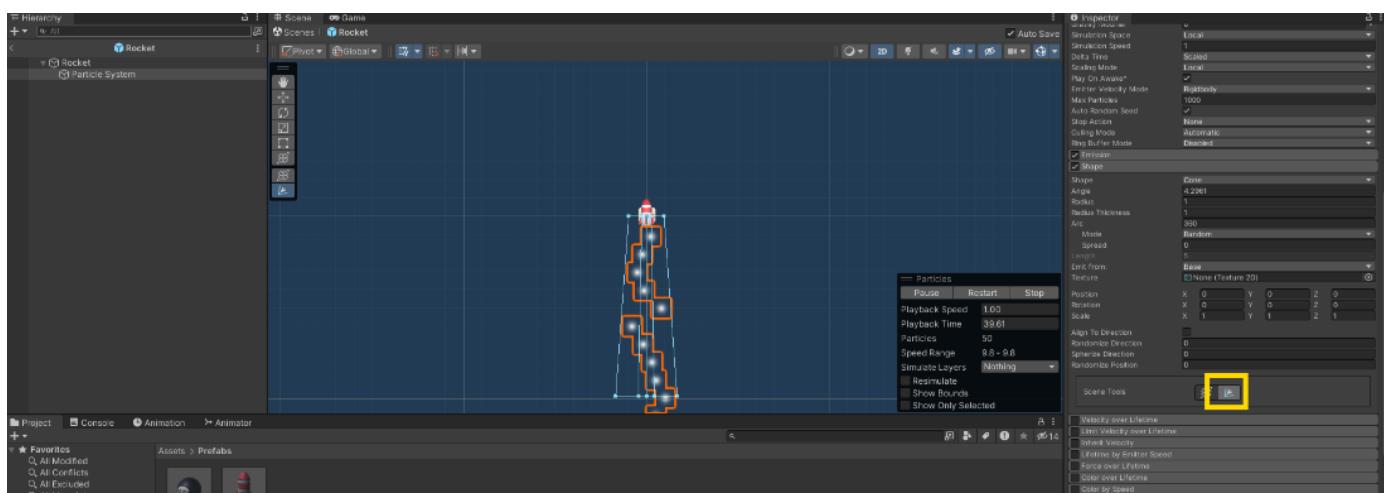


- 5** Let's rotate the cone so the particles emerge from beneath the rocket. In the **Transform** component, change the X rotation by **90** so that the cone is facing down. (You might need to use a rotation of **-90** or even **180** to get the desired effect.)

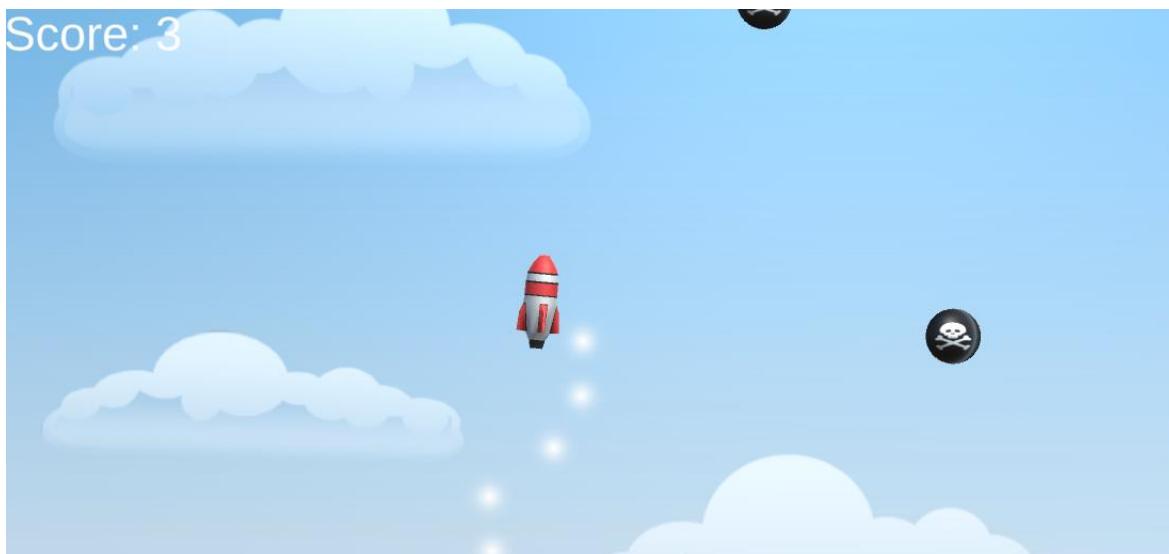


6 The cone seems a bit wide at the moment. Fortunately, at the bottom of the **Shape** component are tools to let you adjust it. The first icon lets you adjust the width at the top and bottom of the cone, while the others work just like the tools in the **Scene** editor. Use the tools to adjust the cone and move it so that it is at the bottom of the rocket as shown below.

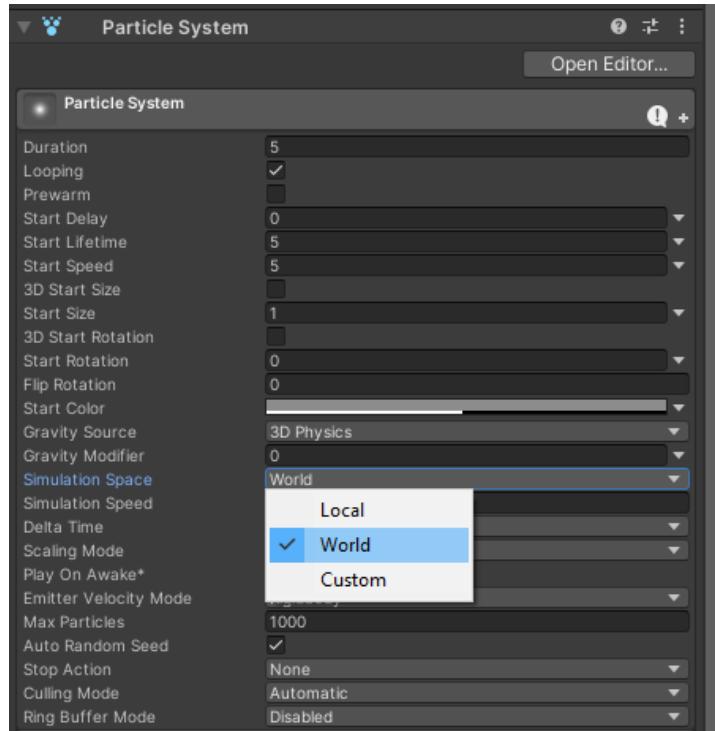
You want the button that looks like a cone being stretched!



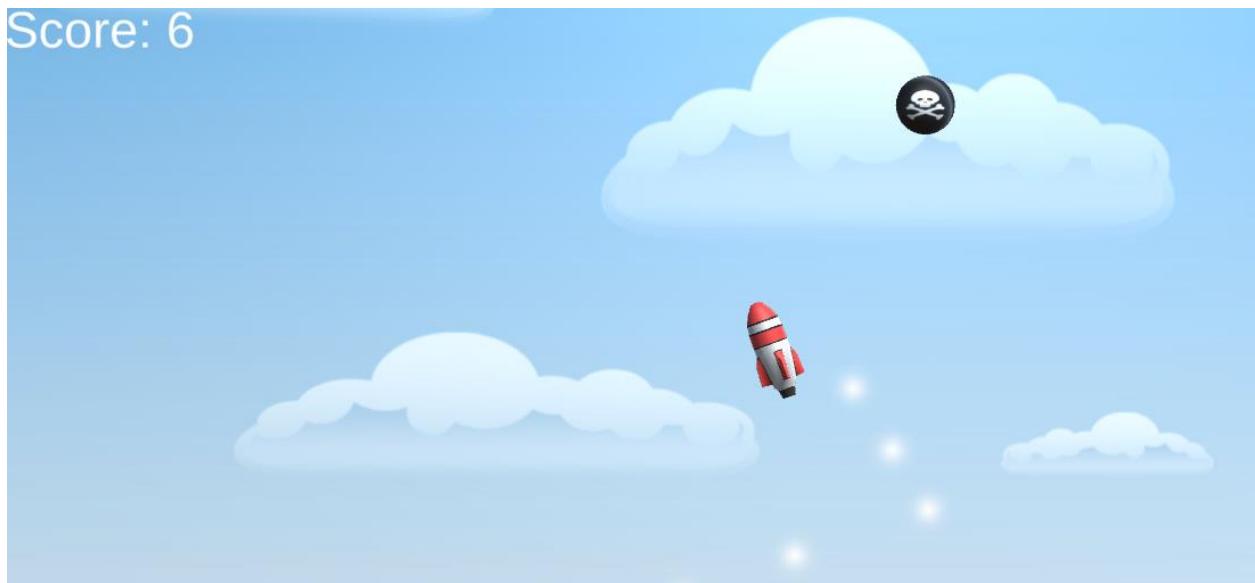
7 Click the **Play** arrow to see how the particles look in the game. They're coming from the rocket, but when the rocket moves, the whole particle system moves with it. Stop the game.



- 8** While we want the particles to originate at the rocket nozzle, we expect them to behave like they're part of the world that the nozzle sends them out into. In the Inspector, find Simulation Space and change it from Local to World.



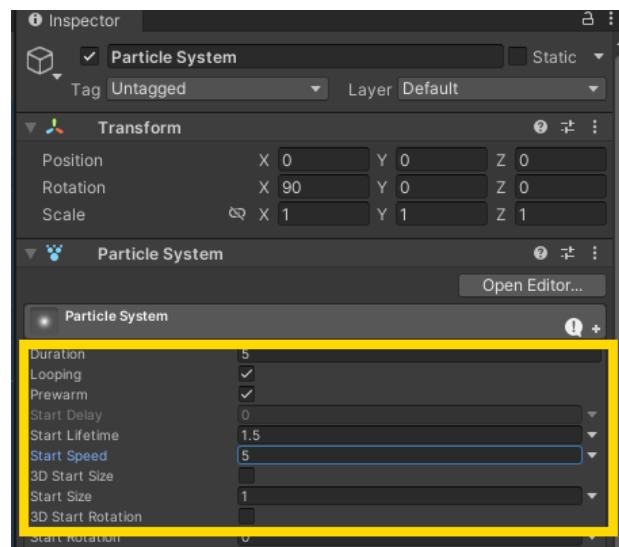
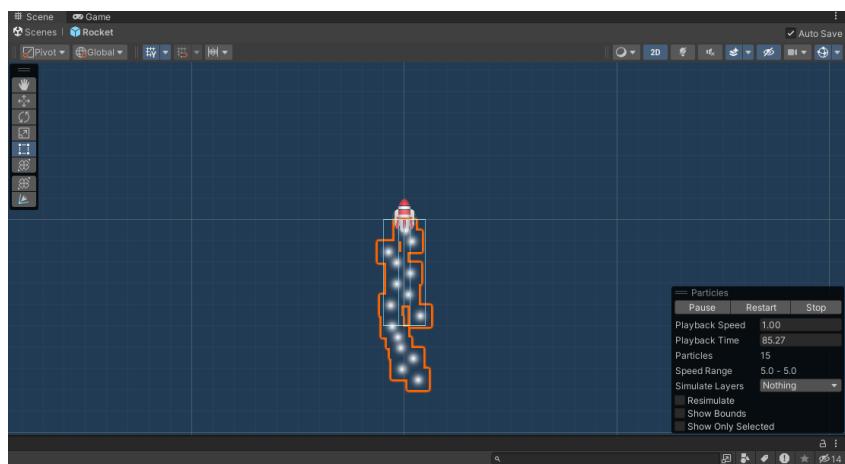
- 9** Start the game again. Now you can see that the particles behave more as expected. Stop the game before continuing to the next step.



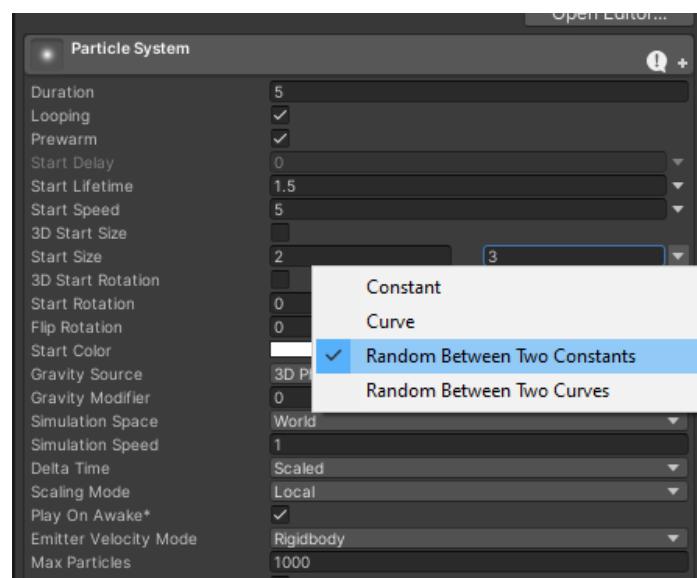
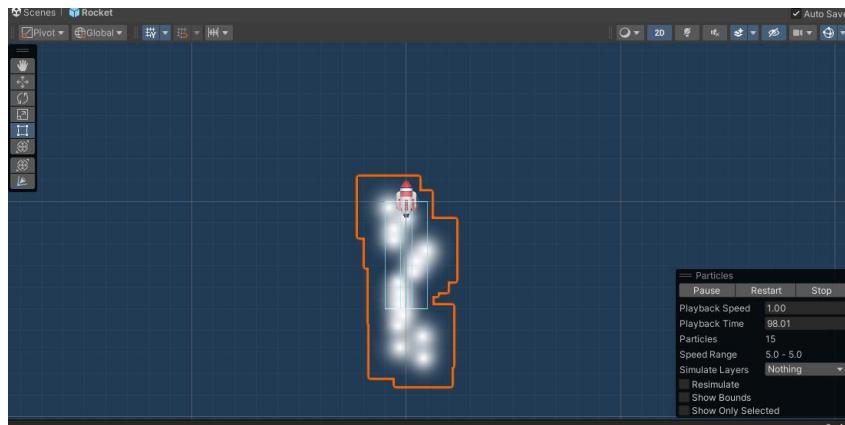
10 You may have noticed that it takes a moment for the particles to get started when the game begins. There is a setting called **Prewarm** that animates the particles as if they had been running for a while and not starting “cold.” In the **Inspector**, make sure that the box for **Prewarm** is checked. Also, let’s make the trail a little shorter by reducing the **Lifetime** of the particles.

Try 1.5.

You may also wish to change other properties. If you haven’t, set the **Start Speed** to 5.



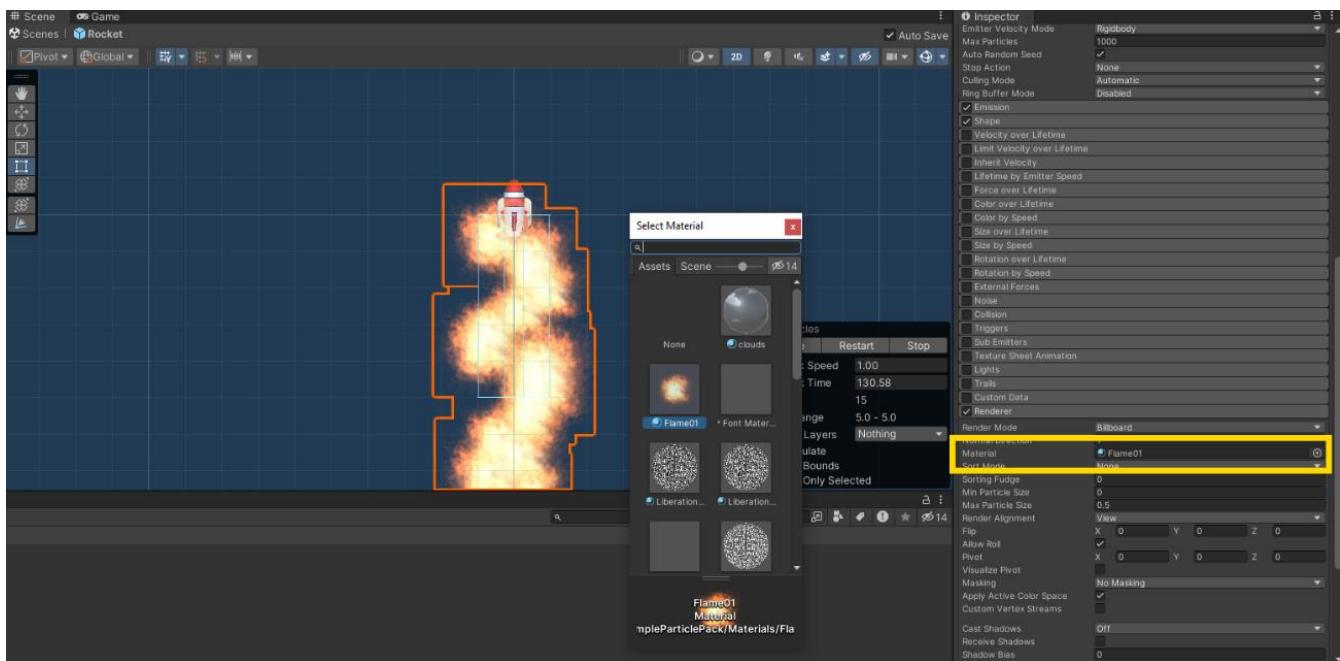
11 We can add a little variety to the particle by randomizing the size of the particles. Click on the triangle to the right of the slot for **Start Size** and choose **Random Between Two Constants** and pick a range of **2 to 3** as shown.



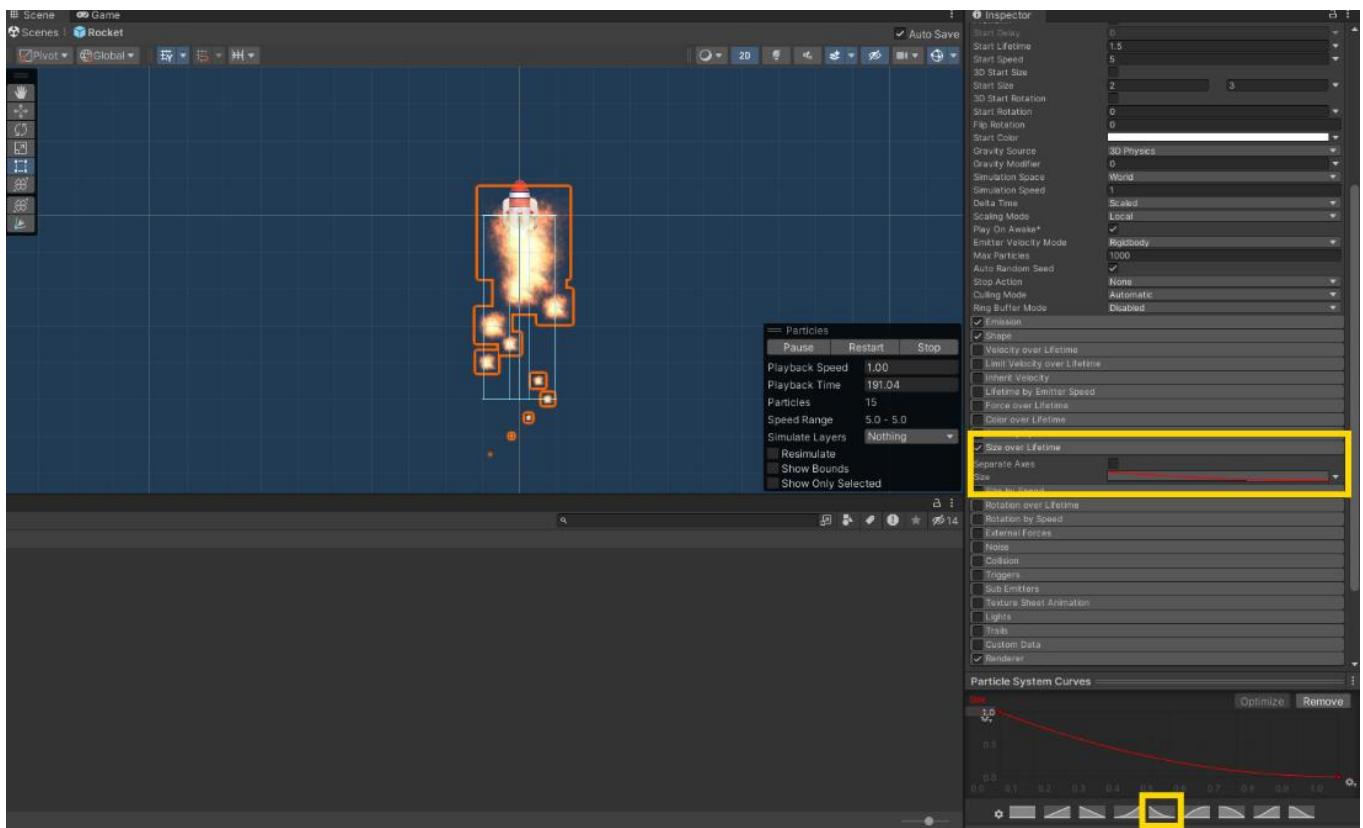
All of the properties in the Particle System have options for constant or curves just like with Start Size.

12

The rocket thrust is looking better, but still looks odd. Let's use a different particle. Click the **Assets** tab and select **Import Packages** and load **Activity 10 - Flame.unitypackage**. At the bottom of the **Particle System** component is a property called **Renderer**. Click on it to expand it and select a new **Material** by clicking on the circle to the right of the **Material** slot. From the menu that opens, select **Flame01** (which you just imported).

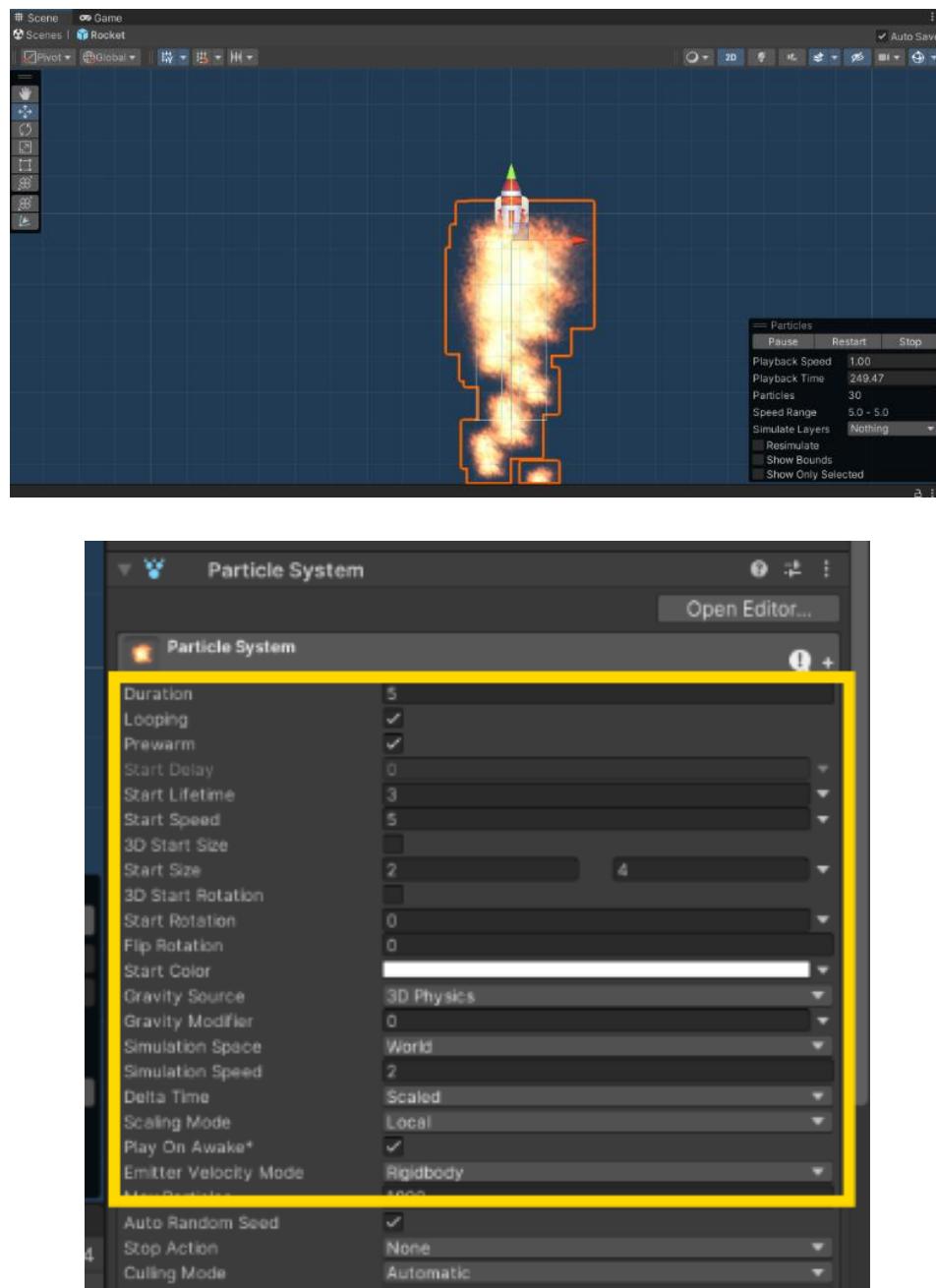


13 In reality, the rocket thrust would taper off as the flames got farther and farther from the fuel source. The **Particle System** has a component to simulate that. Look for the component called **Size Over Lifetime** and check the box to enable it. Then click on the window next to **Size** to access the curves for that component. This opens a new window at the bottom of the **Inspector**. Select a curve that starts high and fades to nothing.

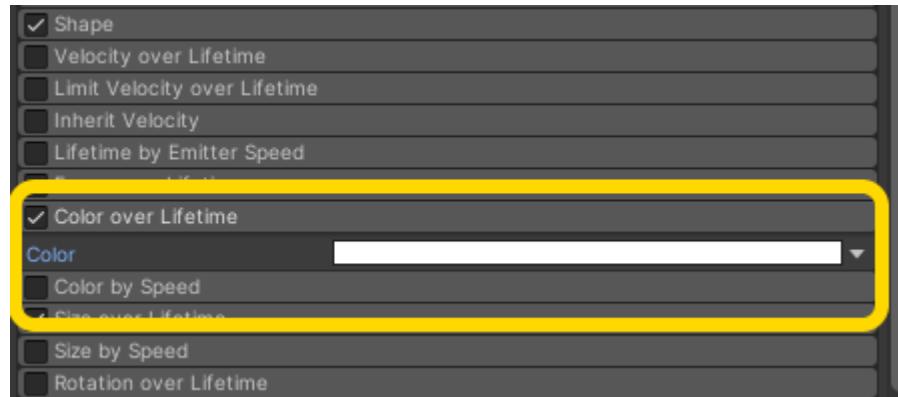
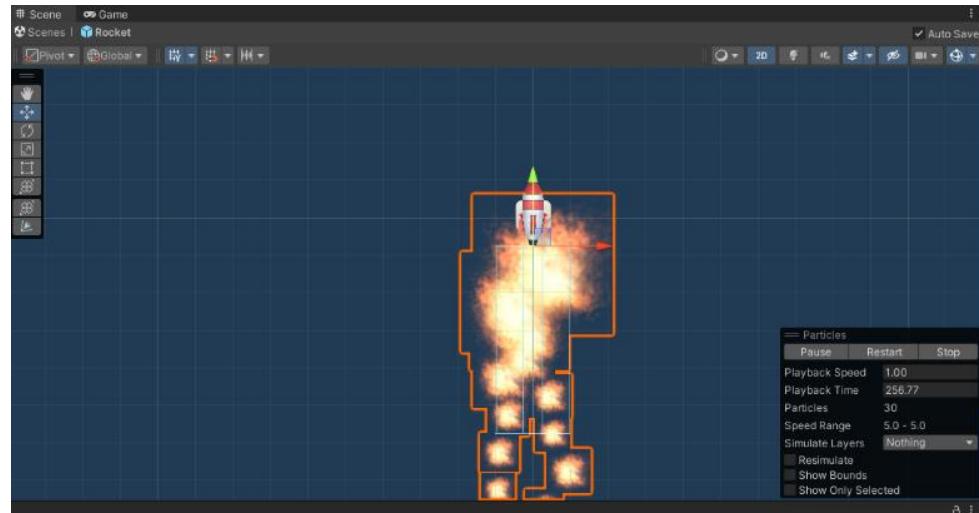


14 Let's adjust some of the numbers to get the thrust where we want it. If we want a longer thrust, increase the **Start Lifetime**. If we want it to be denser, decrease the **Start Speed**. If we want a more powerful flame, increase the **Simulation Speed**.

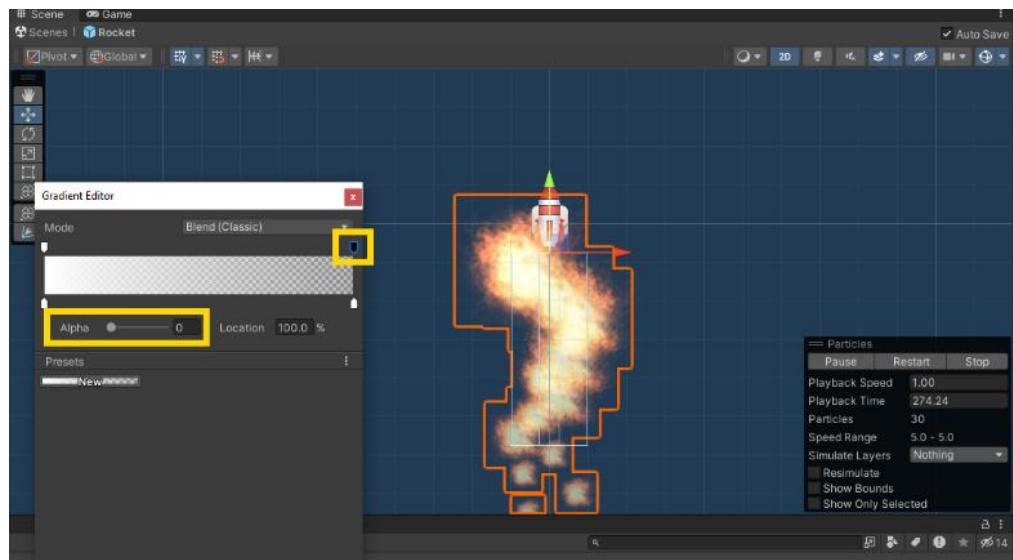
You can use whatever numbers you want, see below for an example.



15 The flames abruptly vanish when they reach the end of their lifetime. Instead, let's make them more transparent over time. In the **Inspector**, find **Color Over Lifetime** and check the box to enable it. Then click on the **Color** window.



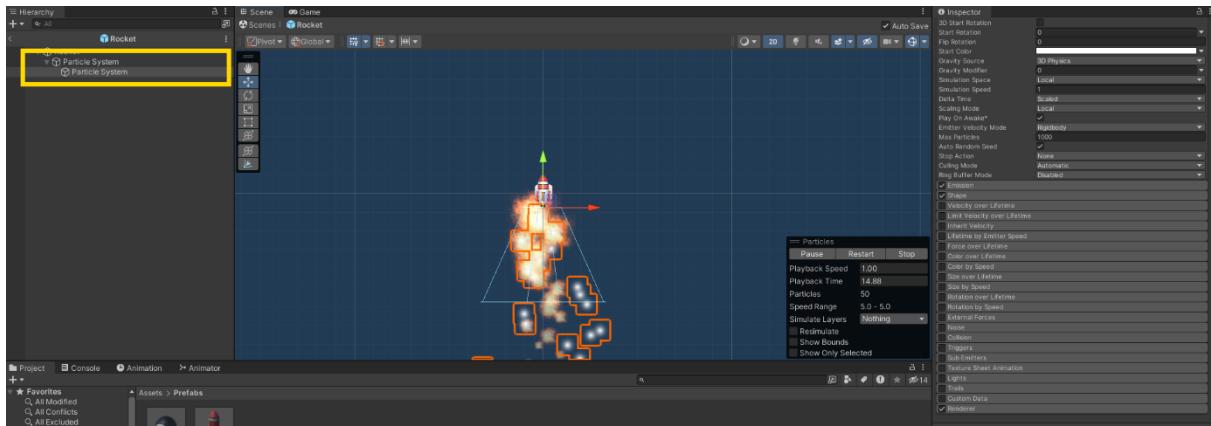
16 Clicking on the color opens a new window for adjusting color. It works a lot like the timeline we used for the animations, beginning at the left and ending on the right. If we wanted to, we could have the particle gradually change color over time. Right now, we just want to change the transparency which is controlled by the arrows at the top. The left arrow is fine as it is. Let's click on the arrow at the top right. The panel at the bottom says the alpha transparency is 100% (completely opaque). Change it to 0 so that the color is completely transparent at the end.



17 Let's test the thrust by playing the game. Not too bad, right? Though real rocket thrust would have some smoke. Stop the game before continuing.

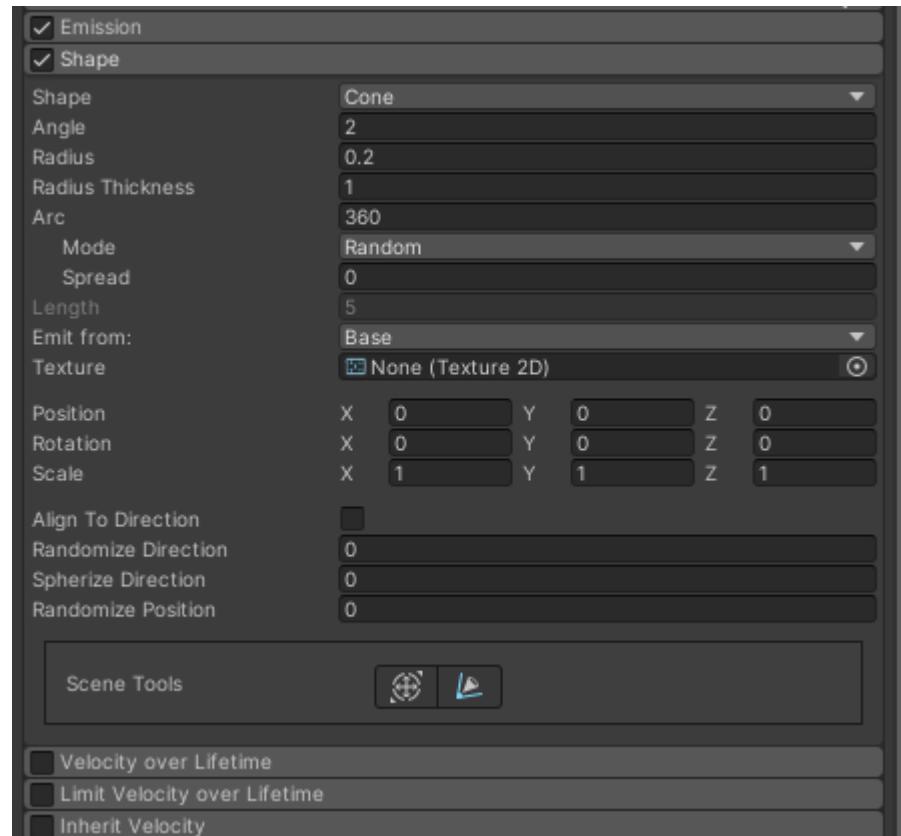
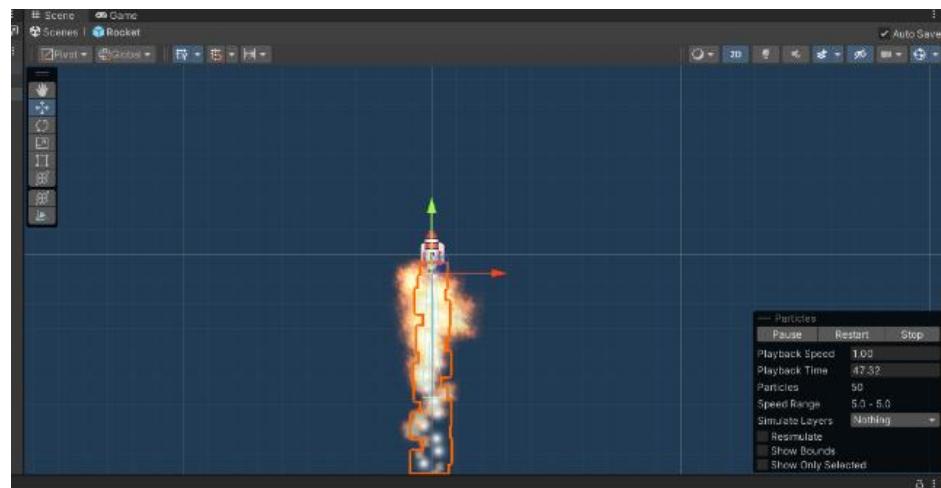


18 In the **Hierarchy**, right-click on the **Particle System** to add a new particle system inside the first one (select Effects and click on **Particle System**). See the Pro Tip below if the new Particle System isn't playing.



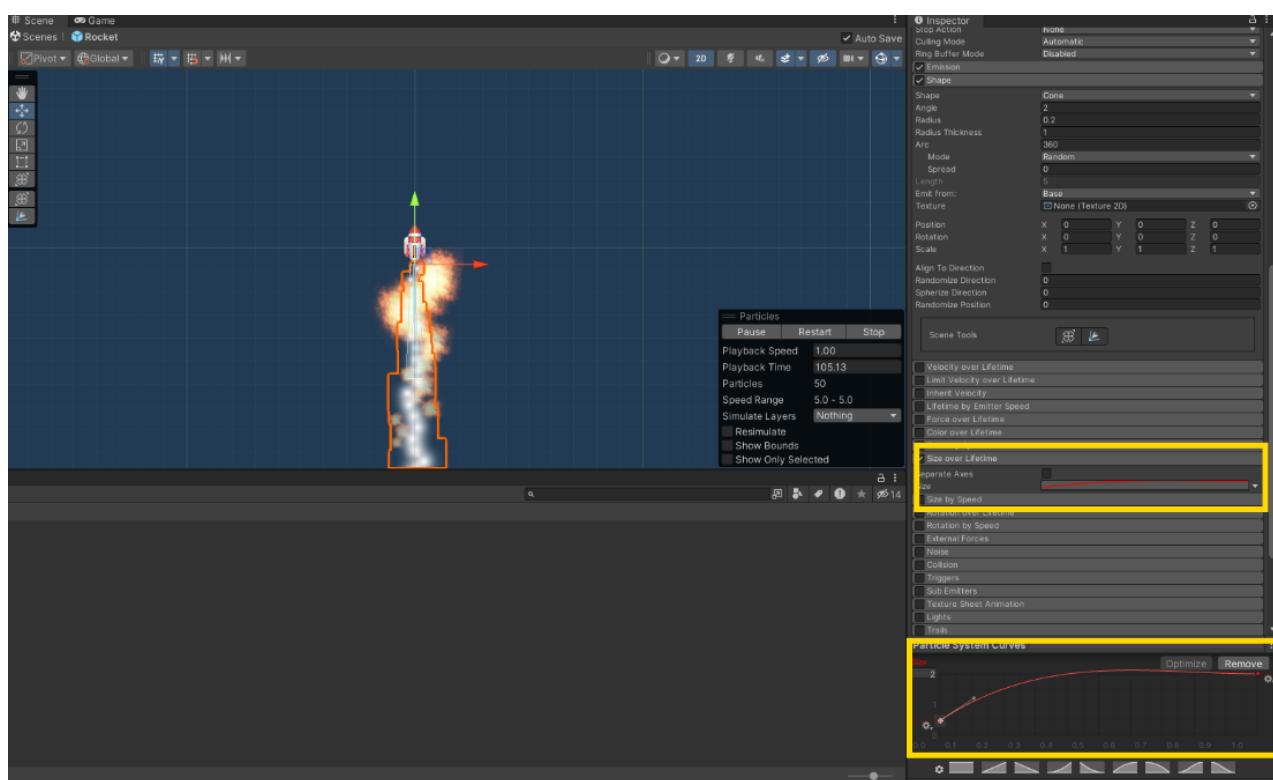
Sometimes, when adding additional Particle Systems, the new system doesn't seem to work. The control panel shows 0 particles being emitted. When this happens, restart or pause the particle system and start it again.

19 The shape of the smoke should be similar to the flame. Click the **Shape** component and use the tools at the bottom to adjust the shape of the cone. Then move it so that it starts a bit further away from the rocket nozzle than the flames.

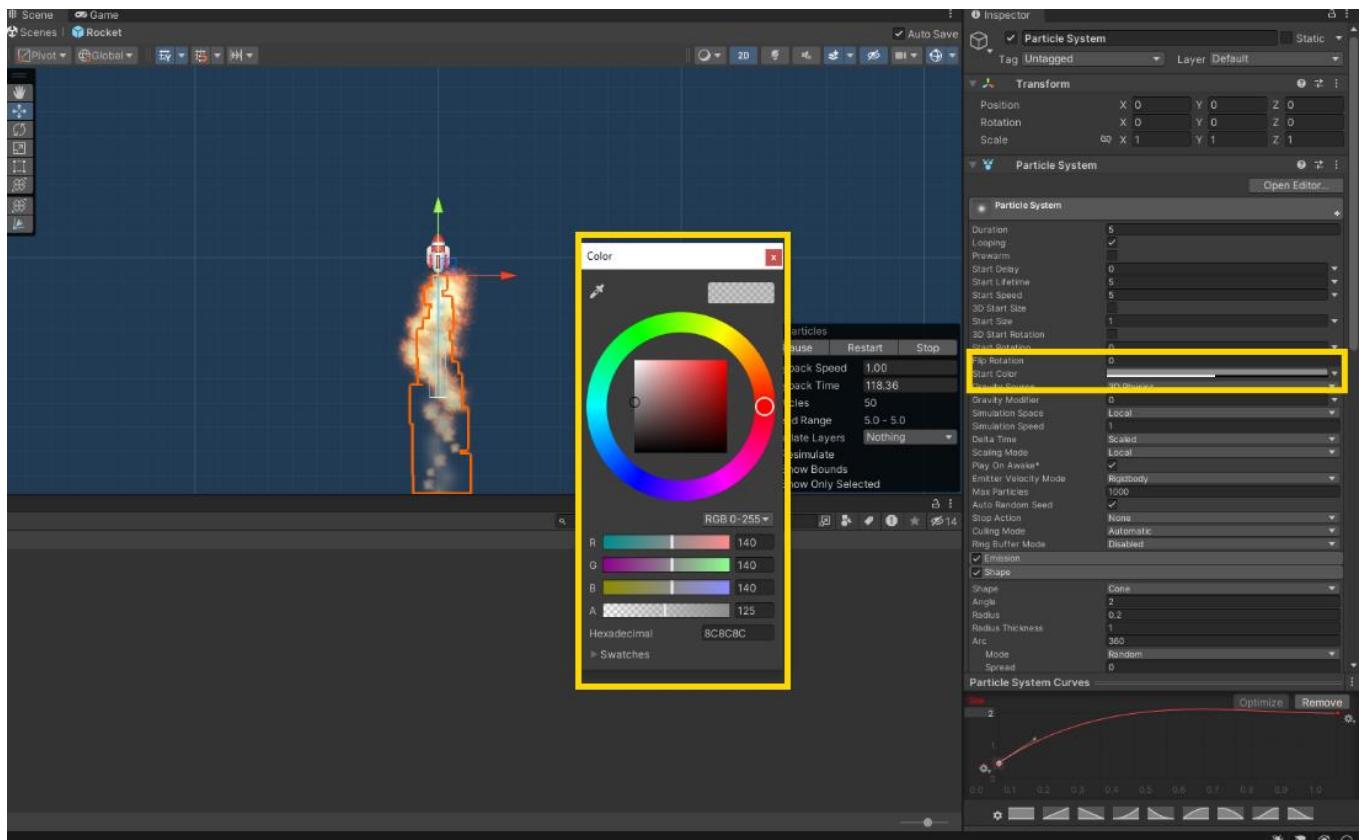


20 Over time, smoke spreads out. So go to the **Size Over Lifetime** component and enable it. Choose a graph that starts small and grows. To ensure that it starts out at a reasonable size, click and drag the left point of the curve so that it's about a third of the way up the scene.

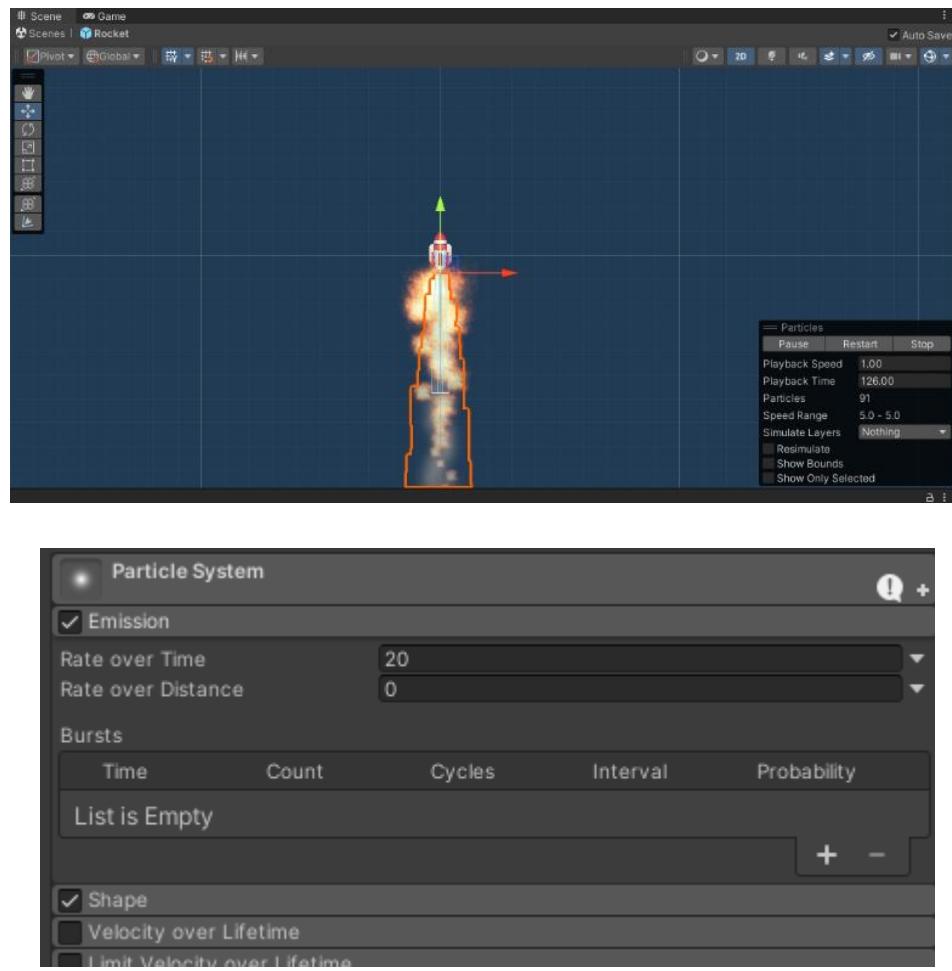
Your curve doesn't have to look exactly the same; you may wish to use a curve that you think looks better.



21 Now let's adjust the color of the smoke. Find **Start Color** in the component and click on the white rectangle to open the **Color** editing window. Drag the selector in the square to get a light grey color. Let's also make it a bit transparent by dragging the Alpha slider ("A") a bit to the left.



22 We can increase the smoke by selecting the **Emission** component and increasing the value of **Rate Over Time**. Let's double it.



23 Play your game to see how the Rocket Thrust effect looks. Do you want more fire? Darker smoke? Adjust the Particle System to get the results that you like. Don't forget to update the Simulation Space to World.

Stop the game and save your project before continuing.

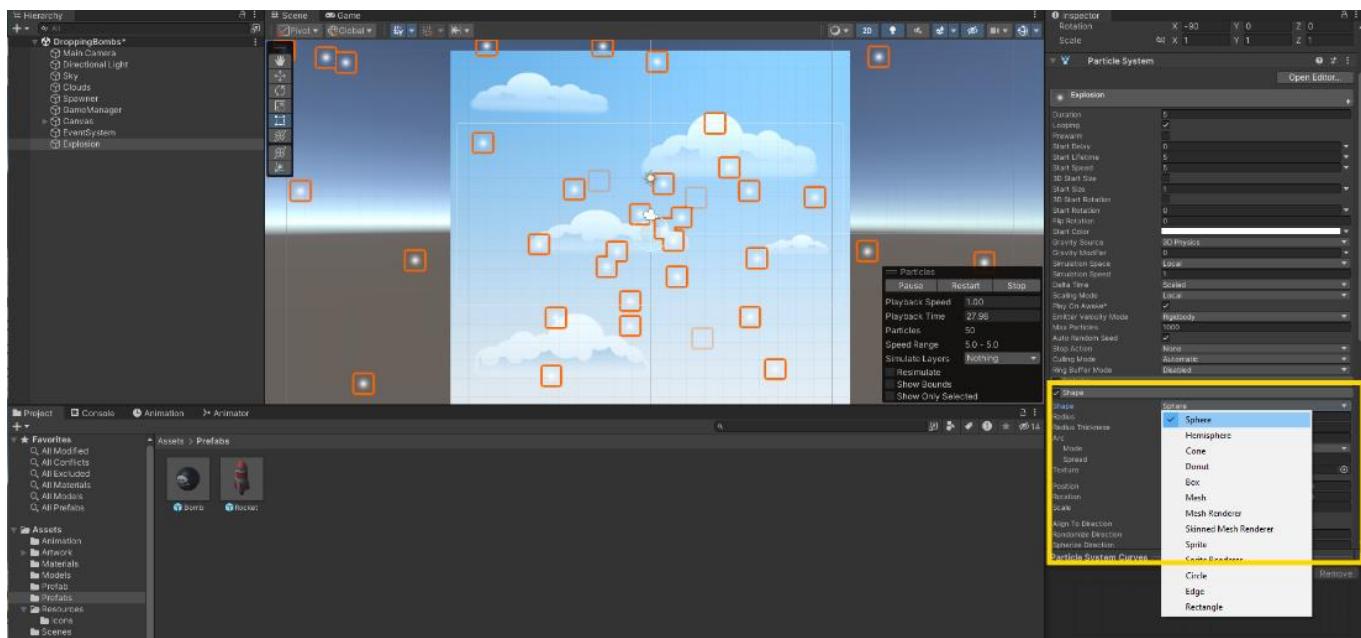


24 What the game needs now is something to happen when the Rocket collides with a bomb. In the **Hierarchy**, click **Create**, select **Effects** and then select **Particle System**. Name it **Explosion**.



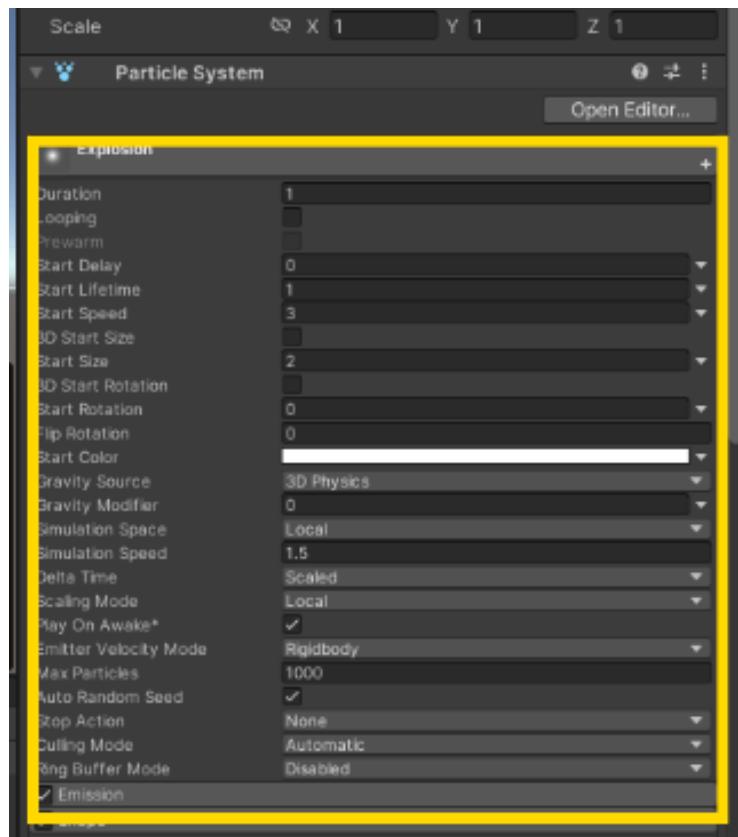
25

The default shape of the new **Particle System** is a cone, but that's not right for an explosion. In the **Shape** component, go to the **Shape** menu and select **Sphere**.



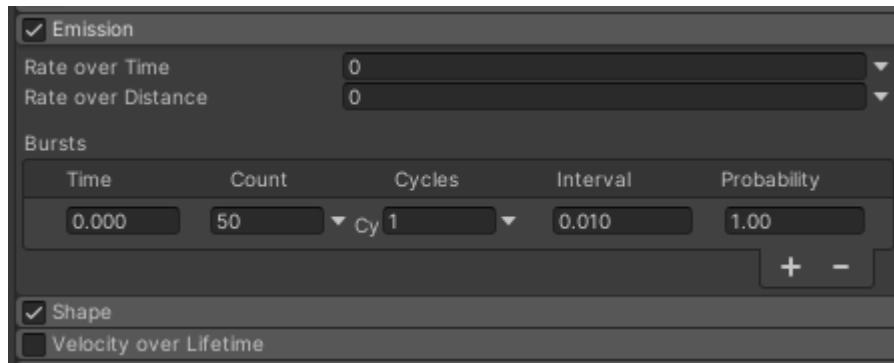
26

The explosion should only play once. Go to the top of the **Particle System** component and uncheck the box for **Looping**. Change the settings for **Duration**, **Start Lifetime**, **Start Speed**, **Start Size** and **Simulation Speed** as shown below.



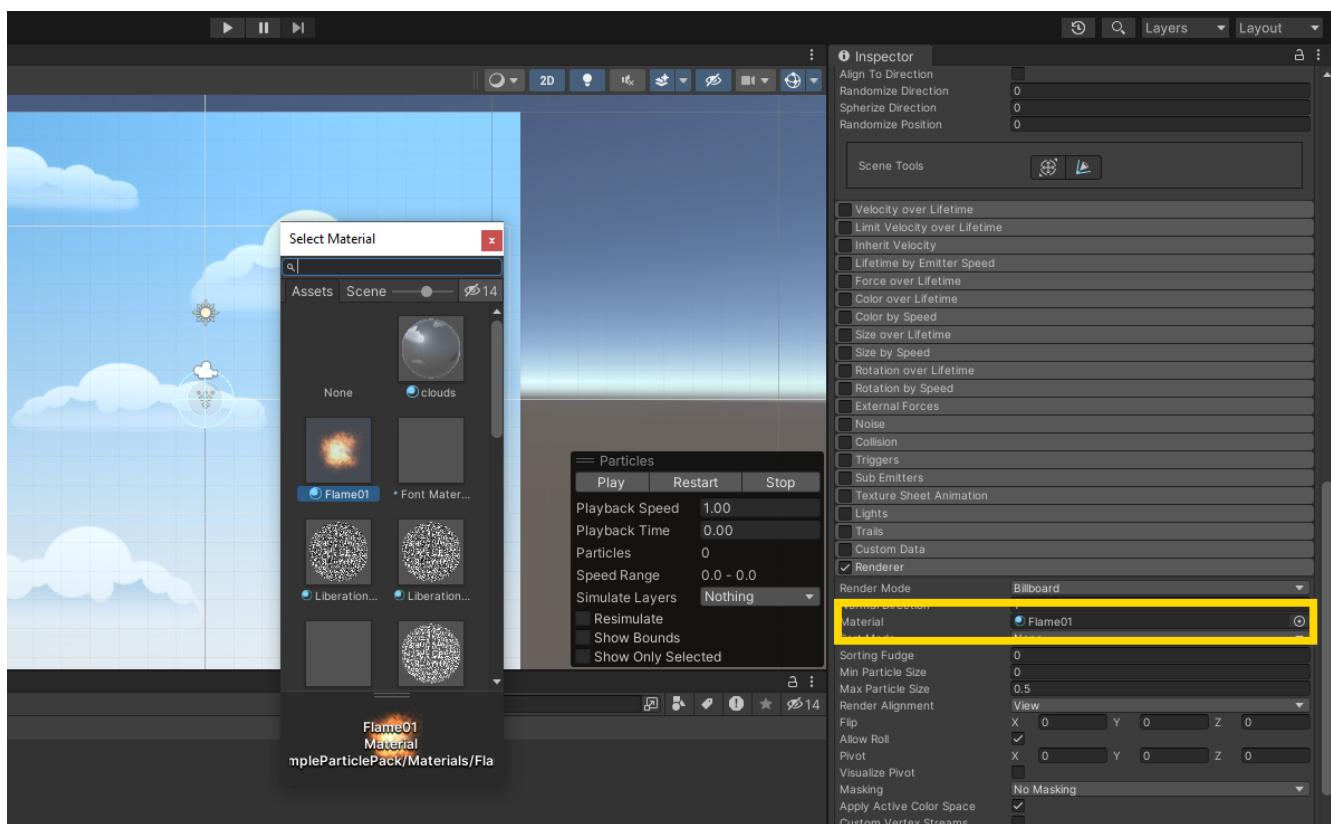
With Looping off, the Particle System no longer plays constantly in the Scene panel. From now on, you will need to press Play in the Particle System control panel when you want to preview this Particle System.

- 27** At the moment, the explosion seems a bit weak. Go to the **Emission** component, but instead of adjusting the **Rate Over Time**, add a **Burst**. Click on the plus (+) symbol at the bottom of the **Bursts** panel to add a **Burst**. Keep all of the default settings as they are but increase **Count** to **50** and reduce the rate over time to **0**.



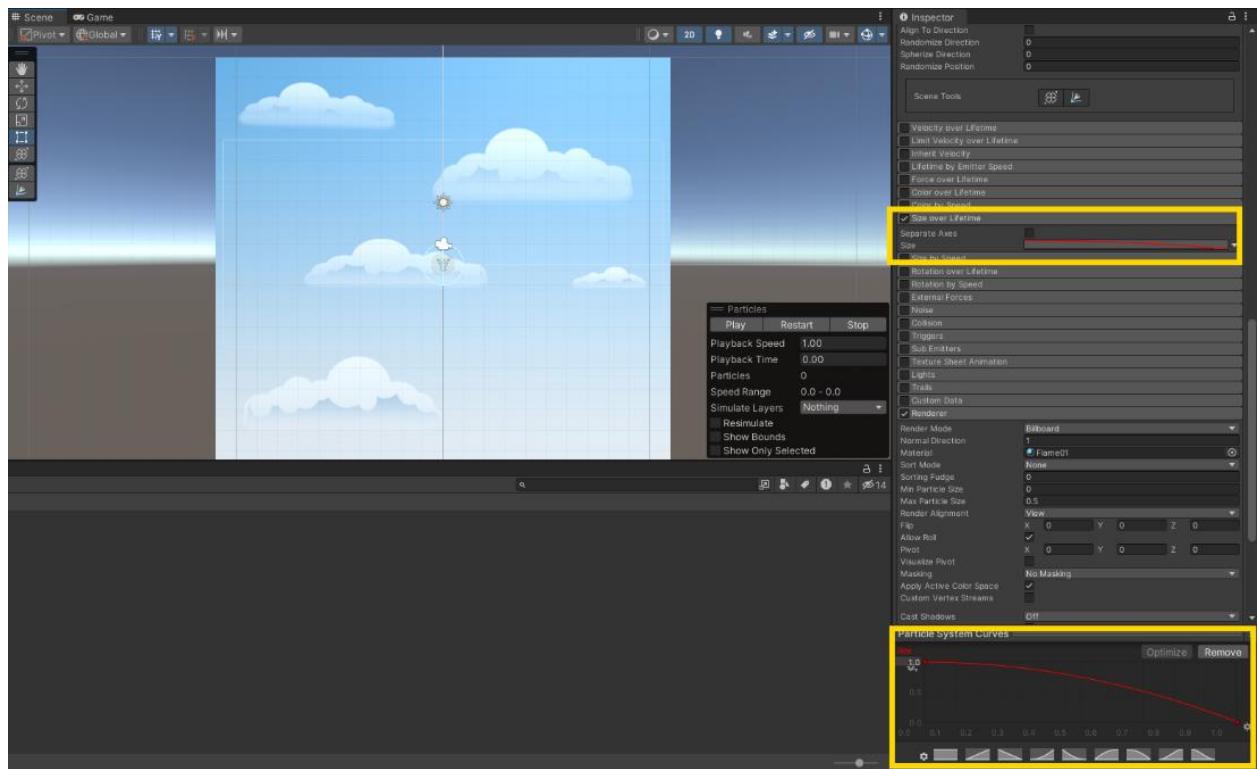
28

Go to the bottom of the **Particle System** component and expand the **Renderer**. Select a new **Material** by clicking on the circle to the right of the **Material** slot. From the menu that opens, select **Flame01** (the same as what you used for the Rocket Thrust).



29

The explosion particles should get smaller over time. Find the **Size Over Lifetime** component and enable it. Then create a curve that starts out large and eventually dwindles to nothing.



30

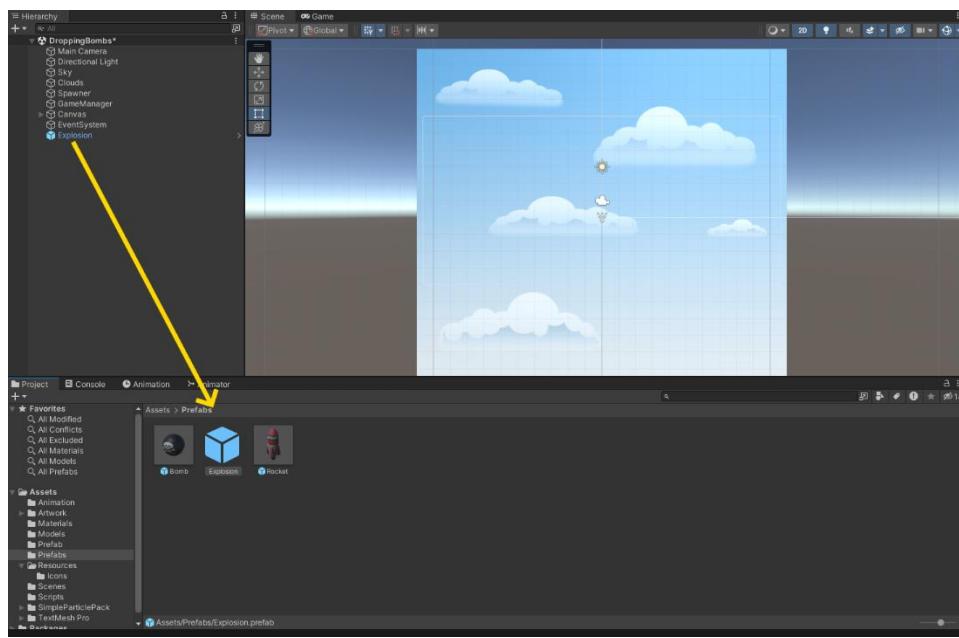
Just as you did with the rocket thrust, select the **Color Over Lifetime** component and enable it. Click on the **Color** rectangle to edit the color. Click the arrow on the top right of the color timeline and set the **Alpha** to **0**.



Let's save this setting so that from now on we can just load it when we want to change Color Over Time. Just click the **New** button and the next time you need this. Click on the **Preset** you just made.

31

Before going any further, let's drag the **Explosion** object into the **Prefabs** folder. Double-click on the **Explosion** prefab to edit it.

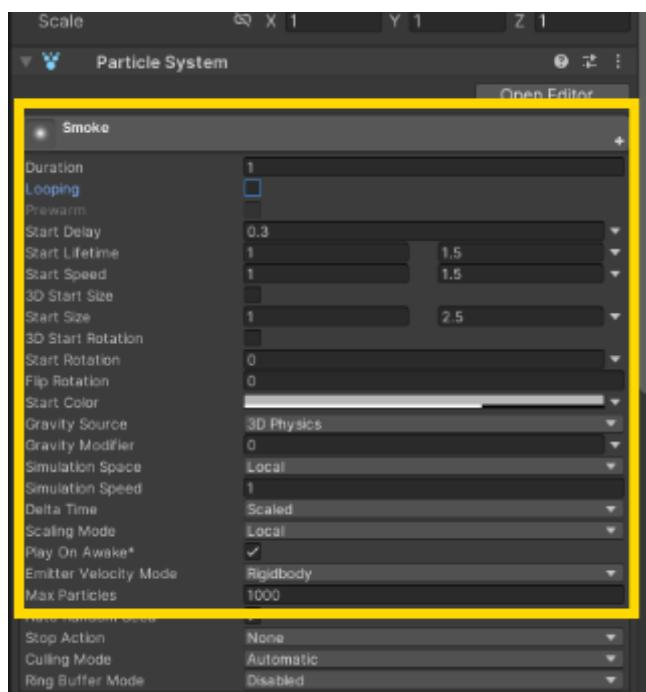


32

Just as you did with the Rocket thrust, right-click on the Explosion to add another Particle System and name it **Smoke**. Turn off looping and change the **Duration** to **1**. Change the **Start Delay** to **0.3** so that the smoke follows a little bit after the flames.

Add random between two ranges for the **Start Lifetime**, **Start Speed** and **Start Size** as shown below.

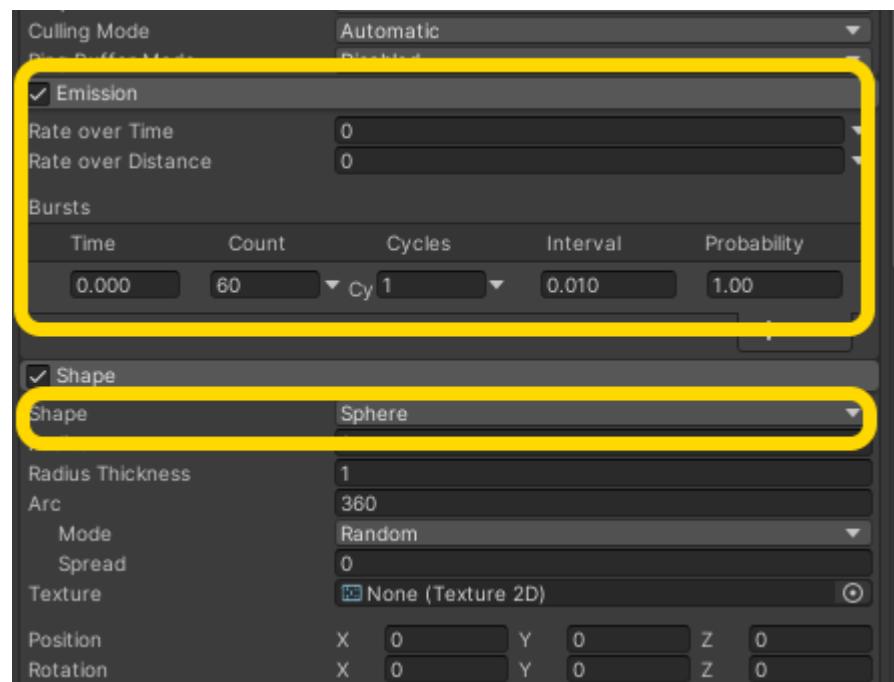
Change the color of the smoke to a medium grey with an **Alpha** of about **75%**.



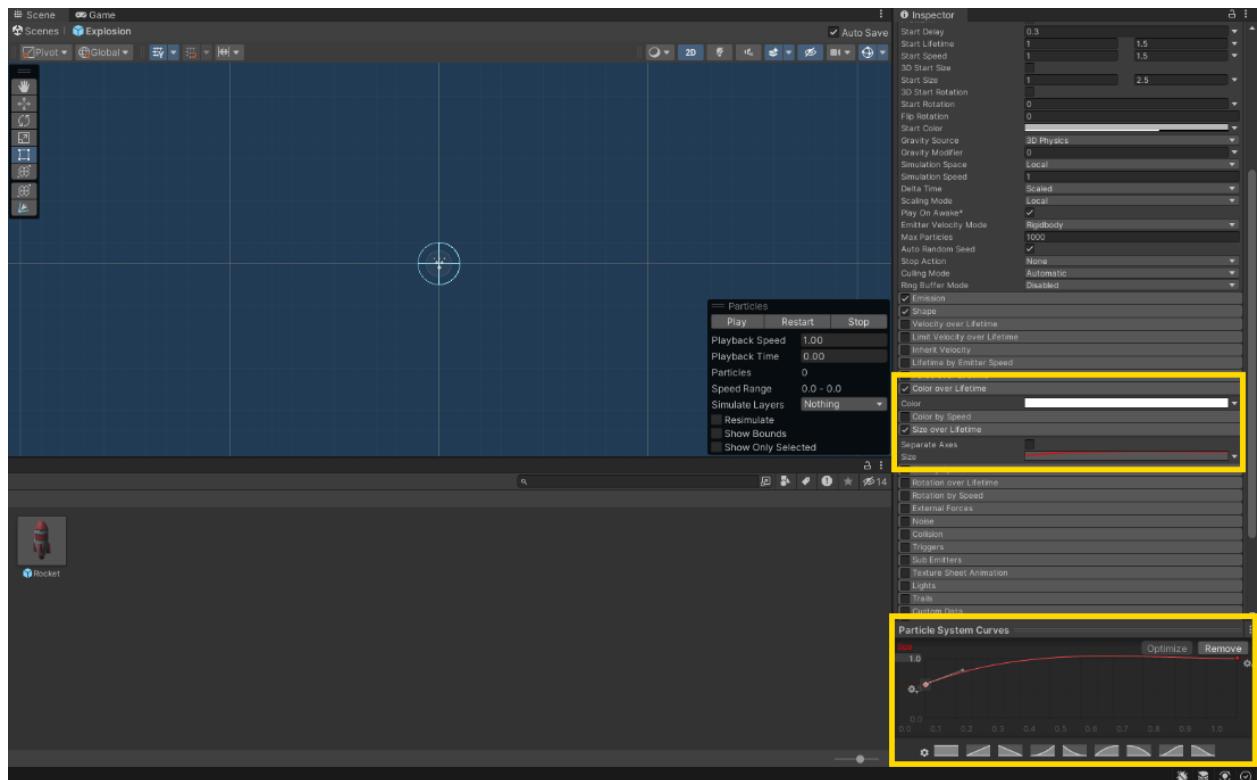
33

Just like the **Explosion**, the **Smoke** will also have a **Burst**. Expand the **Emission** component and change the **Rate Over Time** to **0**. Add a **Burst** to the **Bursts** panel by clicking on the plus (+) symbol in the lower right corner. Change the **Count** to **60**.

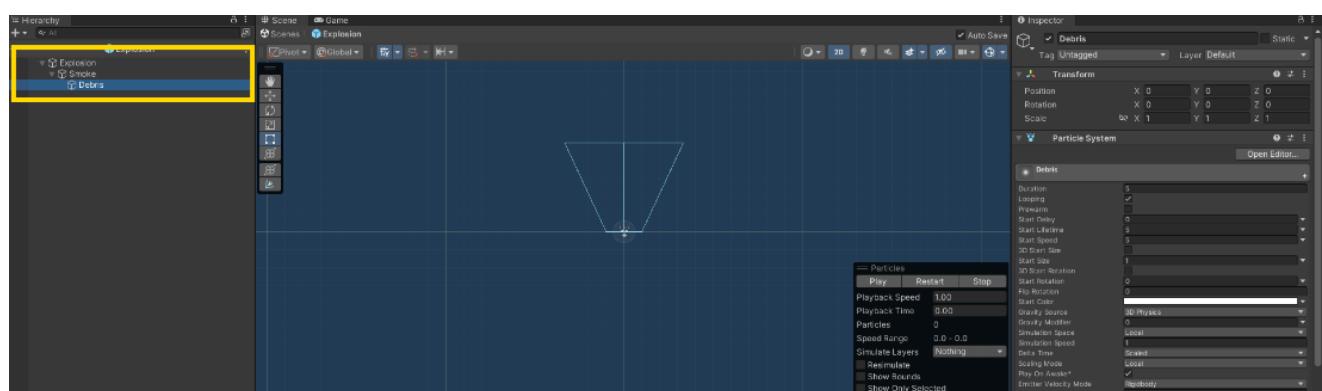
Expand the **Shape** component and change the **Shape** from a **Cone** to a **Sphere**.



34 Enable **Color Over Lifetime** to add the preset gradient that you recently made to have the smoke fade away. Enable **Size Over Lifetime** to give a gradually increasing curve to the size of the smoke. Save your progress.

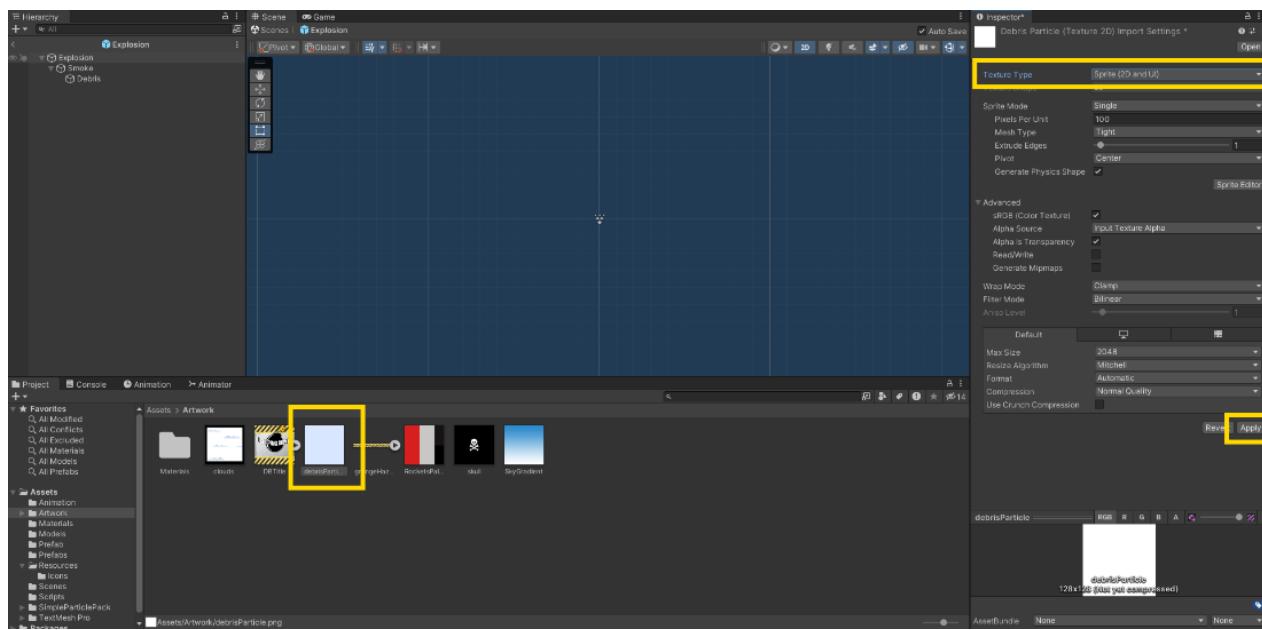


35 Right-click on the **Smoke Particle System** to add a third **Particle System** and call it **Debris**.



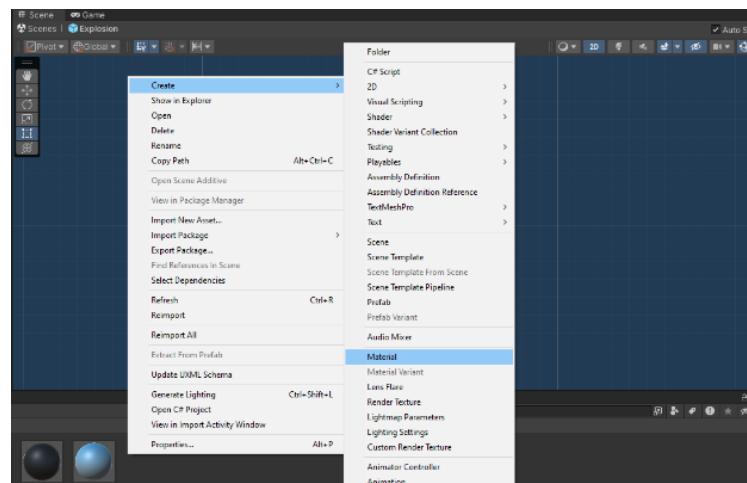
36

We need a new particle for this and this time, we're going to make it ourselves. In the **Project** panel, open the **Artwork** folder. Right-click inside the folder and select **Import New Asset** and select **Activity 10 - debrisParticle.png**. After the image is loaded, select it and then go to the **Inspector** panel where you'll change the **Texture Type** to **Sprite (2D and UI)**.

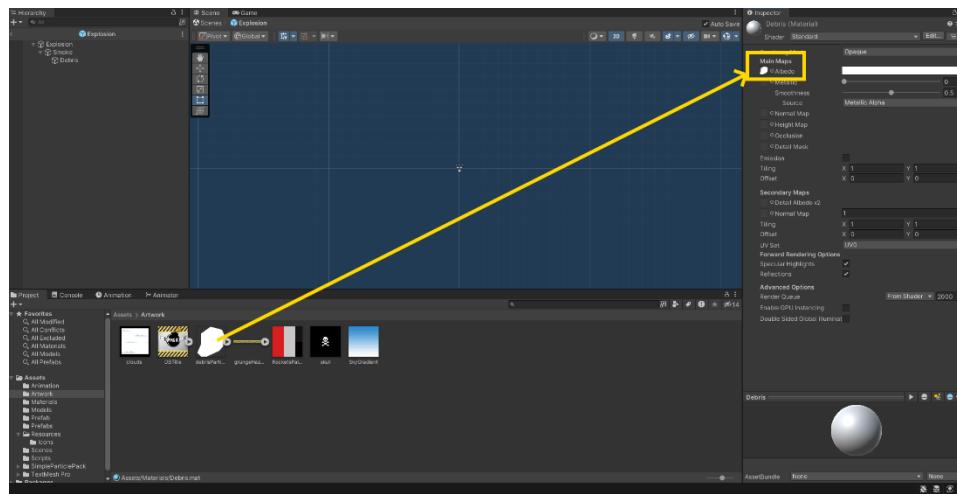


37

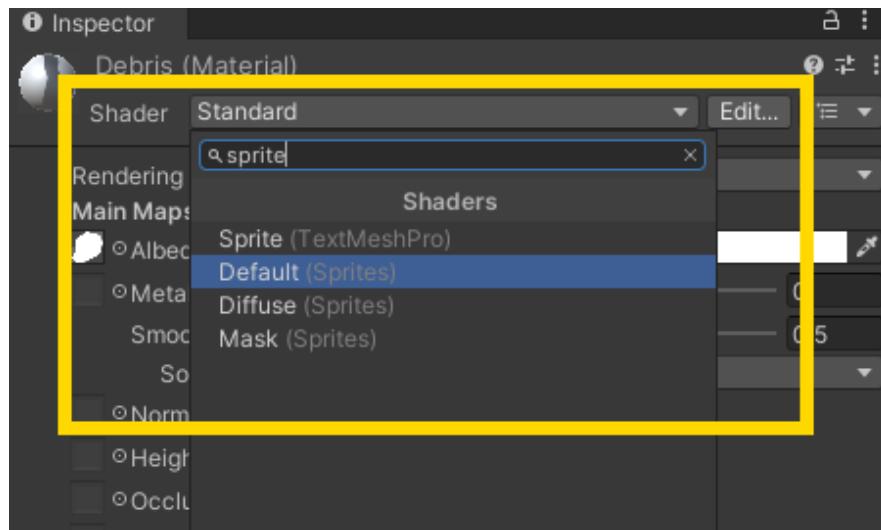
The **Particle System Renderer** needs a **Material**, so we need to create one for the **Debris** image. Go to the **Materials** folder and right-click inside the folder. Select **Create**, then **Material**, and name the Material **Debris**.



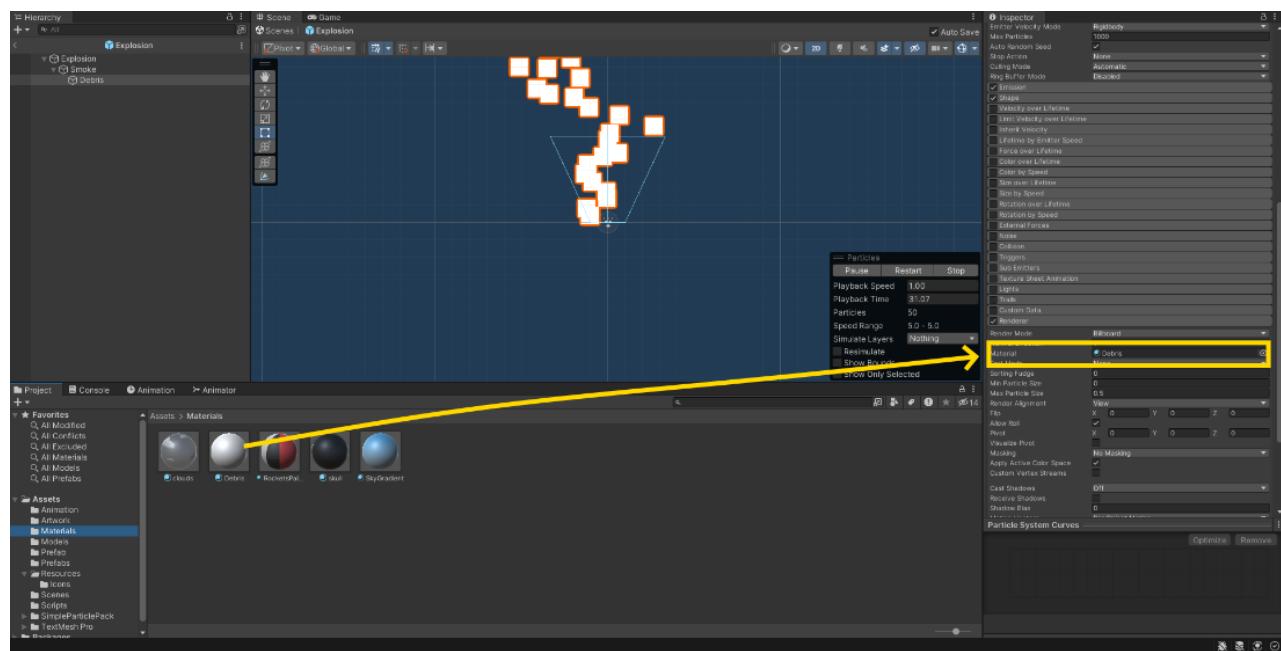
38 To add the image to the **Material**, simply drag it from the **Artwork** folder into the box for **Albedo** in the **Material Shader**.



39 The debris particles need to be rendered as sprites, so you'll need to change the shader. In the **Inspector**, click on **Shader** and search **Sprites** then **Default** from the menu.



40 Go back to the **Explosion** in the Prefabs folder. Select the **Debris Particle System** and drag the **Debris** material from the **Materials** folder into the **Material** slot in the **Particle System Renderer** component.

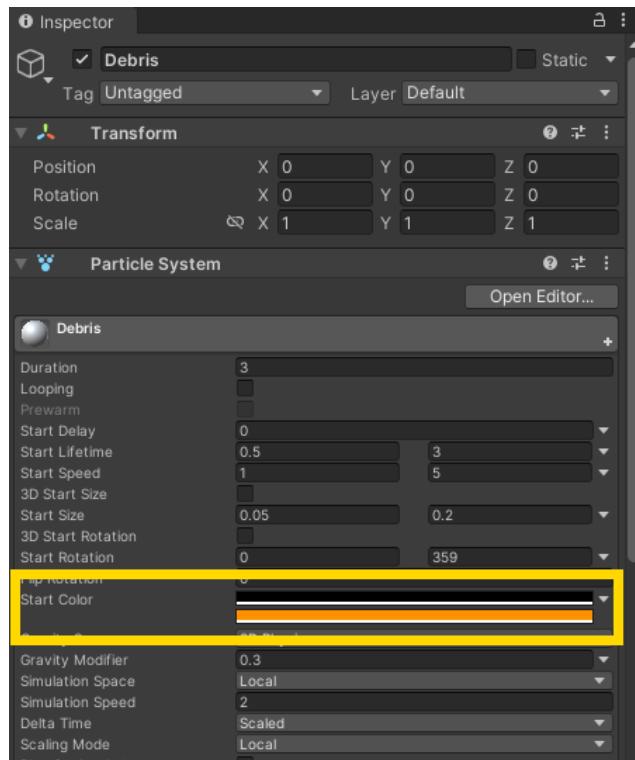


41

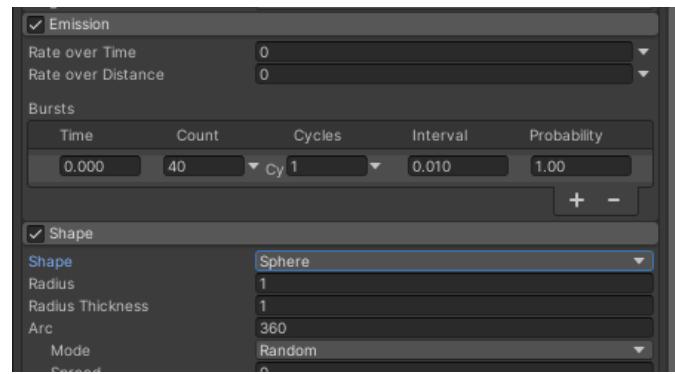
By now, you should be well acquainted with adjusting the settings for a **Particle System**. We want the debris to be little bits that fly out from the explosion and fall to the ground, so this time we are applying gravity. The following settings worked best for us.

- Duration: 3.0
- Looping: off
- Start Lifetime: 0.5 - 3
- Start Speed: 1 - 5
- Start Size: 0.05 - 0.2
- Start Rotation: 0 - 359
- Start Color: see below
- Gravity Modifier: 0.3
- Simulation Speed: 2

Click on the arrow to the right of the color window to select Random Between Two Colors and choose black and orange.

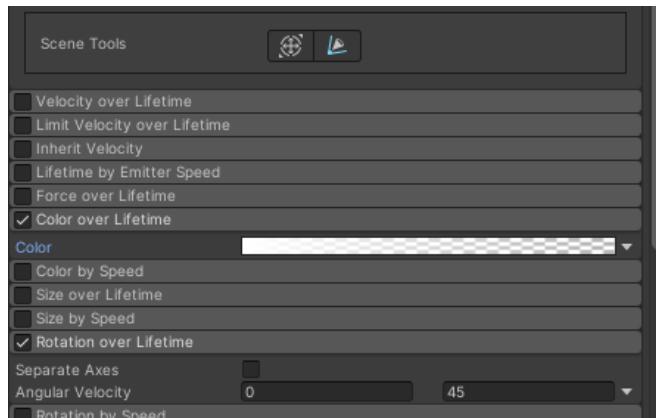


In Emission, set Rate Over Time to 0 and add a Burst with the count of 40. In Shape, change the Shape to Sphere.

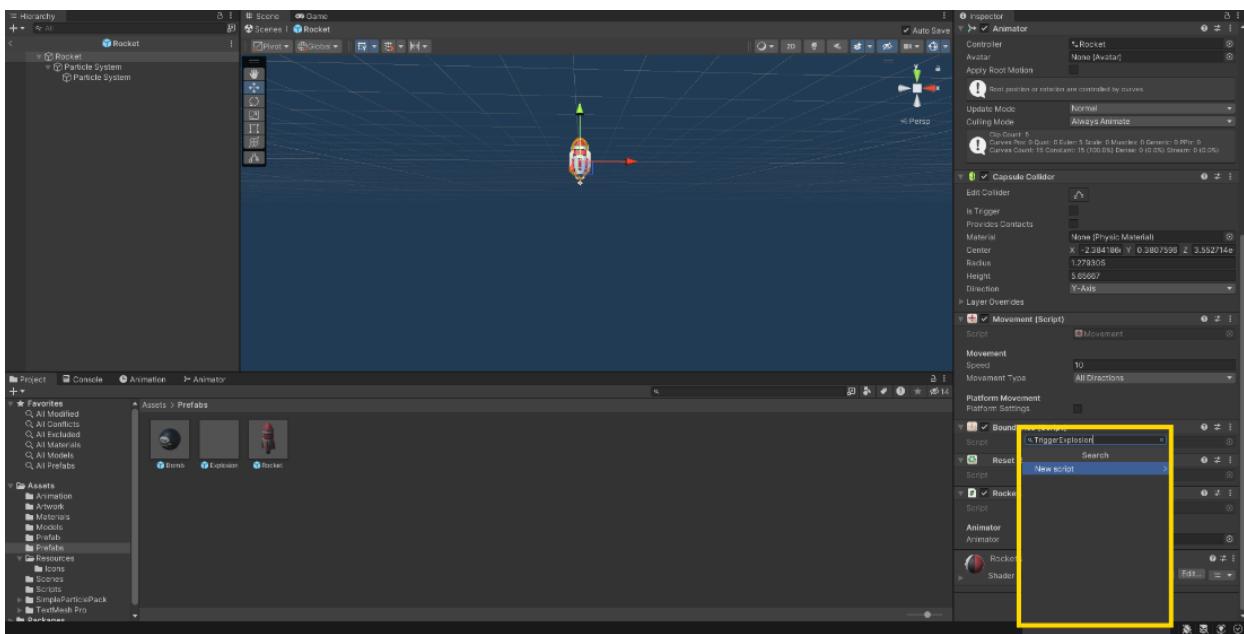


Enable Color Over Lifetime and use the fade to 0 Alpha preset that we saved earlier.

Enable Rotation Over Lifetime and set a Random range between 0 and 45.



42 Go back to the **Hierarchy** and make sure that the original explosion has been deleted from the scene. We still need to trigger the explosion when the **Rocket** hits a **Bomb**. In the **Prefabs** folder, select the **Rocket** and at the bottom of the **Inspector**, click **Add Component** and select **New Script**. Name the script **TriggerExplosion**.



If you've lost track of any of the particle systems that you been working on, double-check their position in the transform component.

43 Open the new script in the script editor. The script needs a variable to identify the explosion prefab, so add a **public** variable of the type **GameObject** and name it **explosion**.

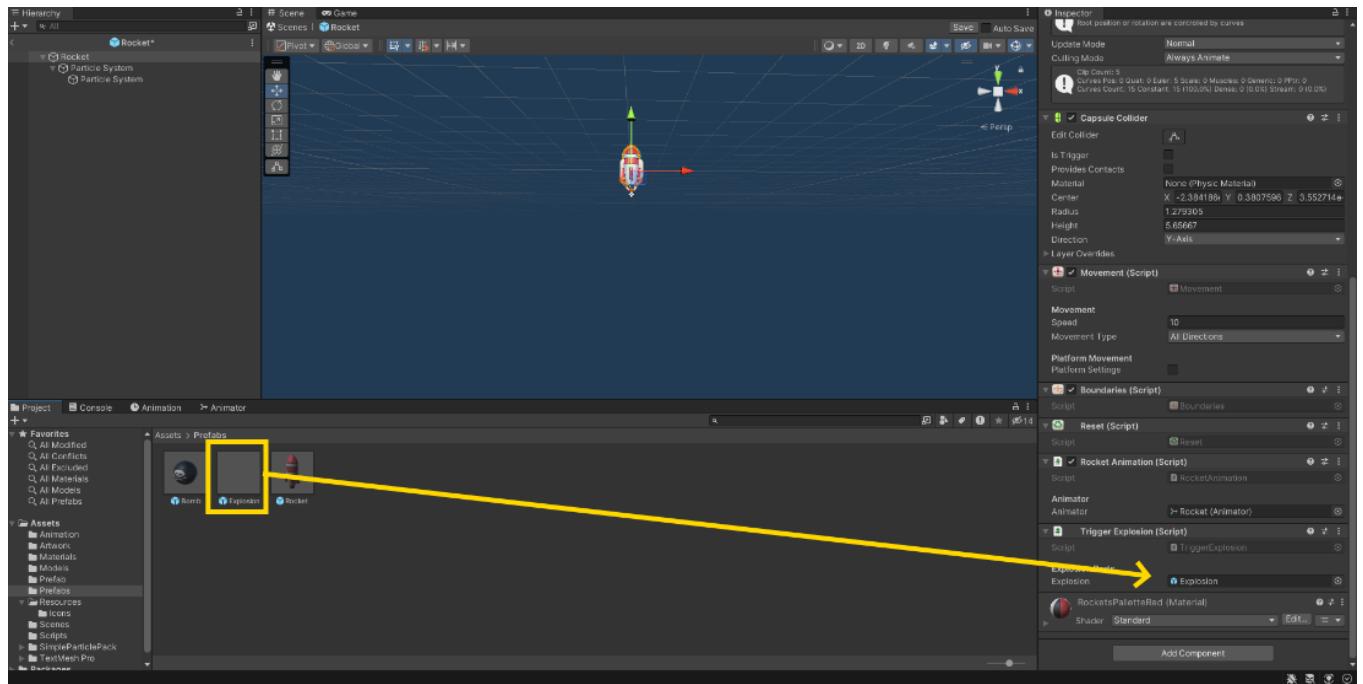
You want the script to activate when anything touches the object this script is attached to (the Rocket) so let's add a function that is a private void **OnCollisionEnter** (the parameter is type of Collision and is named **collision**).

Inside the function Instantiate the explosion prefab at the position and rotation of the **GameObject** this script is attached to (the Rocket).

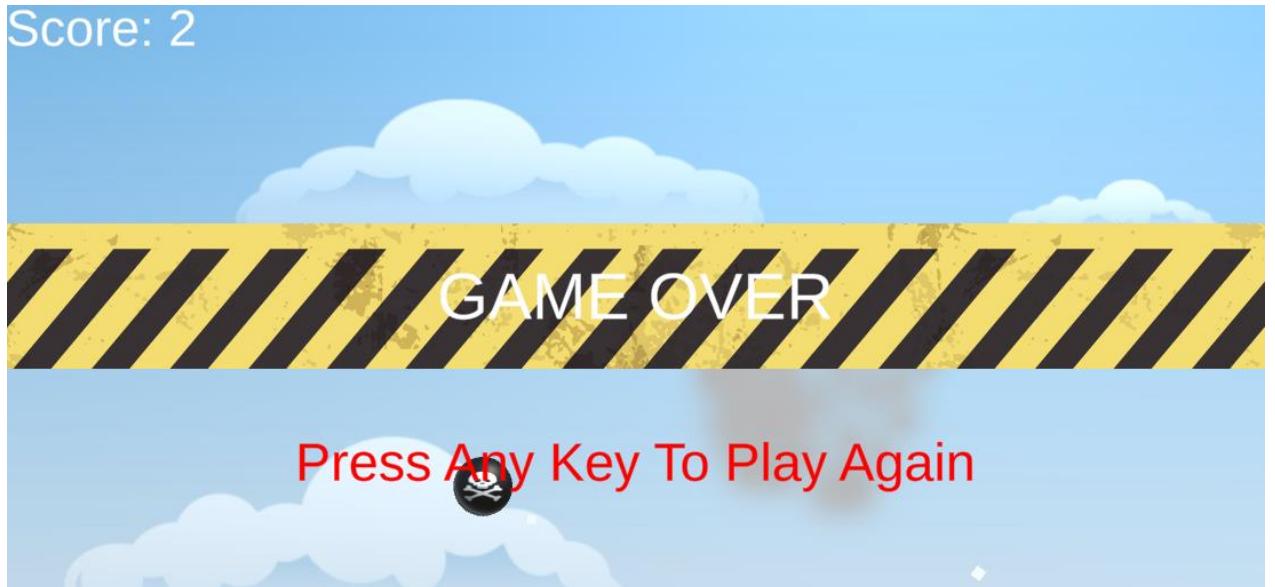
Save the script.

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class TriggerExplosion : MonoBehaviour
6  {
7
8      [Header("Explosion Parts")]
9      public GameObject explosion;
10
11     [UnityMessage]
12     private void OnCollisionEnter(Collision collision)
13     {
14         Instantiate(explosion, transform.position, transform.rotation);
15     }
16 }
```

44 Switch back to Unity and find the **Trigger Explosion** script attached to the **Rocket** prefab. Drag the **Explosion** prefab into the slot for the **Explosion** variable.



45 Now when you play the game, hitting anything with the rocket causes an explosion. The only problem is that the splash screen appears at the same time!



46 To fix this, you need to add a delay before the **GameManager** shows the splash screen. Open the **GameManager** script. To know when the explosion smoke has cleared, add a **private** variable of the type **bool** named **smokedCleared** and set it to **true**.

```
6  Unity Script (1 asset reference) | 0 references
7  public class GameManager : MonoBehaviour
8  {
9      private Spawner spawner;
10     public GameObject title;
11     private Vector2 screenBounds;
12
13     public GameObject splash;
14
15     [Header("Player")]
16     public GameObject playerPrefab;
17     private GameObject player;
18     private bool gameStarted = false;
19
20     [Header("Score")]
21     public TMP_Text scoreText;
22     public int pointsWorth = 1;
23     private int score;
24     private bool smokeCleared = true;
25
26 }
```

47 Go down to the **OnPlayerKilled** function in the script. This function gets called as soon as the rocket is destroyed and that's where the splash is set to active. Replace **splash.SetActive(true)** with **Invoke("SplashScreen", 2f)**.

Invoke will call the named function after an indicated period of time. In this case, the time is 2 seconds.

While here, we are going to add a new function called **SplashScreen** that sets the splash screen to appear.

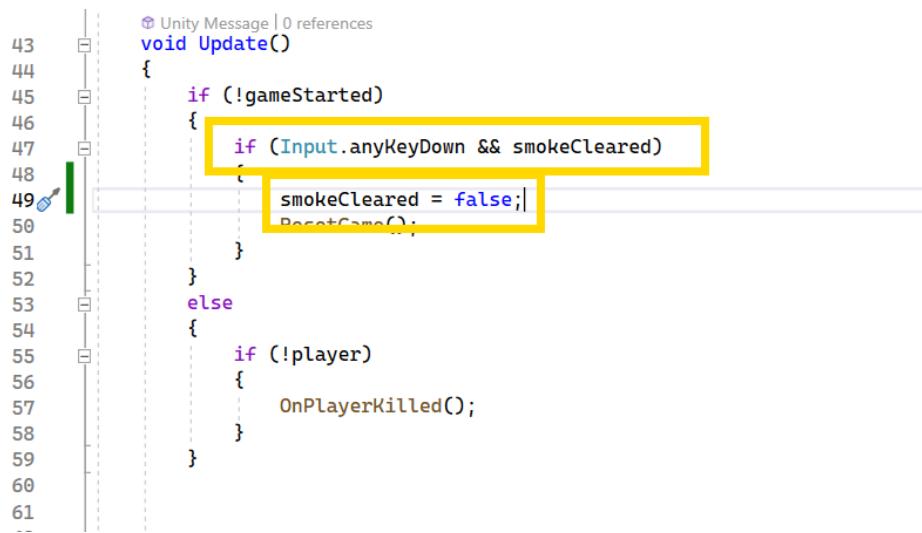
```
79
80     void OnPlayerKilled()
81     {
82         spawner.active = false;
83         gameStarted = false;
84
85         splash.SetActive(true);
86
87         Invoke("SplashScreen", 2);
88     }
89
90     void SplashScreen()
91     {
92         smokeCleared = true;
93         splash.SetActive(true);
94     }
95 }
```

48

Go to the **Update** function in the script. Right now, it is possible to restart the game by pressing any key before the splash screen appears. To fix that, modify the conditional so that nothing happens unless both **Input.anyKeyDown** and **smokeCleared** is true (in C# 'and' is represented by "**&&**").

Inside the conditional, let's add a line to set **smokeCleared** to **false** so that a new game can't be started again until the **SplashScreen** function has been invoked.

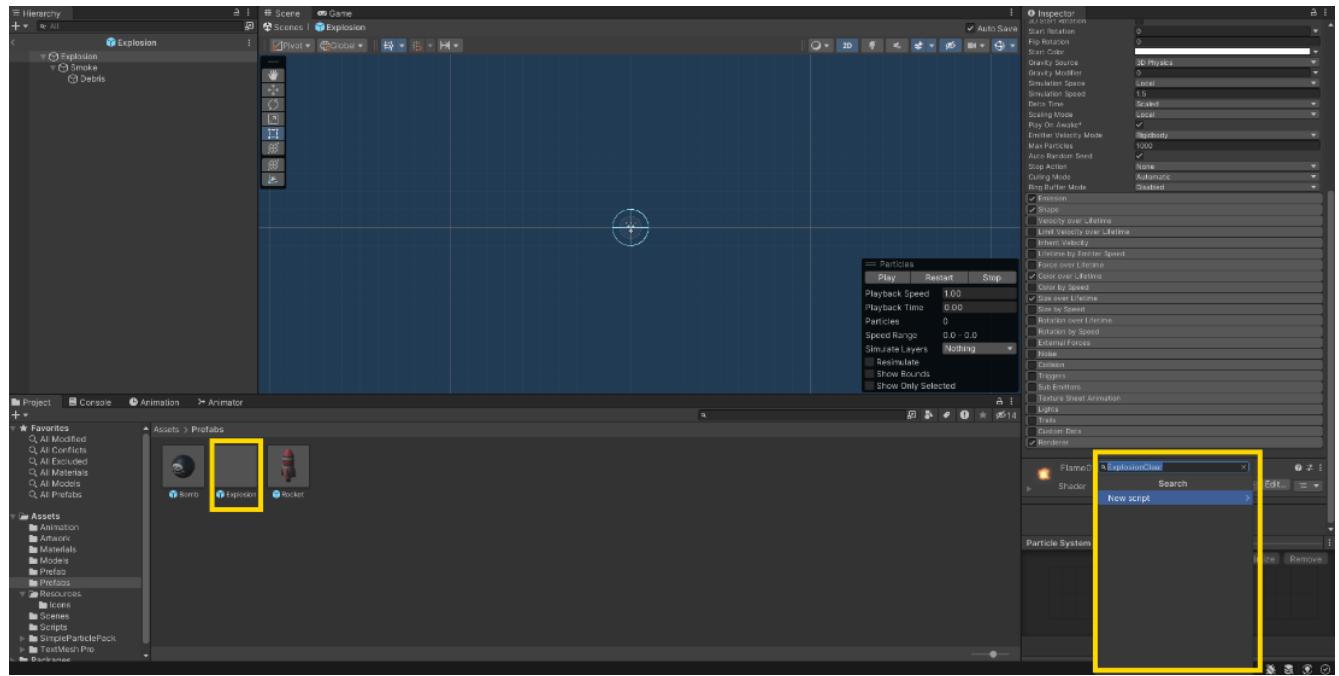
Don't forget to save your script! Return to Unity.



```
43     void Update()
44     {
45         if (!gameStarted)
46         {
47             if (Input.anyKeyDown && smokeCleared)
48             {
49                 smokeCleared = false;
50                 ResetGame();
51             }
52         }
53     }
54     else
55     {
56         if (!player)
57         {
58             OnPlayerKilled();
59         }
60     }
61 }
```

49 One thing left to do. You are using **Instantiate** to make copies of the **Explosion** prefab, but there's nothing to clean them up. In the Prefabs menu, open the **Explosion** and create a new C# script called **ExplosionClear**.

Open it in the script editor.



50

Since the explosion only plays once, it's safe to remove it after the particle systems have stopped playing. In the explosion, the smoke takes the longest to clear, so let's track that.

Add a **private** variable of the type **ParticleSystem** and name it **particleSmoke**.

The moment the object is instantiated, (**public void Awake**) you'll need to define **particleSmoke**. Since it is a child of the explosion object, you can set **particleSmoke** to **gameObject.GetComponentInChildren()**

Finally in the **Update** function, we check to see if **particleSmoke** is still running (**IsAlive**). If it isn't, **Destroy** this **GameObject**.

Save your script and return to Unity.

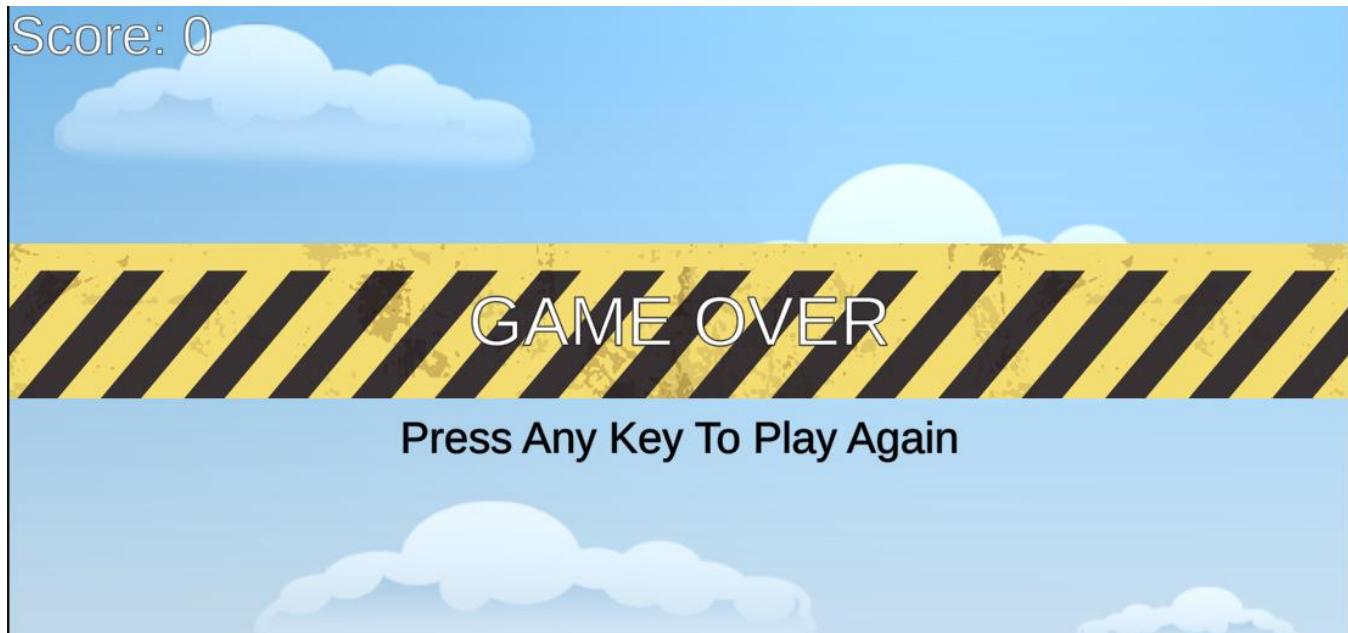
```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class ExplosionClear : MonoBehaviour
6  {
7      private ParticleSystem particleSmoke;
8
9      private void Awake()
10     {
11         particleSmoke = GetComponentInChildren<ParticleSystem>();
12     }
13
14     private void Update()
15     {
16         if (!particleSmoke.IsAlive())
17         {
18             Destroy(gameObject);
19         }
20     }
21 }
22
```

51

Now the game runs and looks pretty sharp! Too bad you can't remember your best score. If only there was something to fix that...

52

Before we move onto the final part, let's change how some things look. Using what you know about UI, change your UI elements to look how you want. You can copy the image below or use what you think looks best. You can also try messing around with the particle systems, try changing the color of the smoke to an alien green or maybe try changing it to a golden yellow.



Activity 11: Dropping Bombs Part 5

What's the point of having a score if you have nothing to compare it to? In this activity, you will complete the Dropping Bombs game by adding a Best Score feature.

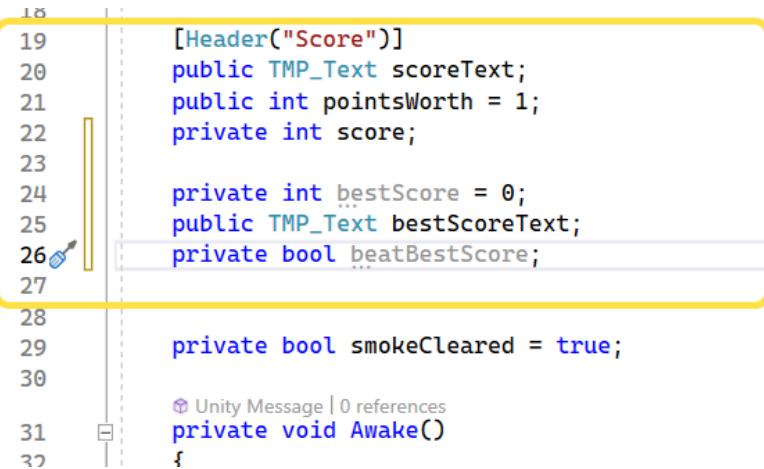
- 1 Before making additional changes to your project, let's make a backup of the entire game. In the **Projects** panel, make sure that the **Assets** folder is selected. Then click on the **Assets** tab at the top of the screen and select **Export Package**. Make sure that all the assets are selected before clicking on the **Export** button. Give the package a name like **JS-DroppingBombsPart4**.
- 2 The first thing we need to do is have something in the interface to display the best score. Select the **Canvas** in the **Hierarchy** panel and find the **Score** Text object. Select it and make a copy of it by clicking **Ctrl+D**. Name the new object **BestScore** and position it so that it is below the Score object. Change the text to **Best Score: 0**.



- 3** Open the **GameManager** script in the script manager. We need to set up some variables to track the best score. First, create a **private** integer for the score itself named **bestScore**. Set it to **0**.

You'll need a **public** variable of the **TMP_Text** type for the new text object that was just created. Let's call it **bestScoreText**.

Finally, in order to know if the high score has been beaten, create a **private bool** called **beatBestScore**.



```
18
19 [Header("Score")]
20 public TMP_Text scoreText;
21 public int pointsWorth = 1;
22 private int score;
23
24 private int bestScore = 0;
25 public TMP_Text bestScoreText;
26 private bool beatBestScore;
27
28
29 private bool smokeCleared = true;
30
31 Unity Message | 0 references
32 private void Awake()
{
```

- 4** When the game is started, you want to hide both the **Score** and **BestScore** objects. Add a line to the **Awake** function to set **bestScoreText.enabled** to **false**.

We are storing the value of **bestScore** as an integer in the **PlayerPrefs**. **PlayerPrefs** is data that gets saved on the local machine with the game and is not lost when the game is stopped.

Set **bestScore** to **PlayerPrefs.GetInt("BestScore")** and set the **bestScoreText.text** to "**Best Score: " + bestScore.ToString()** (note the extra space after the colon and before closing the quotes. **ToString** converts the integer into text).

The screenshot shows a portion of a Unity script with two highlighted sections. The first section, around line 37, contains the line `bestScoreText.enabled = false;`. The second section, around line 48, contains the assignment of `bestScore` from `PlayerPrefs.GetInt("BestScore")` and the assignment of `bestScoreText.text` to the string `"Best Score: " + bestScore.ToString();`. Both sections are enclosed in yellow boxes.

```
31     @ Unity Message | 0 references
32     private void Awake()
33     {
34         spawner = GameObject.Find("Spawner").GetComponent<Spawner>();
35         screenBounds = Camera.main.ScreenToWorldPoint(new Vector3(Screen.width, Screen.height, Camera.main.transform.position.z));
36         scoreText.enabled = false;
37         bestScoreText.enabled = false;
38     }
39
40     // Start is called before the first frame update
41     @ Unity Message | 0 references
42     void Start()
43     {
44         spawner.active = false;
45         title.SetActive(true);
46         splash.SetActive(false);
47         bestScore = PlayerPrefs.GetInt("BestScore");
48         bestScoreText.text = "Best Score: " + bestScore.ToString();
49     }
50 
```

- 5** In the **ResetGame** function, you'll do two things. To start a new game, reset the **beatBestScore** boolean to **false**.

Also set **bestScoreText.enabled** to **true** so that you can see it.

The screenshot shows the **ResetGame** function with a highlighted section around line 116. This section contains the assignments `beatBestScore = false;` and `bestScoreText.enabled = true;`. The entire function block is enclosed in a yellow box.

```
106     1 reference
107     void ResetGame()
108     {
109         spawner.active = true;
110         title.SetActive(false);
111
112         splash.SetActive(false);
113
114         scoreText.enabled = true;
115         score = 0;
116         beatBestScore = false;
117         bestScoreText.enabled = true;
118
119         player = Instantiate(playerPrefab,new Vector3(0,0,0), playerPrefab.transform.rotation);
120         gameStarted = true;
121     }
122 
```

- 6** When the **Rocket** is destroyed, **OnPlayerKilled** is called. This is a good place to see if the player has beat the best score. First, you'll want to get the value of the current score from the **scoreSystem** object. This is assigned to the variable called **score**.

Next, compare **score** to the value of **bestScore** as stored in **PlayerPrefs**. If **score** is greater, then change **bestScore** to equal **score** and save the new value of **bestScore** to **PlayerPrefs**.

Then we set **beatBestScore** to **true** and update the text of the **bestScoreText** object.

Save your script.

```
90      1 reference
91      void OnPlayerKilled()
92      {
93          spawner.active = false;
94          gameStarted = false;
95
96          splash.SetActive(true);
97
98          Invoke("SplashScreen", 2);
99
100         if(score > bestScore)
101         {
102             bestScore = score;
103             PlayerPrefs.SetInt("BestScore", bestScore);
104             beatBestScore = true;
105             bestScoreText.text = "Best Score: " + bestScore.ToString();
106         }
107
108
109     0 references
```

- 7** Switch back to Unity. Select the **GameManager** object and drag the **BestScore** text object into the slot for **Best Score Text** in the script component.



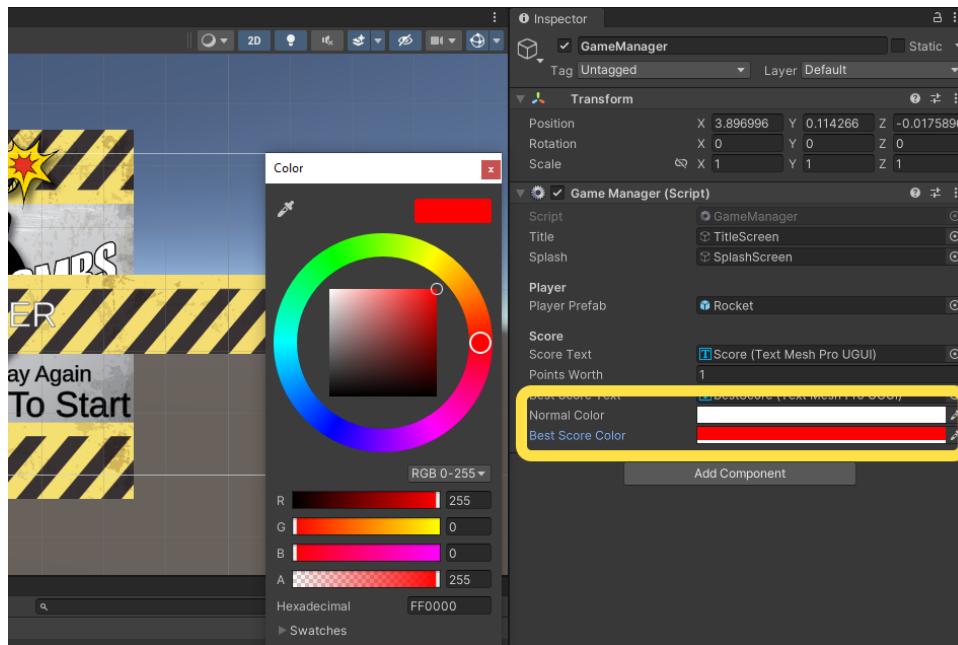
- 8** At this point you can test the game and see the Best Score working, but we are going to add one last detail. When we beat the high score, we are going to change the text color.

To start, let's go back to our **GameManager** script and create two public Color variables.

```
23  
24  
25  
26  
27  
28     public Color normalColor;  
29     public Color bestScoreColor;  
30  
31  
32  
33     private void Awake()  
34     {  
35         spawner = GameObject.Find("Spawner").GetComponent<Spawner>();  
36         screenBounds = Camera.main.ScreenToWorldPoint(new Vector3(Screen.width, Screen.height, Camera.nearPlaneZ));
```

9 Back in Unity, we can see the two **Color** variables that we can click on to change the color. For our **normalColor**, we probably want white or black, but for the **bestScoreColor**, you can pick whatever color you want. I'll use red.

If you can't see the color, try changing the **Alpha (A)** value of the **Color** which defaults to **0**.

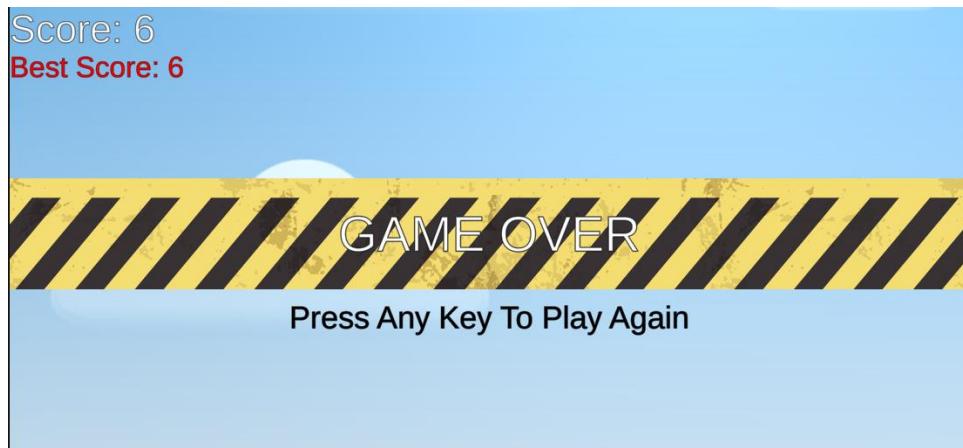


- 10** Finally, we have two lines to add. Back in the **OnPlayerKilled** function, we need to set the **bestScoreText** color to the **bestScoreColor**. And the same in the **ResetGame** function, but we are resetting it back to normal.

```
    90
91
92     1 reference
93     void OnPlayerKilled()
94     {
95         spawner.active = false;
96         gameStarted = false;
97
98         splash.SetActive(true);
99
100        Invoke("SplashScreen", 2);
101
102        if(score > bestScore)
103        {
104            bestScoreText.color = bestScoreColor;
105
106            bestScore = score;
107            PlayerPrefs.SetInt("BestScore", bestScore);
108            beatBestScore = true;
109            bestScoreText.text = "Best Score: " + bestScore.ToString();
110
111    }
112
113     1 reference
114     void ResetGame()
115     {
116         bestScoreText.color = normalColor;
117
118         spawner.active = true;
119         title.SetActive(false);
120
121         splash.SetActive(false);
122
123         scoreText.enabled = true;
124         score = 0;
125
126         scoreText.text = "Score: " + score.ToString();
127
128    }
```

- 11** Save your project and play your game. Notice how the best score from the last time you played is still there? And now when you beat the best score, the color of the text changes to red.

At this point, Dropping Bombs is complete! If there is anything else you wish to change about your UI, colors, or Spawner, now is the time to do it!



GLOSSARY

2D Object: A 2D GameObject such as a tilemap or sprite.

3D Object: A 3D GameObject such as a cube, terrain or ragdoll.

Animation: Animation is the process of creating a sequence of static or dynamic images that give the illusion of movement. These images can either be two-dimensional (2D), or three-dimensional (3D). Animation has been around for centuries as an art form, and while it used to be created with hand-drawn images, modern technology allows animators to create using computer software.

Animation Blend Shape: Having a mesh smoothly change its form to another mesh.

Animation Blend Tree: Used for continuous blending between similar Animation Clips based on float Animation Parameters.

Animation Clip: This represents a single animation asset in Unity. It contains keyframe data for animating specific GameObjects or characters.

Animation Clip Node:

Animation Key: An animation key, also known as a keyframe, is a specific point in time inside an animation where a parameter (such as position, rotation, or scale) is set to a particular value.

Animation Layer: The process of animating visual elements or components arranged in layers within a user interface (UI) or a 2D/3D scene.

Array: A type of collection, which is a set of data that's stored and accessed using a single variable instead of multiple variables.

Aspect ratio: An aspect ratio is a proportional relationship between an image's width and height.

Asset: An asset is a representation of any item that can be used in your game or project. An asset may come from a file created outside of Unity, such as a 3D model, an audio file, an image, or any of the other types of files that Unity supports.

Asset Package: Asset packages are collections of files and data from Unity projects, or elements of projects, which Unity compresses and stores in one file with the .unitypackage extension.

Asset Store: The Unity Asset Store contains a library of free and commercial assets that Unity Technologies and members of the community create.

Boolean: In computing, the term Boolean means a result that can only have one of two possible values: true or false.

Box Collider: The Box collider is a built-in cube-shaped collider. It is useful for in-application items such as boxes and crates, or as a simple collider shape that you can stretch and flatten to make ledges, steps, and any other cuboid objects.

Build: A build in software development, refers to compiling and packaging the source code and assets into an executable or deployable package for testing or distribution.

Camera: A camera is a device through which the player views the world.

Canvas: The Canvas is the area that all UI elements should be inside. The Canvas is a Game Object with a Canvas component on it, and all UI elements must be children of such a Canvas.

Capsule Collider: The Capsule collider is a built-in 3D capsule-shaped collider made of two half-spheres joined together by a cylinder.

Character Controller: A CharacterController allows you to easily do movement constrained by collisions without having to deal with a rigidbody.

Cinemachine: Cinemachine is a modular suite of camera tools for Unity which give AAA game quality controls for every camera in your project.

Collider: A collider is a Unity component that defines the shape of a GameObject for the purposes of physical collisions.

Component: Components are the functional pieces of every GameObject. Components contain properties which you can edit to define the behavior of a GameObject. For more information on the relationship between components and GameObjects, see GameObjects. To view a list of the components attached to a GameObject in the Inspector window, select a GameObject in either the Hierarchy window or the Scene view.

Console Window: The Console Window shows errors, warnings and other messages generated by Unity. You can also show your own messages in the Console using the Debug class.

Content: The content is the term used to describe the objects in your scene that are being rendered. Their appearance is a result of the lighting context acting on the materials that have been applied to the objects.

Culling Mask: In Unity, the "Culling Mask" refers to a property of cameras used to control which layers of objects are rendered in a scene. This property determines which layers are visible to the camera and which are not.

Event System: The Event System is a way of sending events to objects in the application based on input, be it keyboard, mouse, touch, or custom input.

Extrude Edges: The Extrude Edges tool pushes a new edge out from each selected edge, connected by a new face for each edge

FirstPersonShooter: A type of video game whose gameplay involves shooting enemies and other targets and in which a player views the action as though through the eyes of the character they are controlling.

Float: Float is a shortened term for "floating point." By definition, it's a fundamental data type built into the compiler that's used to define numeric values with floating decimal points.

Frame: In Unity, a frame is considered a rendered image presented to the player's screen.

Frames Per Second: Frames per second (fps) is a measure of how many still images, or frames, are displayed in a single second of video or animation.

Function: A function is a self-contained program segment that carries out some specific, well-defined task.

GameObject: GameObjects are the fundamental objects in Unity that represent characters, props and scenery.

Gizmo: Gizmos are graphics associated with GameObjects in the Scene.

HDR: HDR stands for High Dynamic Range and refers to a technique that expresses details in content in both very bright and very dark scenes.

IDE: An integrated development environment (IDE) is a software application that helps programmers develop software code efficiently. The most commonly used IDE with Unity is Microsoft Visual Studio, which is often recommended and provided as a default option when installing Unity on Windows.

Input Key: An "Input Key" is an identifier used in software development, particularly in game development with platforms like Unity, to detect and respond to specific user inputs, such as keyboard presses, mouse clicks, or touchscreen gestures.

Input Manager: The Input Manager is where you define all the different input axes and game actions for your project.

Inspector: The Inspector displays detailed information about your currently selected GameObject, including all attached Components and their properties. Here, you modify the functionality of GameObjects in your scene.

Instantiate: Instantiating means bringing the object into existence.

Integer: A number without any decimal value, like 3 or 200.

Interactable: An object in a Scene that the user can interact with (for example, grab it, press it, or throw it).

Interpolation: Interpolation provides a way to manage the appearance of jitter in the movement of your Rigidbody GameObjects at run time.

Invoke: The Invoke functions allow you to schedule method calls to occur at a later time.

Invoke Repeating: The InvokeRepeating method in Unity is a built-in function that allows you to schedule a method to be called repeatedly at fixed time intervals.

Keyframe: Keyframe animation records the state of the object and then interpolates between the changes of each keyframe.

Layer: Layers are most commonly used by Cameras to render only a part of the scene, and by Lights to illuminate only parts of the scene. But they can also be used by raycasting to selectively ignore colliders or to create collisions.

Layer Mask: In Unity, a "layer mask" is a bitmask that allows you to selectively include or exclude specific layers from various operations, such as physics calculations, raycasting, and rendering.

Material: All rendered meshes use a material to define how that object is displayed. A material's attributes are based on what Shader is used by the material.

Mesh: All 3D objects in Unity are broken down into triangles or polygons which comprise of the mesh.

Mesh Collider: A collider component that takes the shape of the current mesh attached to it.

Mesh Renderer: A component used to control how a mesh appears in the scene

Model: Any 3D asset. Often created in a 3D modeling program.

MonoBehavior: Unity's base class that many Unity Scripts derive from. Offers many functions such as void Update, Start and Awake

Near clipping plane: The clipping plane that is closest to the camera.

Normal: A vector representing the direction perpendicular to a mesh. Unity uses Normals to apply shading and calculate object orientation.

Orthographic: A camera that doesn't measure depth. All objects regardless of how far away they are will look the same distance away from each other.

Package: A single file containing a collection of assets or features for importing to and exporting from Unity Projects.

Parent: An object that contains one or more child objects in the Hierarchy. Child objects move, scale and rotate as their parent does.

Particle: An individual object that is created and managed by a particle system. A particle can be a mesh, texture or even a collection of sprites.

Particle System: A means of creating various effects such as smoke, fluid and gas by using small, simple particles.

Perspective: A type of camera that measures Depth, similar to real-life cameras.

Physics: Physics in Games represents real-world attributes such as Gravity, Collisions, Time, and more. Physics can be changed through the settings and through scripting.

Physics Engine: Uses mathematics to simulate physics (such as gravity and collision) in a virtual environment

Physics Material: Used to simulate physics for 3D objects. A Physics Material 2D can be used for 2D objects.

Pixel: The smallest part of a computer image. Pixel size depends on screen resolution.

Player Settings: A window that contain settings for how Unity will build and display the final application

Plug-in: Code that can be added to create additional functionality in Unity.

Position: A value that represents where an object is positioned in space based on an X, Y and Z axis.

Prefab: A type of asset that is a stored version of a gameobject that includes all attached components. A prefab can be instantiated during runtime.

Project: A collection of scenes, assets and settings that make up the entirety of what you create in Unity.

Project Settings: Settings that you can adjust to determine how physics, input and other parts of the project behave.

Project View: A panel in the Unity Editor that gives you access to all the assets in your project.

Quad: A 3D object that is similar to a plane but has a much more efficient mesh.

Quaternion: represents an object's rotation in 3D space

Rendering: The process through which Unity takes the objects in a scene and creates a graphic image.

Rendering Mode: A shader parameter that gives you transparency options.

Rig: A collection of joints in a mesh, much like a skeleton.

Rigidbody: A Component that is added so forces such as gravity affect the object.

Rotation: A value that represents how an object is rotated in space based on an X, Y and Z axis.

Scale: A value that represents how big an object is in the X, Y and Z axis.

Scene: A world that holds many GameObjects as needed by the developer. Scenes can be loaded and unloaded and are primarily used for multiple levels.

Scene View: The window in the Unity editor through which the current scene can be viewed and modified.

Scripts: Written code that directs the behavior of objects in a Unity Project. Scripts in Unity are written in C# (C Sharp)

Shader: A component that applies a material to a GameObject. Some shaders are more advanced than others and can even manipulate how the texture moves.

Skybox: A material/texture used to represent the game's sky.

Sphere Colliders: A collider component that is the shape of a primitive sphere.

Sprite: A graphic that has no depth. Can be anything from an image used as a texture to a 2D object.

Sprite Mask: A 2D texture used to hide parts of an underlying graphic from the renderer

Sprite Renderer: A component similar to a mesh renderer but specifically for 2D sprites.

Sprite atlas: A collection of 2D textures in a single image to save processing time.

State Machine: Used in the Animator Controller to keep track of and edit the various possible animation states and their transitions. It can also be a term that refers to an AI and how they act. For example, an AI may have a walk state or an attack state that code will transition between.

String: An object which can only have a value of text.

Tag: A reference that is applied to objects so they can be easily identified by a script.

Terrain: A term for a landscape in your scene, usually natural. A terrain GameObject can be edited through that object's inspector to quickly create a detailed environment.

Terrain Collider: A collider component that takes the shape of the terrain object it is attached to.

Text: Part of the UI that presents information to the user in a non-interactive way.

Text Input Field: A UI object that allows for the user to enter requested text.

Text Mesh: the mesh that is used to display a text string

Text Mesh Pro: A Unity Package that produces high-quality text, as well as other features involving UI elements.

Texture: An image that is applied to a GameObject, sprite or mesh.

Third Person Shooter: A shooter game where the camera is positioned above and behind the player.

Toggle: A UI checkbox that can be either on or off

Toolbar: A row of buttons and tabs to allow quick editing.

Trail Renderer: Applies trails behind a GameObject as it moves in the game.

Transform: A component that handles the object's position, rotation and scale.

Trigger: A type of collider that activates code but isn't physically touchable by Gameobjects,

Variable: A symbol that represents an object. For example, you may have a variable that represents a number, vector, object or an array of something.

Vector: A mathematical concept that allows you to describe a direction and a magnitude. In games, a vector is used to describe properties such as position, rotation or scale.

Visual Studio: An IDE created by Microsoft that is used to type in code.

Void Awake: A function that happens on the frame when a script is enabled, but before void Start.

Void FixedUpdate: Similar to Void Update, FixedUpdate is called every fixed frame-rate frame. This function is called at a fixed rate and is not dependent on the game's speed.

Void Start: A Unity Function that happens on the frame when a script is enabled, just before any of the void Update function is called.

Void Update: A Unity Function that happens once every frame. This function is called at a variable rate depending on the game's speed.