Spaceships and Asteroids (Rigidbodies & physics!)

[Introduction](#_3ic5zvd1qn1g)

[Creating our spaceship](#_vhp5hhm0ah3t)

[Configuring the Rigidbody2D](#_k07269k05dwp)

[Controlling the Spaceship](#_u8t5vpkd6kx8)

[Defining the game area](#_h3g82me820tl)

[Including the movement boost](#_xmoadg5ypio8)

[Including the flames](#_4dagxe4uteu4)

[Asteroids & prefabs](#_gd4xza2ygd7j)

[Optional exercises](#_gdlxxhncc5lv)

# 

# Introduction

For today’s lab we are going to create a project inspired by the classic **Asteroids** game. To do this, we are going to use the power of Unity’s physics. In the previous session we created a Nyan cat and we learnt to control it through manipulating its **transform**. However, there are more realistic ways of moving elements in Unity, and physics is the answer to this. The main component that allows us to do this is the **Rigidbody**.

# 

Rigidbodies in Unity allow simulating physical behaviours that we can see in the real world, such as gravity, velocity, acceleration, collisions and forces. These interactions are calculated based on different mathematical formulas like . Luckily for us, we don’t have to worry about them! Unity handles all the maths while we have fun. During this lab session, we will learn how to use and manipulate them, as well as how to use forces and colliders.

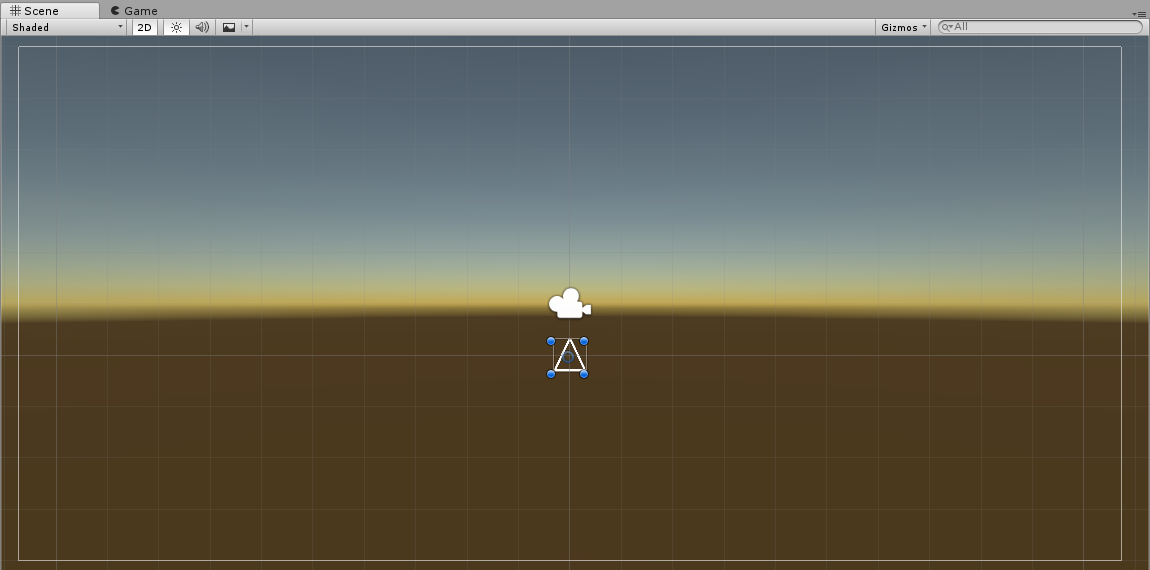
# 

# Creating our spaceship

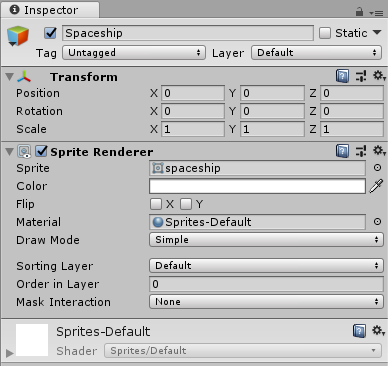
For this project, we will start creating the spaceship and include physics to it.

**Step 1.** Create a new 2D project in Unity (You should know how to do this by this session!)

**Step 2.** Create a **GameObject**. You can create an empty GameObject in the scene by right-clicking in the hierarchy and selecting **Create Empty**. However, in our case, the easiest way is to drag the spaceship sprite to our scene window.

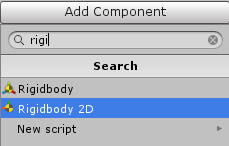


That will create a game object with a transform and a sprite renderer.



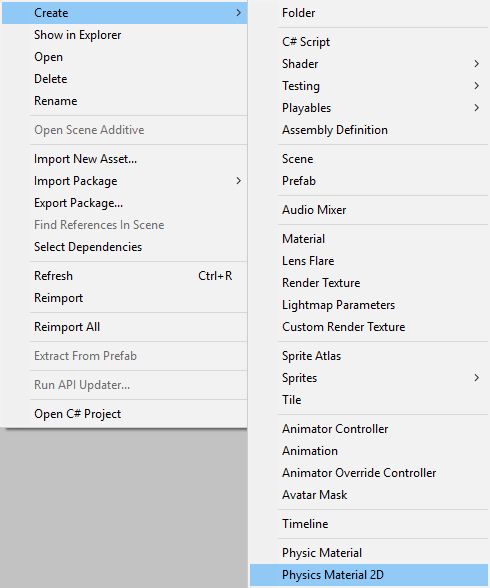
# Configuring the Rigidbody2D

**Step 3.** To start playing with physics we need to add a **Rigidbody2D** component to our spaceship. To do this, go to the inspector having the spaceship GameObject selected and click on **Add Component**. Type **Rigidbody2D** and select it to include it to our spaceship.



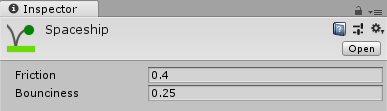
|  |
| --- |
| **📘 Rigidbody2D**  The **Rigidbody2D** component puts a GameObject under the control of the physics engine. By itself, this means that the sprite will be affected by gravity and can be controlled from scripts using forces. By adding the appropriate collider component, the sprite will also respond to collisions with other sprites. This behaviour comes entirely from Unity's physics system; very little code is required to get impressive and authentic physical behaviour and allows for "emergent" gameplay that was not explicitly coded into the game.  <https://docs.unity3d.com/ScriptReference/Rigidbody2D.html> |

**Step 4.** The Rigidbody2D component allows us to play with different physical properties of the object, such as its mass, if it’s affected by gravity or it’s fiction. To start setting up our spaceship properties we have to create a physics material first. Right click on the project window and select **Create > PhysicsMaterial2D.**

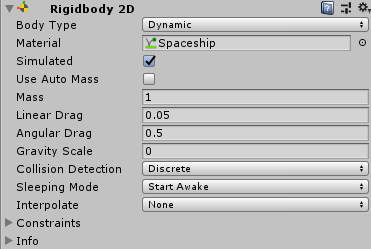


|  |
| --- |
| **📘 Physics Material 2D**  A **Physics Material 2D** is used to adjust the friction and bounce that occurs between 2D physics objects when they collide.  <https://docs.unity3d.com/Manual/class-PhysicsMaterial2D.html> |

Define the **Physics Material 2D** properties in the inspector as the ones in the following image:

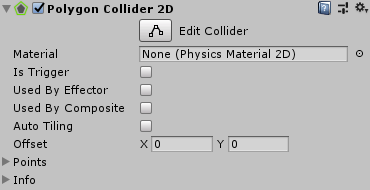
****

**Step 5.** With the spaceship Physics Material 2D defined, check the spaceship Rigidbody2D in the inspector and attach the Physics Material to it. Define the rest of the properties of the rigidbody as the ones in the following image:



Setting the **Gravity Scale** to 0 allows the spaceship to float around the scene freely. If you want to know more about Rigidbodies and its properties, feel free to play with them and change the numbers to see what happens!

**Step 6.** Currently our spaceship is affected by physics and it’s almost ready to go. However, to be able to interact with other physics objects, the spaceship needs a collision box. Collision boxes allow Unity to know which space is being occupied by the object, and therefore, to detect if two or more objects are colliding. To include a collision shape just right-click in the inspector having selected the spaceship GameObject and click on **Add Component > Polygon Collider 2D**. The collider component should look like this:



# Controlling the Spaceship

**Step 7.** Once we have all the components ready, it’s time to do some coding. In the following script, we will learn how to move our spaceship using forces. Add a new script component to the spaceship and open it in Visual Studio/MonoDevelop/Your favourite script editor.

**Step 8.** To start with our script we will create the variables for our spaceship. These variables will be the Rigidbody (The script needs to know which of all the Rigidbodies in the scene is it going to move), the speed and the rotation speed. We will include as well the corresponding headers, tooltips and ranges for each variable so we can change them easily from the editor.

|  |
| --- |
| **public class Spaceship : MonoBehaviour**  {  [Header("Physics")]  **public Rigidbody2D Rigidbody;**  [Tooltip("Speed when moving forward, in units/second")]  [Range(0,10)]  **public float Speed = 1;**  [Tooltip("Torque applied when rotating, in Newton\*metre")]  [Range(0,10)]  **public float Torque = 1;**  **}** |

**Step 9.** Once these variables have been defined, to set up the controller for the spaceship. Write the following code inside the **FixedUpdate** function (Yes, you’ll have to write the **void FixedUpdate(){}**). The **FixedUpdate** is a function very similar to **Update**: it’s called every time the physics engine update the position and rotation of the rigibodies in your game.

|  |
| --- |
| **void FixedUpdate()**  {  /\* If the vertical axis is not zero,  \* we move the spaceship.  \*/  float vertical = Input.GetAxis("Vertical");  if (vertical != 0)  {  /\* We add force in the "up" direction,  \* which points in the direction the spaceship is  pointing at.  \*/  Rigidbody.AddForce(transform.up \* vertical \* Speed);  }  float horizontal = Input.GetAxis("Horizontal");  if (horizontal != 0)  {  Rigidbody.AddTorque(-Torque\* horizontal);  }  } |

The instructions inside this function are:

1. Getting the value for the vertical axis input.
2. If the vertical axis is not 0 (This means that the vertical axis input is being used) we add a force with the defined speed to the Rigidbody of our spaceship with a forward direction. This will make the spaceship move forward or backwards using physics!
3. Gett the value for the horizontal axis input.
4. If the horizontal axis is not 0 (This means that the horizontal axis input is being used) we add a torque with the defined rotation speed to the Rigidbody of our spaceship with a forward direction. AddTorque adds a rotation force to a rigidbody, so we use it to change the direction of our spaceship.

|  |
| --- |
| **💡 FixedUpdate**  The Rigidbody2D is not a component like all the others. The physics simulation does not run in the **Update** method, but is executed in a separate *thread*. To make sure that the Update thread and the physics thread communicate properly, all code that interacts with the Rigidbody should be in a **FixedUpdate** method. |

|  |
| --- |
| **💡 Newton**  Force is measured in **Newtons**. A Newton is the force necessary to change the acceleration of an object with mass 1 Kg by 1 metre per second per second.  The method does not require Time.deltaTime. |

|  |
| --- |
| **📘 AddTorque**  Apply a torque at the rigidbody's centre of mass.  A torque is conceptually a force being applied at the end of an imaginary lever, with the fulcrum at the centre of mass. A torque of five units could thus be equivalent to a force of five units pushing on the end of a lever one unit long, or a force of one unit on a lever five units long. Unity's units are arbitrary but the principle that *torque = force x lever length* still applies.  <https://docs.unity3d.com/ScriptReference/Rigidbody2D.AddTorque.html> |

Once you have written these lines of code, make sure that you assign the Rigidbody2D of the spaceship to the script in the inspector, and assign values to the variables. To assign the rigidbody to the script, drag the spaceship gameobject from the hierarchy to the script variable in the inspector. After doing it, it should look like this:



Press play to test the spaceship controllers and make sure that everything is working!

# 

# 

# 

# 

# 

# Defining the game area

Now that the spaceship is moving you might have realized that there is a problem: the spaceship can fly off the limits of the camera view. To avoid this, we will define in the script certain limits and, in case that the spaceship goes beyond them, it will be transported to the other side of the screen. By doing this we will avoid that the spaceship goes missing and out of the sight of our camera!

To define our game area we will have to include a new variable to our script. Include it under the other variables defined:

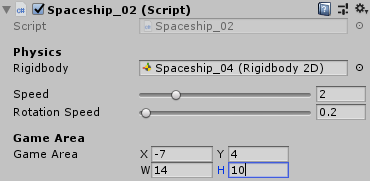
|  |
| --- |
| **public class Spaceship : MonoBehaviour**  {  [Header("Physics")]  **public Rigidbody2D Rigidbody;**  [Tooltip("Speed when moving forward, in units/second")]  [Range(0,10)]  **public float Speed = 1;**  [Tooltip("Torque applied when rotating, in Newton\*metre")]  [Range(0,10)]  **public float Torque = 1;**    [Header("Game Area")]  **public Rect GameArea;**  **}** |

Rect is a data type that allows to define the limits of a rectangle. Now, we will have to include some code in the Update method to detect that the spaceship is off the limits of our rectangle and transport it to the opposite side of the script.

|  |
| --- |
| **void Update()**  **{**  // Wraps around the edges of the game area  // From right to left  **if (transform.position.x > GameArea.xMax)**  {  /\* When the player moves too far on the right,  \* we need to bring it back to the left.  \*  \* Theoretically, we should update the x component of the position.  \* like this:  \* transform.position.x = GameArea.xMin;  \* sadly, that operation is not allowed in Unity.  \* We need to extract the position in a separate Vector3,  \* change its x component and then re-assign it back.  \*  \* Also, since we are using a rigidbody, we should change  \* rigidbody.position instead of transform.position.  \*/  **Vector3 position = Rigidbody.position;**  **position.x = GameArea.xMin;**  **Rigidbody.position = position;**  **} else**  // From left to right  **if (transform.position.x < GameArea.xMin)**  **{**  **Vector3 position = Rigidbody.position;**  **position.x = GameArea.xMax;**  **Rigidbody.position = position;**  **}**  // From top to bottom  **if (transform.position.y > GameArea.yMax)**  **{**  **Vector3 position = Rigidbody.position;**  **position.y = GameArea.yMin;**  **Rigidbody.position = position;**  **}**  **else**  // From bottom to top  **if (transform.position.y < GameArea.yMin)**  **{**  **Vector3 position = Rigidbody.position;**  **position.y = GameArea.yMax;**  **Rigidbody.position = position;**  **}**  } |

What these lines of code are doing is pretty simple: Each frame, the spaceship is checking if its position is outside the rectangle limits (using the if condition and checking if its x & y position is greater/smaller than the maximum/minimum ones of the rectangle). In case that the spaceship surpases the boundaries defined, we set its position to the opposite limit position of the rectangle.

Don’t forget to define in the inspector the rectangle. The inspector should show now something like this:



Finally, write the following function to be able to see the defined rectangle in the scene:

|  |
| --- |
| **private void OnDrawGizmos()**  **{**  // Draw a blue box (wire cube) around the game area  Gizmos.color = Color.blue;  Gizmos.DrawWireCube(GameArea.center, GameArea.size);  **}** |

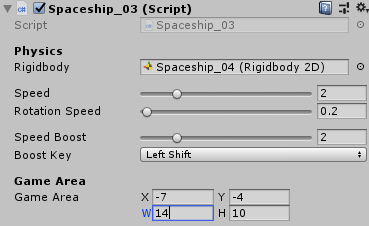
If you play the game now, the spaceship shouldn’t be able to get out of the camera’s sight!

# Including the movement boost

Our spaceship can fly around space without getting lost. However, in order to reach the speed of light and go to other galaxies, it needs a boost! In this section we will include another key to our controller that will increase the speed of the spaceship while pressed. The first step to do this is to define a variable for the boost key. Include it under the other variables:

|  |
| --- |
| [Header("Physics")]  public Rigidbody2D Rigidbody;  [Space]  [Tooltip("Speed when moving forward, in units/second")]  [Range(0,10)]  public float Speed = 1;  [Tooltip("Torque applied when rotating, in Newton\*metre")]  [Range(0,10)]  public float RotationSpeed = 1;  **[Space]**  **[Tooltip("Speed boost")]**  **[Range(0, 10)]**  **public float SpeedBoost = 1;**  **public KeyCode BoostKey = KeyCode.LeftShift;**  [Header("Game Area")]  public Rect GameArea; |

Once this is done, set a value for these new variables in the inspector.



Now, we just have to make a slight change to our controller to make this boost work.

|  |
| --- |
| **void FixedUpdate()**  **{**  /\* If the vertical axis is not zero,  \* we move the spaceship.  \*/  float vertical = Input.GetAxis("Vertical");  if (vertical != 0)  {  **bool boost = Input.GetKey(BoostKey);**  **Rigidbody.AddForce**  **(**  **transform.up \* vertical \* Speed**  **\* (boost ? SpeedBoost : 1) // Ternary operator**  **);**  }  float horizontal = Input.GetAxis("Horizontal");  if (horizontal != 0)  Rigidbody.AddTorque(-RotationSpeed \* horizontal);  **}** |

With this little addition to the code, we are first getting the value for boost (That can be either true or false depending on if the BoostKey is pressed or not). If you pay attention, we have added **(boost ? SpeedBoost : 1)** inside the Rigidbody.AddForce() instruction and multiplying the resulting value by the speed applied. This structure is called **ternary operator** and means the following:

If the value of boost is true, then use multiply the speed by SpeedBoost.

Otherwise, multiply it by one.

By doing this, when we press the boost key we will increase the speed by multiplying it by the value defined in SpeedBoost.

Try playing the game and experience true speed.

# Including the flames

Our spaceship has all the functionalities that it needs to travel through space. However, it doesn’t look as interesting as we would like. It’s time to include some effects to make it feel more real. We are going to include the flames from the different rockets of the spaceship, that are going to be visible based on if we are using them or not.

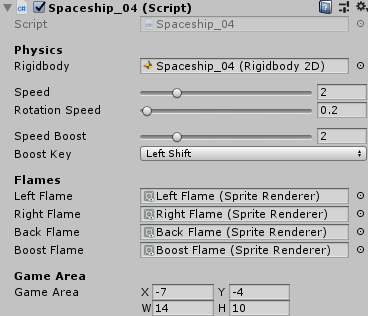
First, we need to define where the flames are, and parent them to our spaceship so they move with it. Drag the flame sprite to the spaceship in the hierarchy to parent it. Do this four times (for the left flame, right flame, back flame and boost flame). Once you finish, move and resize each of them based on where you want them to be in relation to the spaceship. After this process it should look similar to this:



However, the flames are always visible, regardless of if we are using the engines or not. We need to do some coding to fix this. Start by including 4 sprite variables to the script.

|  |
| --- |
| [Header("Physics")]  public Rigidbody2D Rigidbody;  [Space]  [Tooltip("Speed when moving forward, in units/second")]  [Range(0,10)]  public float Speed = 1;  [Tooltip("Torque applied when rotating, in Netwon\*metre")]  [Range(0,10)]  public float RotationSpeed = 1;  [Space]  [Tooltip("Speed boost")]  [Range(0, 10)]  public float SpeedBoost = 1;  public KeyCode BoostKey = KeyCode.LeftShift;  **[Header("Flames")]**  **public SpriteRenderer LeftFlame;**  **public SpriteRenderer RightFlame;**  **public SpriteRenderer BackFlame;**  **public SpriteRenderer BoostFlame;**  [Header("Game Area")]  public Rect GameArea; |

Drag the flames from the hierarchy to the script in the inspector:



Now that the script has a reference to the flames, we can alter them based on different conditions.

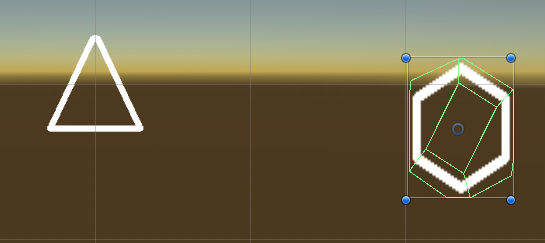
|  |
| --- |
| **void Update()**  **{**  **float vertical = Input.GetAxis("Vertical");**  **float horizontal = Input.GetAxis("Horizontal");**  **bool boost = Input.GetKey (BoostKey);**  // Flames  /\* Inside the spaceship game object there are 4 flames  \* that can be used for graphical purposes.  \*  \* Their game objects can be activated/deactivated using  \* gameObject.SetActive(true) or gameObject.SetActive(false)  \*  \* One could write:  \* if (horizontal > 0)  \* LeftFlame.gameObject.SetActive(true);  \* else  \* LeftFlame.gameObject.SetActive(false);  \*  \* which is equivalent to:  \* bool horizontalFlame = horizontal > 0;  \* LeftFlame.gameObject.SetActive(horizontalFlame):  \*  \* which is equivalent to:  \* LeftFlame .gameObject.SetActive(horizontal > 0);  \*  \*/  **LeftFlame .gameObject.SetActive(horizontal > 0);**  **RightFlame.gameObject.SetActive(horizontal < 0);**  **BackFlame .gameObject.SetActive(vertical > 0);**  **BoostFlame.gameObject.SetActive(boost );**  // Wraps around the edges of the game area  // From right to left  if (transform.position.x > GameArea.xMax)  …  **}** |

This code is checking every frame if any of the keys are pressed, and based on that we will activate or deactivate the flames, so they are coherent to the movement.

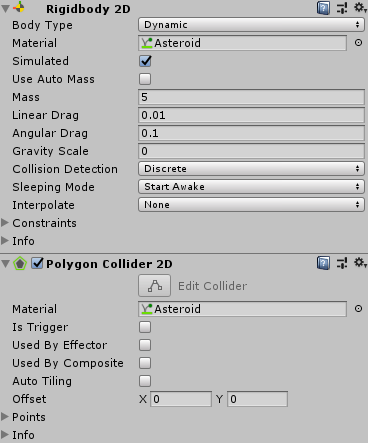
Our game is almost ready! Test that everything works as intended and go to the final step: Creating asteroids.

# Asteroids & prefabs

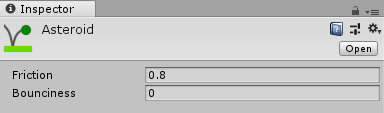
The asteroids are petty similar to our spaceship, with the difference that they don’t move unless other object applies a force to it. Drag the asteroid sprite to the scene to start with its cosmic creation.



Include a Rigidbody2D and a polygon collider just like you did with our spaceship. Modify its variables until it looks like this.



Don’t forget to create the asteroid physics material with the following values!



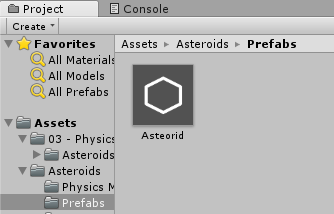
Finally, we need to create an script to make sure that the asteroids can’t get lost in space by defining the rectangle boundaries. The code for the asteroid is a simplified version of the spaceship:

|  |
| --- |
| **public class Asteroid : MonoBehaviour**  **{**  **[Header("Physics")]**  **public Rigidbody2D Rigidbody;**  **[Space]**  **[Tooltip("Max initial velocity, in units/second")]**  **[Range(0, 10)]**  **public float MaxVelocity;**  **[Header("Game Area")]**  **public Rect GameArea;**  **void Start()**  **{**  Rigidbody.velocity = Random.insideUnitCircle \* MaxVelocity;  **}**  **void Update()**  **{**  // Wraps around the edges of the game area  // From right to left  if (transform.position.x > GameArea.xMax)  {  Vector3 position = Rigidbody.position;  position.x = GameArea.xMin;  Rigidbody.position = position;  }  else  // From left to right  if (transform.position.x < GameArea.xMin)  {  Vector3 position = Rigidbody.position;  position.x = GameArea.xMax;  Rigidbody.position = position;  }  // From top to bottom  if (transform.position.y > GameArea.yMax)  {  Vector3 position = Rigidbody.position;  position.y = GameArea.yMin;  Rigidbody.position = position;  }  else  // From bottom to top  if (transform.position.y < GameArea.yMin)  {  Vector3 position = Rigidbody.position;  position.y = GameArea.yMax;  Rigidbody.position = position;  }  **}**  **private void OnDrawGizmos()**  **{**  // Draw a blue box (wire cube) around the game area  Gizmos.color = Color.blue;  Gizmos.DrawWireCube(GameArea.center, GameArea.size);  **}**  **}** |

Don’t forget to assign the variables in the inspector:

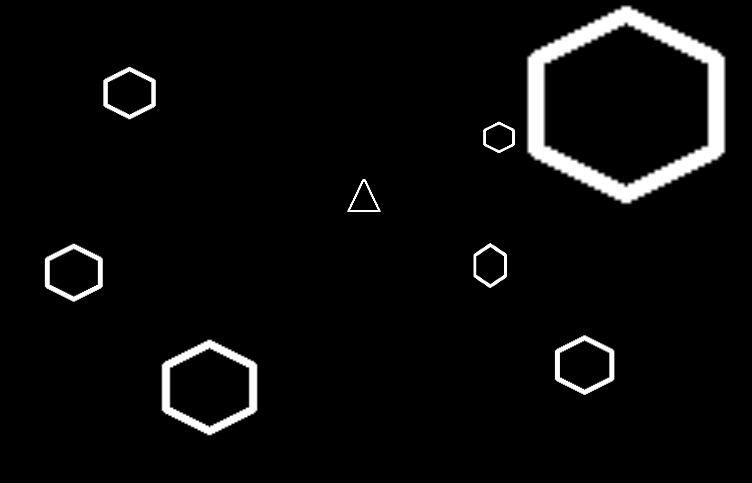


And that’s it! We have a spaceship and an asteroid! But it feels pretty empty. Do I have to go through all the process again to create another asteroid and fill the space with them? The answer is no! Unity handles that though the use of prefabs. To create a prefab, click and drag the element that you want to have available to copy as many times as you want to the project window. Easy peasy. Now, if you want to create more asteroids, just drag the prefab from the project window to the scene as many times as you want!



|  |
| --- |
| **📘 Prefabs**  Unity has a **Prefab** **asset** type that allows you to store a GameObject object complete with components and properties. The prefab acts as a template from which you can create new object instances in the scene. Any edits made to a prefab **asset** are immediately reflected in all instances produced from it but you can also *override* components and settings for each instance individually.  <https://docs.unity3d.com/Manual/Prefabs.html> |

Now enjoy playing with your first spaceship game! Mass Effect who?



# Optional exercises

1.- Try creating an offline multiplayer mode for our game. **Hint**: Different spaceships need to pay attention to different controllers! Try modifying that

2.- Create a shooting star that moves constantly around the map using forces and Rigidbodies (not controlled by the player)

3.- Create simple 2D platformer game level using Rigidbodies and forces. This time the gravity can’t be 0!

4.- Brave enough? Try creating the catapult mechanic from Angry Birds using Rigidbodies! You’ll need to study some vector maths for this!