**TEMASEK POLYTECHNIC**

**SCHOOL OF INFORMATICS & IT**

**DIPLOMA IN IMMERSIVE MEDIA & GAME DEVELOPMENT**

**AY2024/2025 OCTOBER SEMESTER**

**GADV (CGE2C25)**

**Introduction to Unity**

**Runtime Scripting**

To see the additional comments and resources, make sure you select **All Markup** in the **Review/Tracking** pane



**Objectives**

In this worksheet, you should develop a good understanding of Unity’s component-based architecture. You will learn the key differences between component-based design and traditional object-oriented programming (OOP), exploring how Unity’s approach allows for more modular and flexible game development.

Through hands-on exercises, you will create and modify GameObjects, both manually in the Unity Editor and dynamically through C# scripting. You will also learn how to add and configure Components, use public variables to expose values in the Inspector, and write simple scripts to modify an object’s properties, such as its size and color.

By the end of this worksheet, you should be able to create GameObjects, attach components, and manipulate their behaviour in a structured and reusable way.

Have fun! 😊

1. **Runtime Scripting**

In the previous worksheet, you wrote some simple scripts to change the colour of a sprite when the player presses the space bar.

This is called runtime scripting. This means that the script affects some part of the game scene as the game runs.

You can also call this *gameplay scripting*.

There are different types of scripting, which we can think of in general as:

* **Runtime scripting** (for in-game behaviours, logic, and interactions).
* **Editor scripting** (for extending and automating the Unity Editor, e.g. creating custom panels).
* **Tool programming** (if creating tools for designers or developers).
* **Procedural programming** (if generating/modifying content dynamically).

We’ll focus on runtime scripting.

* 1. **A Simple Script**

Previously, you saw that components on GameObjects are just scripts. You also saw that a component script can reference the other components on the same GameObject.

You did this to:

* change the sprite colour when the space bar is pressed.
* access the Transform component to make the sprite rotate

Now, we’ll write a script to **change the image** displayed by a sprite.

First, we need to figure out how to do this.

A good place to start is always the Unity documentation! Look at the Unity documentation for [SpriteRenderer](https://docs.unity3d.com/6000.0/Documentation/ScriptReference/SpriteRenderer.html).

Scroll down to the **Properties** section.

Notice this property:



Click on the sprite link.

The documentation states that:

The rendered sprite can be changed by specifying a different sprite in the sprite variable.

So what we must do is change the SpriteRenderer’s **sprite** property! This is done in the code below.

This code will access the **SpriteRenderer** component and set the value of its **sprite** property to **newSprite**.

spriteRenderer = GetComponent<SpriteRenderer>();

if (newSprite != null)

{

spriteRenderer.sprite = newSprite;

}

But where does **newSprite** come from?

We could load an image from a file, and then create a new Sprite object and set its texture to the loaded image.

Texture2D texture = Resources.Load<Texture2D>("Images/Asteroid");

Sprite newSprite = Sprite.Create(

texture,

new Rect(0, 0, texture.width, texture.height),

new Vector2(0.5f, 0.5f));

But that’s rather advanced for now.

Instead, we can use the Inspector.

1. Create this script:

using UnityEngine;

public class SpriteChanger : MonoBehaviour

{

    public Sprite newSprite; // Assign this in the Inspector

    private SpriteRenderer spriteRenderer;

    void Start()

    {

        // Get the SpriteRenderer component from this GameObject

        spriteRenderer = GetComponent<SpriteRenderer>();

        // Change the sprite if a new one is assigned

        if (newSprite != null)

        {

            spriteRenderer.sprite = newSprite;

        }

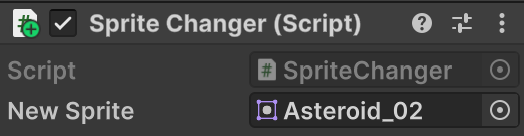
    }

}

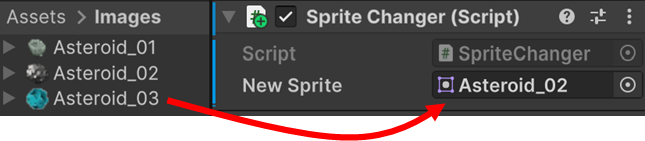
1. Add this script to one of the Asteroid GameObjects in your scene.

Select the Asteroid GameObject.

You will see the SpriteChanger script as a component in the Inspector, with a field for **New Sprite**:

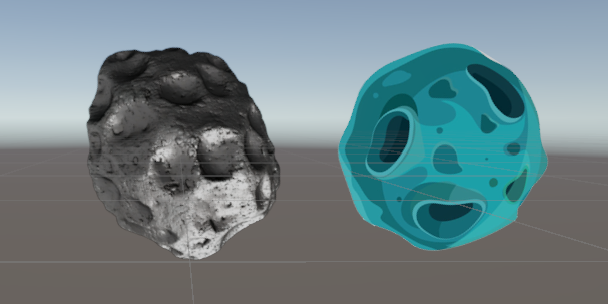


1. Drag a different sprite from the Images folder into the New Sprite field:



Run the project.

The GameObject sprite’s image should change:



Stop the project.

The sprite displays its original image!

This shows that the changes to the sprite only affect the game while it is running. After the game stops, the scene returns to its original state. This is why it is often called runtime scripting.

**Summary**

After this brief introduction to modifying the properties of GameObjects via script, we’ll look at how to create new GameObjects dynamically in code. This is a core part of game development.

From spawning enemies to firing a hail of projectiles in a bullet hell game, you will need to create new GameObjects dynamically as the game runs, often in response to user actions.

1. **Intantiating GameObjects**

**2.1. Creating an empty GameObject**

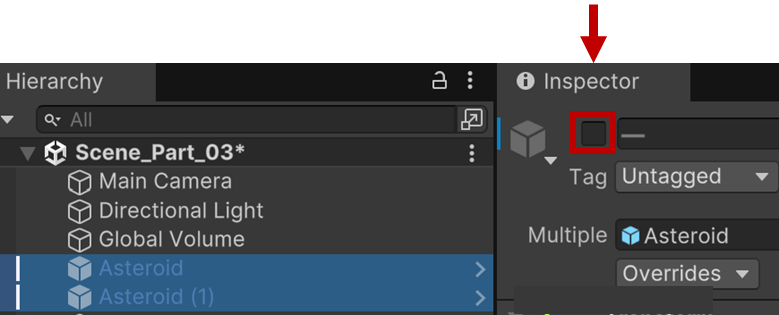
In the previous worksheets, you created GameObjects via the Unity Editor. Now, you’ll create a simple empty GameObject dynamically in code.

Continue with your current scene. You don’t need the existing sprite GameObjects, so hide them.

Select both GameObjects, then deselect the checkbox in the Inspector as shown in the screenshot below.

This will deactivate the GameObjects and hide them from the scene.

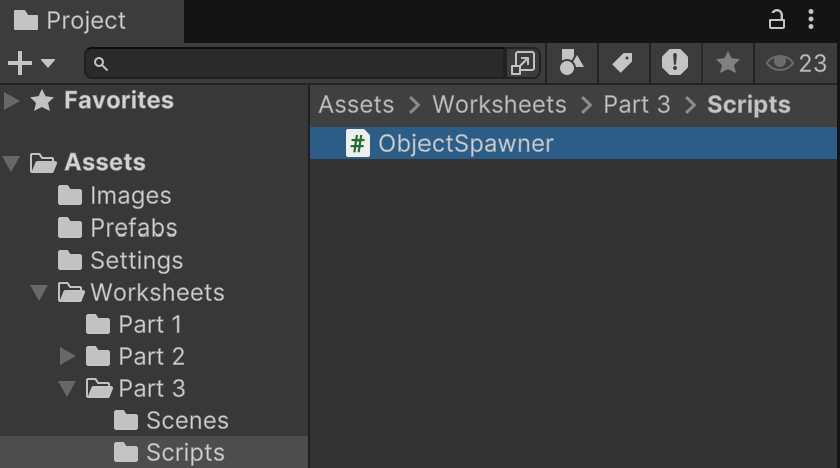
This also stops all the GameObject’s components from running automatically when the game starts.



Ok, now to create a GameObject using code!

1. Right-click inside the Scripts folder, and create a new MonoBehaviour script. Click on the name and change it to **ObjectSpawner**.

A new script called **ObjectSpawner.cs** will be created, as shown below. You don’t need to add the .cs when renaming the script.



Remember to check that Visual Studio is set as your external IDE. Check the previous worksheet for how to do this.

The code below creates an empty GameObject and adds it to the scene.

1. Modify your ObjectSpawner script:

using UnityEngine;

public class ObjectSpawner : MonoBehaviour

{

    void Start()

    {

        GameObject newObject = new GameObject("New GameObject");

    }

}

Run the project.

Nothing happens!

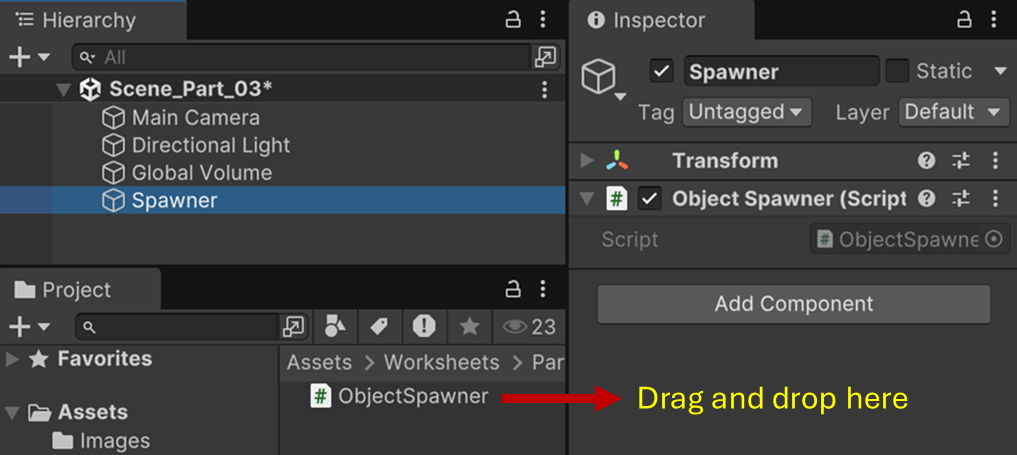
Why not?

Remember that scripts run as components on GameObjects. So, to run a script, it must be attached to a GameObject.

It’s very easy to forget to add scripts to GameObjects, so be careful.

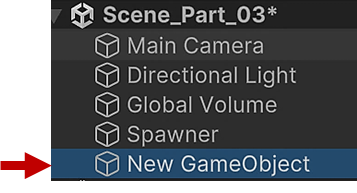
Let’s fix that now.

1. Right-click in the Hierarchy and create an empty GameObject named **Spawner**. Select the Spawner object, then drag the ObjectSpawner script into the Inspector window.



Run the project again.

Nothng seems to happen, but if you look closely you’ll see the newly created GameObject:



Great! You’ve just dynamically instantiated a new GameObject.

Let’s access its Transform component and change some properties.

Add this line to the Start method:

void Start()

{

GameObject newObject = new GameObject("New GameObject");

**newObject.transform.position = new Vector3(-5, 0, 0); // Add**

}

Now when you run the project and select the new GameObject, you’ll see that it’s positioned -5 units along the X-axis. But you can’t really see anything yet, since it’s just an empty game object. Let’s see how we can add a SpriteRenderer component to make it visible.

**2.2. Adding a Component via Code**

Modify your ObjectSpawner class:

public class ObjectSpawner : MonoBehaviour

{

**public Sprite newSprite;**

private SpriteRenderer spriteRenderer;

void Start()

{

GameObject newObject = new GameObject("New GameObject");

**spriteRenderer = newObject.AddComponent<SpriteRenderer>();**

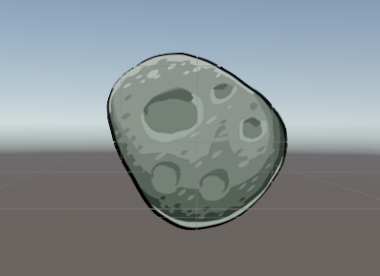
**spriteRenderer.sprite = newSprite;**

}

}

Remember the public Sprite variable from your previous worksheet. Set the value for this in the Inspector and run the project.

Now you’ll see a sprite with an asteroid image.



When you stop the project, the asteroid GameObject is removed. Remember that changes to the scene only occur while the game is running.

This is the basics of how you can create complex GameObjects dynamically in your code:

1. Create the base empty GameObject.
2. Add the required components via the AddComponent method.
3. Set the component properties to their required values.

But it’s a hassle to have to add every component in code and then set the required properties.

Shouldn’t there be a more efficient way to do this?

In a previous worksheet you learned about **prefabs**. A prefab is like a template for complex GameObjects. You already created an Asteroid prefab. We’ll create an instance of that directly in our code.

Modify your ObjectSpawner class:

public class ObjectSpawner : MonoBehaviour

{

    public GameObject AsteroidPrefab;

    void Start()

    {

        GameObject newAsteroid = Instantiate(

AsteroidPrefab,

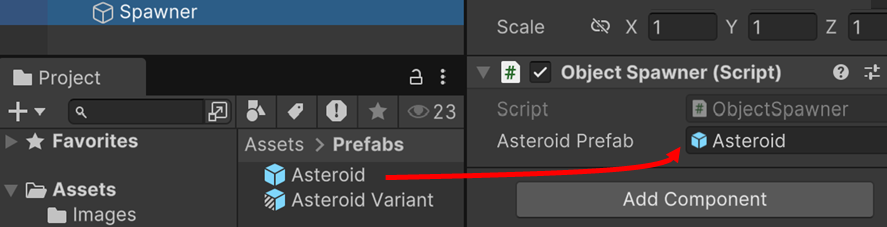
new Vector3(0, 0, 0), Quaternion.identity);

    }

}

From the **Prefabs** folder, drag the **Asteroid** prefab into the **Asteroid Prefab** field in the Inspector.

This sets the value of the **AsteroidPrefab** variable.



The code then uses the **Instantiate** method to create a new instance of the Asteroid prefab.

Note that we need to declare the prefab as a **GameObject** type of variable. The Instantiate method creates a new instance of this prefab in the scene.

Look at the documentation for the [Instantiate](https://docs.unity3d.com/6000.0/Documentation/ScriptReference/Object.Instantiate.html) method.

This function makes a copy of an object in a similar way to the Duplicate command in the editor. If you are cloning a GameObject you can specify its position and rotation (these default to the original GameObject's position and rotation otherwise).

Run the project. You should see that an Asteroid instance is created, with all the functionality in place (i.e. its rotate behaviour, and setting the colour to blue on pressing the space bar).

You will rarely create a complex GameObject from scratch.

The usual workflow is to first create a prefab, and then instantiate copies of this in script.

You’ll do this a lot once you get started on your project!

For fun, let’s instantiate a more complex prefab.

1. Download **FlyingCube** from LMS into your Prefabs folder.
2. Drag FlyingCube into the the ObjectSpawner’s **Asteroid** field in the Inspector.
3. Run the project.

You should see a cube that flies around and rotates randomly as if blown by a strong wind. It also changes colour as it moves.

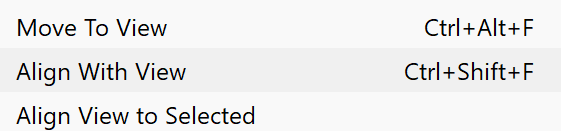
A square object in the middle of a flat field

Description automatically generated

If you can’t see it, zoom the camera out in the Scene view by scrolling the middle mouse button.

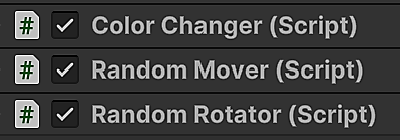
Click on **Main Camera** in the Hierarchy and then select the menu option:

* **GameObject/Align With View**



This will make the Game view align with the Scene view.

Double-click on the FlyingCube prefab to open the Prefab Editor. In the Inspector, you’ll see that it has three script components:



So far, you’ve explored basic scripting by modifying component properties, creating GameObjects from scratach, and instantiating prefabs.

Now complete the exercises below.

**Exercise 1: Create multiple instances**

The script below should create multiple asteroids in random positions.

Fill in the blanks.

using UnityEngine;

public class ObjectSpawner : MonoBehaviour

{

public **??? ???**;

public int **???** = **???**; // Number of asteroids to spawn

public Vector2 spawnRangeX = new Vector2(-5f, 5f); // X pos range

public Vector2 spawnRangeY = new Vector2(-3f, 3f); // Y pos range

void Start()

{

**???**;

}

void SpawnAsteroids()

{

for (int i = 0; i < numberOfAsteroids; i++)

{

Vector3 randomPosition = new Vector3(

Random.Range(spawnRangeX.x, spawnRangeX.y),

Random.Range(spawnRangeY.x, spawnRangeY.y),

0f);

GameObject asteroid = **???**(

**???**,

randomPosition,

Quaternion.identity);

}

}

}

**Exercise 2: Add a RigidBody component**

We’ll look at Unity Physics later, but for now just know that to make a 2D GameObject (such as a Sprite) interact with the Unity Physics engine, we must add a **RigidBody2D** component.

You know how to add a **SpriteRenderer** component. Now add a RigidBody2D component to each asteroid GameObject from Exercise 1 and run the project.

What happens?

**Exercise 3: Destroy GameObjects**

You can destroy a GameObject using the [Destroy](https://docs.unity3d.com/6000.0/Documentation/ScriptReference/Object.Destroy.html) method.

Modify the code so that each asteroid is destroyed after a random time from 0 to 5 seconds.

Look at the code above for how to create random values.

**Summary**

In this worksheet, you explored **runtime scripting** in Unity, focusing on how to create and modify GameObjects dynamically.

You learned about Unity’s **component-based architecture** and how it differs from traditional object-oriented programming (OOP).

Through practical examples and exercises, you applied scripting techniques to manipulate GameObjects at runtime, reinforcing core programming concepts in game development.

You gained experience in:

* **Modifying GameObjects via scripting**, such as changing a sprite dynamically.
* **Creating new GameObjects in code**, instead of manually adding them in the Unity Editor.
* **Adding and configuring components at runtime**, such as attaching a SpriteRenderer or Rigidbody2D.
* **Instantiating prefabs**, a crucial technique for efficiently spawning objects in a game.
* **Destroying objects after a delay**, using Destroy() with a random time value.
* **Instantiating** (potentially) complex prefabs instead of constructing complex GameObjects from scratch in code.

These core skills will help you build more interactive and dynamic game elements in future projects.

