**TEMASEK POLYTECHNIC**

**SCHOOL OF INFORMATICS & IT**

**DIPLOMA IN IMMERSIVE MEDIA & GAME DEVELOPMENT**

**AY2025/2026 APRIL SEMESTER**

**GADV (CGE2C25)**

**Unity Physics**

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



Many games include interactive objects that must behave like their real-world counterparts—like a pool ball rolling on a table.

To simulate this, Unity provides built-in 2D and 3D physics engines. You'll begin with the 3D engine before exploring the 2D one.

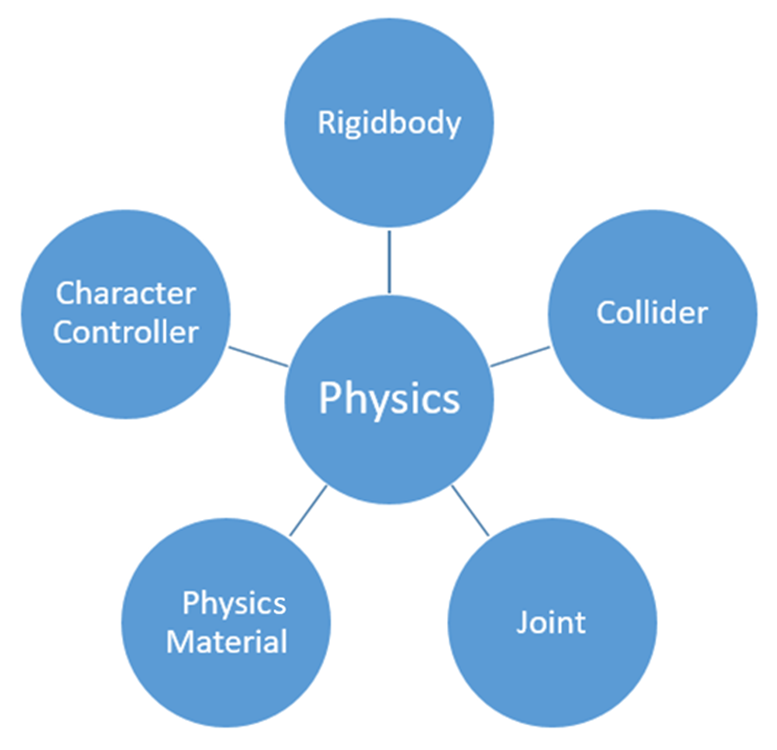
In this worksheet, you’ll learn how to use Unity’s Rigidbody, Collider, and Joint components to simulate realistic movement and interaction.

You’ll apply different types of forces to objects, control characters using the CharacterController, and detect and handle collisions through scripting.

You’ll also use raycasting for line-of-sight checks, and create basic 2D and 3D physics-based interactions suitable for gameplay.

1. **Unity Physics Engine Components**

The Unity physics components we’ll look at are:



You’ll look at the basic theory of each of these, answers some questions, then create a simple physics demo.

**Rigidbodies**

In physics, a rigid body is a solid body in which deformation is zero or so small it can be neglected. The distance between any two given points on a rigid body remains constant in time regardless of external forces exerted on it. A rigid body is usually considered as a continuous distribution of mass.

A rigid body is a simplification of a physical body in the real world, so it’s easier to simulate mathematically.

This means that it’s faster to simulate, too.

Unity models rigid bodies with the [Rigidbody](https://docs.unity3d.com/ScriptReference/Rigidbody.html) component.

When bodies are deformable, these are modelled using soft body physics.

Read these forum discussions: [forum1](https://stackoverflow.com/questions/32917449/how-to-move-an-object-without-going-through-colliders) [forum 2](https://answers.unity.com/questions/61401/pong-aipaddle-controller-help.html)

Answer **Question 1** in the **WORKSHEET** document.

**Colliders**

[Collider](https://docs.unity3d.com/Manual/CollidersOverview.html) components define the shape of an object for the purposes of physical collisions.

A collider is responsible for detecting when an object collides with another object.

A collider basically defines a shape in space.

If object A’s collider intersects with object B’s collider, then the objects are said to have collided.

In FPS games, objects can be hit by bullets and other ammo. This is called hit detection. Colliders also allow hit detection with other objects called rays, which we’ll look at later.

Look at this [webpage](https://answers.unity.com/questions/42558/whats-the-difference-between-a-rigid-body-and-a-co.html), then read this forum [thread](https://gamedev.stackexchange.com/questions/44131/should-i-be-sharing-data-between-graphics-and-physics-engine-in-the-game).

Answer **Question 2** in the **WORKSHEET** document.

**Joints**

A [Joint](https://docs.unity3d.com/Manual/Joints.html) component connects a Rigidbody to another Rigidbody or a fixed point in space. Joints apply forces that move rigid bodies, and joint limits restrict that movement.

There are many different types of joints. Also note that the Unity 2D physics engine has [more types](https://docs.unity3d.com/Manual/Joints2D.html) of joints than the 3D engine.

Watch this [video](https://www.youtube.com/watch?v=oAWGs0kt_vM).

Answer **Question 3** in the **WORKSHEET** document.

**Physics Materials and Physics Material 2D**

The [Physics Material](https://docs.unity3d.com/6000.0/Documentation/Manual/class-PhysicsMaterial.html) is used to adjust the friction and bouncing effects of the colliding objects.

Watch [video1](https://www.youtube.com/watch?v=SuUNnwswH94) and [video2](https://www.youtube.com/watch?v=FVAMRARJNbg).

Answer **Question 4** in the **WORKSHEET** document.

**Character Controller**

A [CharacterController](https://docs.unity3d.com/ScriptReference/CharacterController.html) allows you to easily do movement constrained by collisions without having to deal with a rigidbody.

Watch this [video](https://youtu.be/e94KggaEAr4?t=58) from **iHeartGameDev**.

Answer **Question 5** in the **WORKSHEET** document.

1. **Movement and Forces**

Forces are used to move Rigidbodies.

Force can be linear (applied in a straight line) or angular (rotational force, more technically called **torque**).

**2.1** [**Constant Force**](https://docs.unity3d.com/ScriptReference/ConstantForce.html) **component**

This is a small physics utility class used to apply a continous force to an object.

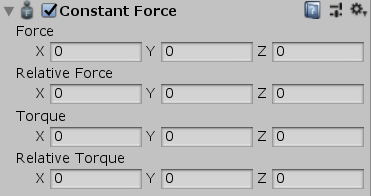
You can add a component that will add a constant force to a rigidbody.

* Create a new Unity project.
* Add a plane at 0, 0, 0, and scale it x2 in X and Z.
* Add a cube at 0, 0, 0.
* Lift the cube up so that it isn’t touching the plane.
* Add a Rigidbody to the cube, and set **Use Gravity** to false.
* Add a **Constant Force** component to the cube.

Your scene should look something like this:



Click on the cube, and look at the **Constant Force** component in the Inspector.



Complete the exercises below. You’ll have to submit answers to some questions about these, so make sure you think about what you’re doing!

* Set the Force X value to 1.

Run the scene. What happens?

* Set all the values back to 0.
* Set the Torque X value to 1.

What happens?

* Set the Relative Force X value to 1.
* Run the scene.

Is this any different from Force X?

* Now rotate the cube around its Y axis.
* Run the scene with a Relative Force X value of 1.

Now what happens?

Answer **Question 6(a)** in the **WORKSHEET** document.

Now set Use Gravity to true for the cube’s rigidbody.

* Set Force X to 1.

Answer **Question 6(b)** in the **WORKSHEET** document.

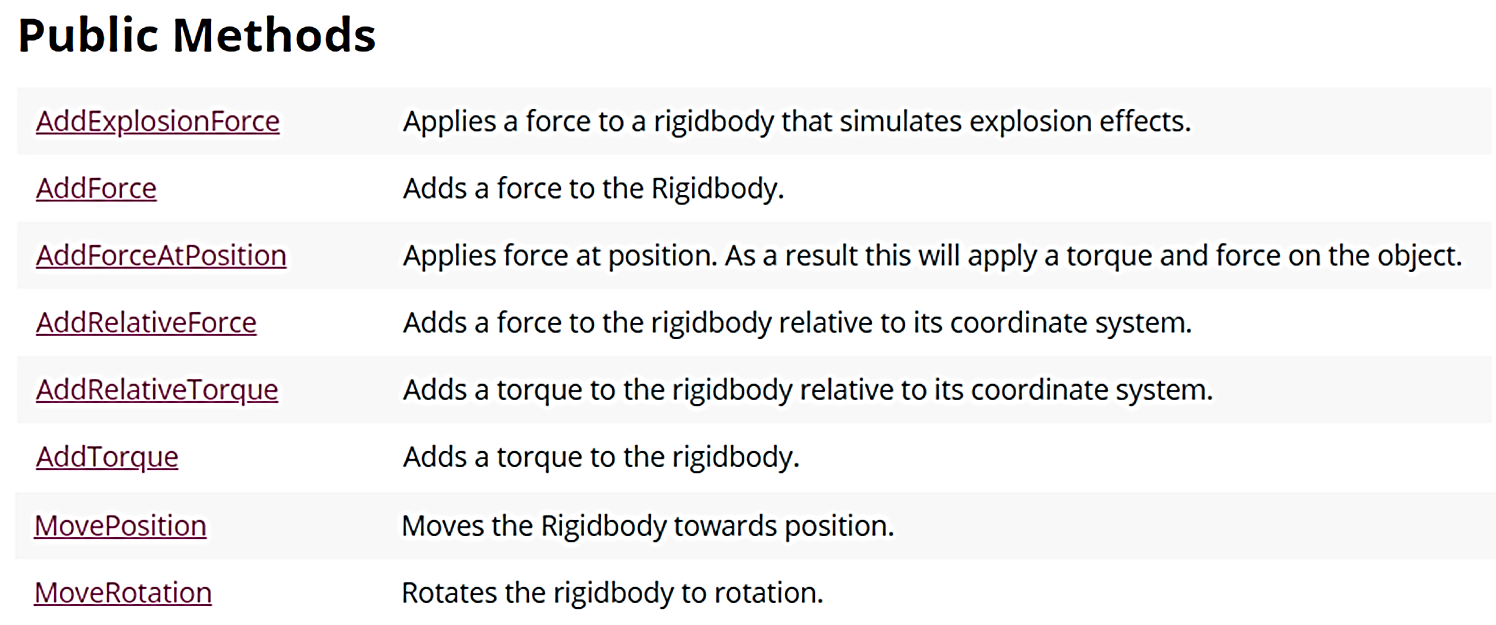
* Apply a constant force to the object along its local positive Z axis.

Answer **Question 6(c)** in the **WORKSHEET** document.

The Constant Force component is good for any object that you want to accelerate constantly in the scene, such as a rocket or missile. It isn’t appropriate for objects like projectiles such as bullets, which have an initial force applied to start them moving, but no more force after that (bullets don’t get faster as they move towards their target!).

**2.2 Applying Forces to a Rigidbody**

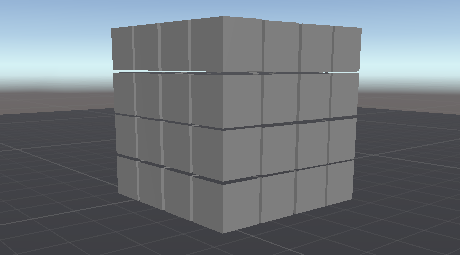
The [RigidBody](https://docs.unity3d.com/ScriptReference/Rigidbody.html) class has various functions for applying a linear or rotational force.



**AddExplosiveForce**

We’ll look at **AddExplosionForce** first—since it’s first in the list above, but also because it’s fun to play with.

* Create a new folder called Forces.
* Create a new scene inside the Forces folder.
* Add a cube to the scene.
* Attach a rigidbody to the cube.
* Create a prefab from the cube.
* Add an empty gameobject to the scene called **Cubes**.
* Using the prefab, create a stack of 4x4 cubes on a plane, as shown below. The empty GameObject Cubes should be the parent of all the stacked cubes.



* Add a capsule gameobject, called player.
* Add a CharacterController to the player.
* Watch this [video](https://www.youtube.com/watch?v=GWbMQQAliVw), then add a **Player.cs** script to the player that [moves](https://docs.unity3d.com/ScriptReference/CharacterController.Move.html) the player around using its CharacterController component.
* When the E key is pressed, an explosive force is applied to all rigidbodies with a given radius of the player. The radius and force applied should be set in the Inspector. If each cube has a mass of 1, you can start off with a force of 100.

Player.cs should look be structured like this:

using UnityEngine;

using System.Collections;

public class Player : MonoBehaviour

{

CharacterController controller;

// Your variables used for moving the player

// Code to set the radius and force of the explosion

public float radius = 5.0F;

public float power = 10000.0F;

void Start()

{

controller = GetComponent<CharacterController>();

controller.detectCollisions = false; // See comment about this!

}

void CheckExplosion()

{

if (Input.GetKeyDown(KeyCode.E))

{

// Your code to generate the explosion

}

}

void FixedUpdate()

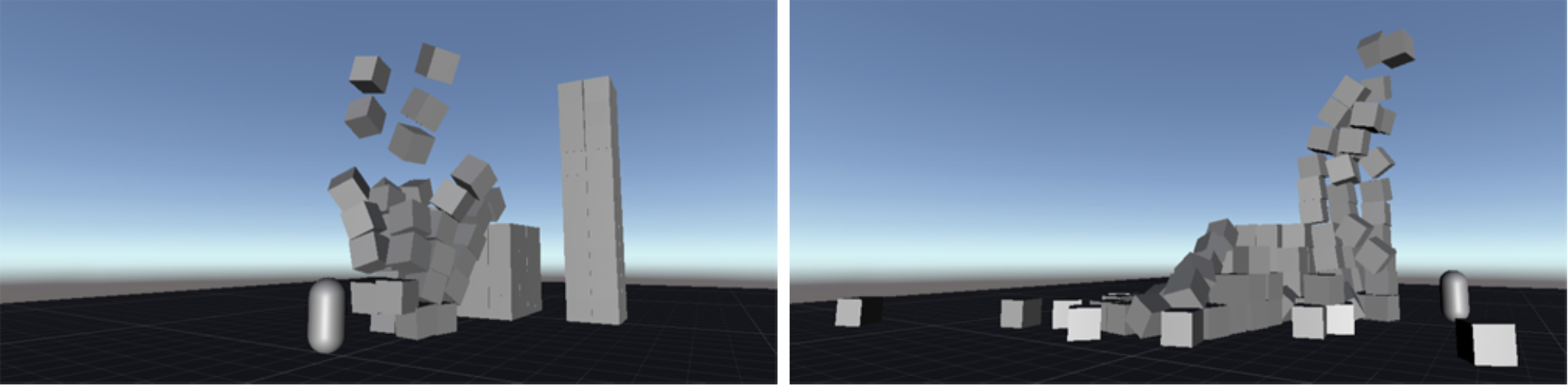
{

CheckExplosion();

// Your code to move the player

}

The player should be able to blast the cubes that are within the set radius.



Answer **Question 7(a)** in the **WORKSHEET** document.

Now add a function called **Kick** to the Player class. When the K key is pressed, this will apply a force in the player’s forward direction. Use a public variable **kickStrength** to set the force of the kick in the Inspector.

Answer **Questions 7(b)** and **7(c)** in the **WORKSHEET** document.

**AddForce & AddTorque**

**AddForce** and **AddTorque** are the most commonly used [Rigidbody](https://docs.unity3d.com/ScriptReference/Rigidbody.html) functions.

**AddForce**: Force is applied continuously along the direction of the force vector.

Answer **Question 8(a)** in the **WORKSHEET** document.

Create a scene as shown, with a plane, a sphere at (0, 0, 0), and a cylinder 5 units from the sphere in each of the left, right, forward, back and up directions.

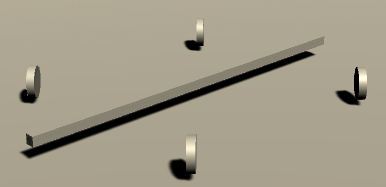


* Add a rigidbody to the sphere and cylinders. Set Gravity to 0 for the cylinders.
* Each cylinder should rotate constantly about its Y axis.
* Create a new script called Impulse.cs. Add this to the sphere.
* When the up arrow key is pressed, add an impulse force to the sphere so that it moves forward and hits the cylinder in front of it. The force applied should be set in the Inspector.
* When the down arrow key is pressed, add an impulse force to the sphere so that it moves backwards and hits the cylinder behind of it.
* Likewise for the left and right keys to hit the left and right cylinders.
* Use the space key to hit the cylinder above the sphere.

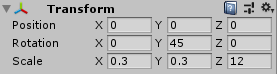
Answer **Question 8(b)** in the **WORKSHEET** document.

**AddTorque**: Torque is the rotational force that causes an object to spin around an axis.

* Modify the scene (or create a new scene) as shown below, to create a beam from a cube. Add a rigidbody to the beam and disable gravity.



Give the beam the following transform properties:



* When the Z key is pressed, make the beam rotate clockwise to hit the cylinders. Use an impulse force.
* When the X key is pressed, make the beam rotate anti-clockwise to hit the cylinders. Use an impulse force.

Answer **Question 8(c)** in the **WORKSHEET** document.

We won’t look at all the different Rigidbody force-related functions, but you should take some time to explore these yourself.

1. **Detecting and Handling Collisions**

In scientific simulations, forces are set at the start and left to run, but games need to react to things like collisions—for example, reducing health when a player hits an enemy.

This is a collision **event**, and our code must detect it and respond. This is called event-handling, a common technique in all software—for handling key presses, mouse clicks, and more.

Here, we’ll look at basic event-handling for collisions.

**Unity’s built-in collision handlers**

Event-handling runs a function—called an **event-handler**—whenever a specific event occurs.

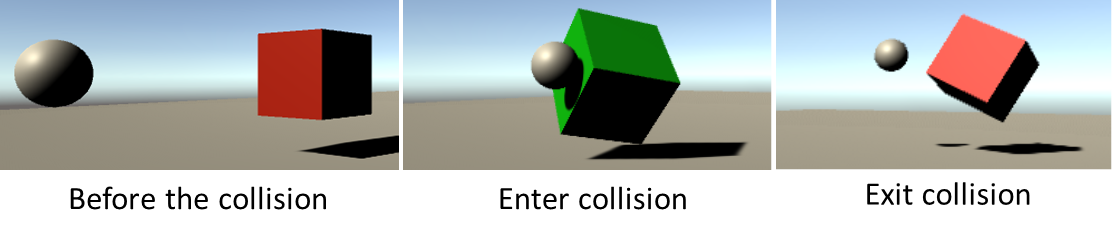
In Unity, when a collision happens, it looks for a script on the collider’s GameObject and calls the relevant collision event-handler.

Unity passes in collision data as an argument, stored in a class instance depending on the collision type.

This system of calling functions in response to events is called messaging.

The Unity **Collider** class has the event-handler functions listed [here](https://docs.unity3d.com/ScriptReference/Collider.html). Look at the section about **Messages**.

* Create a cube called CosmicCube, positioned at (0, 2, 0).
* Create a new material coloured red, and apply it to the CosmicCube.
* Add a rigidbody and turn off gravity.
* Make the CosmicCube rotate around all its axes constantly.
* Create a sphere called Orb, positioned at (0, 2, -5).
* Add a rigidbody and turn off gravity.
* Add a script called Orb.cs to the Orb.
* When the spacebar is pressed, the orb has an impulse applied along its forward direction (make the impulse value accessible in the Inspector).
* Add a script called CosmicCube.cs to the CosmicCube.
* When the CosmicCube is collided with, it turns green.
* When the CosmicCube stops being collided with, its colour returns to red.



Answer **Questions 9(a)** in the **WORKSHEET** document.

1. **Raycasting**

We’ve looked at the basics of the Unity 3D physics engine.

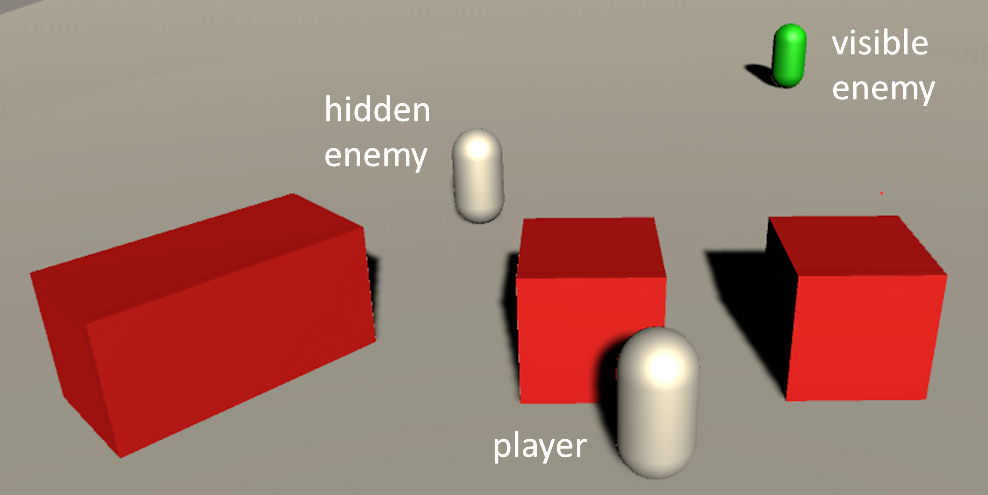
We’ve seen the core components, how forces are used, and how collisions are handled.

Finally, we’ll take a brief look at something called raycasting.

A ray is basically an invisible line (a vector, which we’ll look at later).

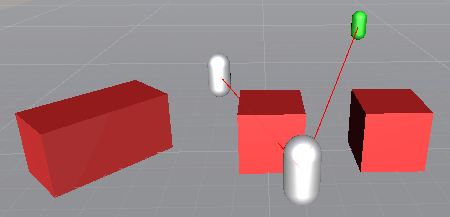
A ray has a magnitude (length) and the direction in which it points.

Here, we’ll look at how raycasting can be used for line of sight calculations.

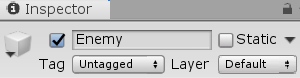


In the image above, the player is facing two enemies. One enemy is hidden by a red block, while the other enemy is visible to the player.

To determine if an enemy is in the player’s line of sight, we cast a ray (in red below) from the player to each enemy.



* Create 3 cubes and scale/position them to look like the scene above.
* Duplicate the player object, and remove the Player script component.
* Make a prefab from this new object, called Enemy.
* Add two instances of the new Enemy prefab to the scene, called Enemy1 and Enemy2.
* Create a new [tag](https://docs.unity3d.com/Manual/Tags.html), “Enemy”
  + Select the Enemy prefab.
  + Select the Tag dropdown in the Inspector:



* + Select Add Tag, click the + icon and add a tag called “Enemy”.



* + Set the Enemy prefab’s tag to “Enemy”.



* Add two enemy objects as shown in the scene above. Note that since you set the Enemy prefab’s tag to “Enemy”, the enemy object are also tagged as “Enemy”.
* One enemy should be hidden from the player behind a cube. The other enemy should be visible to the player.
* Open Player.cs and add the function below.

Look at the skeleton code below, and answer **Questions 10** in the **WORKSHEET** document.

void CheckLineOfSight()

{

/\* add your code here \*/ enemies =

GameObject.FindGameObjectsWithTag("Enemy");

[RaycastHit](https://docs.unity3d.com/ScriptReference/RaycastHit.html) hitData;

foreach (/\* add your code here \*/ enemy in enemies)

{

Vector3 vec = enemy.transform.position - transform.position;

Debug.DrawRay( /\* add your code here \*/ );

if (Physics. /\* add your code here \*/ (transform.position,

vec, out hit, 30f))

{

if (/\* add your code here \*/)

{

/\* add your code here \*/ = Color.green;

}

}

}

}

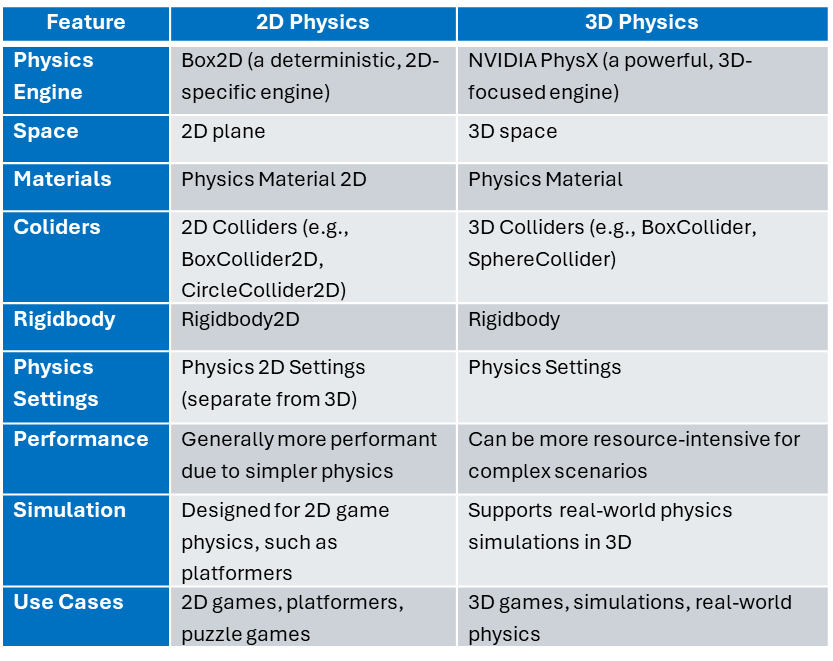
Call this function in **Start**. Play around with the enemies and add more cubes!

**Unity 2D Physics**

Unity has a separate physics engine for 2D games. The components are similar to those in 3D physics, but they work only on 2D GameObjects (sprites), and use the Physics2D API.

**Important: Don’t mix 3D and 2D physics components — 2D components won’t interact with 3D ones.**

**Comparison between 2D and 3D physics**



**Exercise 1: Bouncing Ball Demo (2D)**

Goal: Make a ball bounce around the screen using Rigidbody2D and a Physics Material 2D.

1. Create a new scene called **2DExercise1**.

We want to configure the scene for 2D.

Follow these steps:

1. Select **Main Camera** in the Hierarchy.
2. In the **Inspector**:

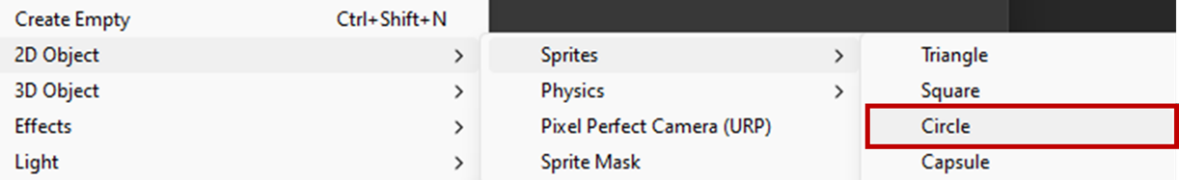
* Set **Projection** to **Orthographic**.
* Set **Position** to (0, 0, -10) (facing XY plane).
* Set **Rotation** to (0, 0, 0).

1. In the top-right of the Scene window, click the **2D toggle**. (This locks the view to the XY plane.)



Now we add the sprite GameObjects.

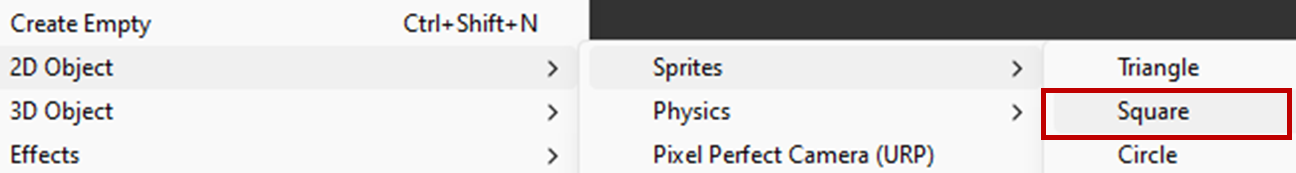
1. Add a **Circle Sprite** (e.g., a circle from Unity's default 2D shapes) called **Ball**.



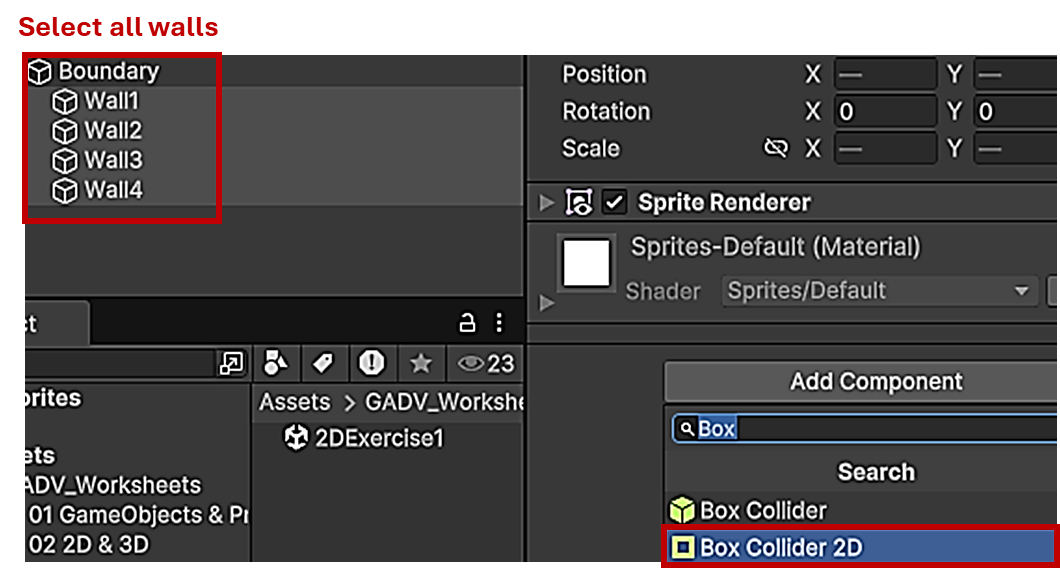
1. Add a **Rigidbody2D** and a **CircleCollider2D** to it.
2. Create a new **Physics Material 2D** and set:

* Friction = 0
* Bounciness = 1

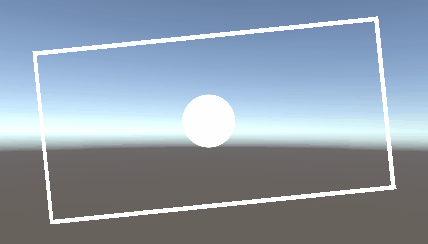
1. Assign this material to the ball’s collider.
2. Create 4 sprites (thin, wide rectangles) and position them to form a box boundary.



1. Add a **BoxCollider2D** to each wall, but with no Rigidbody. Note that it’s good practice to create an empty GameObject to group related objects together.



1. Press Play and watch the ball bouncing inside the box.
2. Rotate the boundary and press Play again.



**Optional:**

* Give the ball an initial velocity using **Rigidbody2D.velocity = new Vector2(x, y);** in a script.
* Experiment with gravity scale.

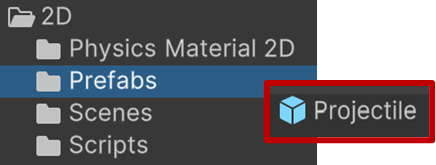
**Exercise 2: 2D Projectile Launcher**

Goal: Launch a 2D projectile using AddForce() and experiment with ForceMode2D.

1. Create a new 2D scene.
2. Add a **Capsule Sprite** and name it **Player**.



1. Add a small **Circle Sprite** and name it **Projectile**.
2. Add **Rigidbody2D** and **CircleCollider2D** to the projectile.
3. Create a **Prefabs** folder, then drag the **Projectile** GameObject into it to create a prefab.



1. Delete the **Projectile** GameObject from the scene.
2. Write a script **Launcher.cs**:

using UnityEngine;

public class Launcher : MonoBehaviour

{

public GameObject projectilePrefab;

public Transform firePoint;

public float force = 20f;

void Update()

{

if (Input.GetKeyDown(KeyCode.Space))

{

GameObject clone = Instantiate(

projectilePrefab,

firePoint.position,

Quaternion.identity);

Rigidbody2D rb = clone.GetComponent<Rigidbody2D>();

rb.AddForce(firePoint.right \* force);

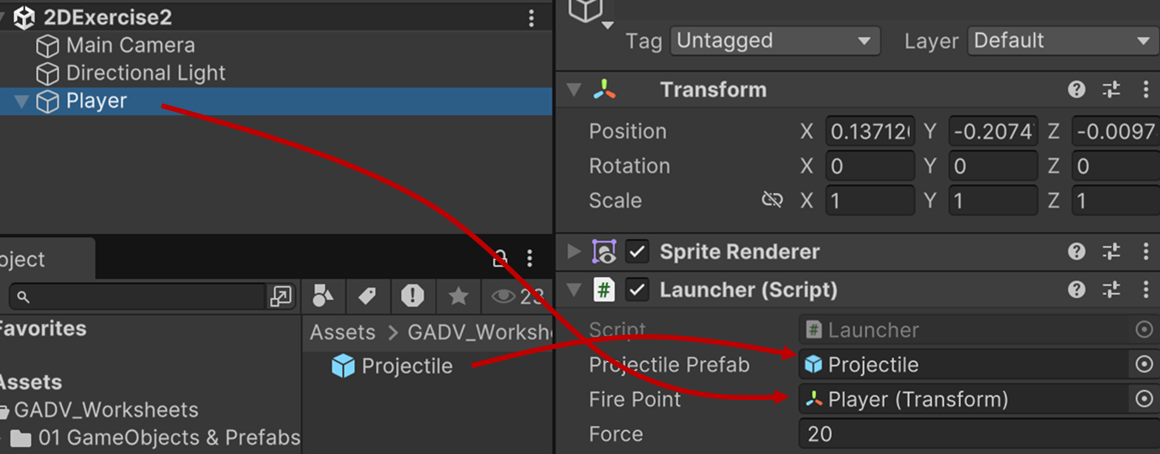
}

}

}

Note that the script has two public variables:

* projectilePrefab: the prefab that is instantiated every time the space bar is pressed.
* firePoint: the position from which the projectile will be fired. This is of type Transform, since a GameObject’s Transform has data for its position.
* Set the Fire Point to be the player’s Transform.
* Set Projectile Prefab to the Projectile prefab you just created.



1. Press Play.

Press the space bar to fire a projectile.

**Challenge Problems**

1. **Projectile lifetime**

Note that when you fire a projectile, the code create a new instance of the projectile prefab. These remain in the scene forever (each projectile just keeps moving until you press Stop). But in a real game, the projectile should hit something or move out of sight and be destroyed.

How could you implement that?

**Hint:** Could you check if the ball [collides](https://docs.unity3d.com/6000.0/Documentation/ScriptReference/MonoBehaviour.OnCollisionEnter2D.html) with another object and if so it destroys itself? Could you check the [distance](https://docs.unity3d.com/6000.0/Documentation/ScriptReference/Vector2.Distance.html) of the ball from its spawn point to decide if it should be destroyed?

1. **Projectile spawn location**

The player’s transform is used as the projectile spawn point. This means that the projectile is instantiated at the center of the player. But you may need to spawn the projectile from some other position relative to the player (e.g. from their hand).

How could you implement that?

**Hint:** Add an empty GameObject as a child of the player GameObject and set this to be the fire point.

1. **Projectile direction**

The projectile fired to the right, using:

rb.AddForce(firePoint.right \* force)

How could you set the projectile to be fired in any direction?

**Hint:** Could you apply a force to its rigidbody?

1. **Summary**

In this worksheet, you explored the core features of Unity’s 3D and 2D physics engines.

You learned how to:

* Control object movement using physics components such as Rigidbodies and Colliders
* Apply different types of forces
* Handle collision events
* Simulate real-world behaviours like jumping, bouncing, and explosions.
* Perform ray-tracing to simulate basic perception, i.e. line of sight.
* Use the CharacterController for scripted player control

By completing these exercises, you should now be able to:

* Choose the right movement technique for different gameplay scenarios.
* Apply forces and torques to objects using code.
* Use Unity’s physics materials to simulate friction and bouncing.
* Write scripts that respond to collisions and other physical interactions.
* Build simple 2D physics-based games using Rigidbody2D.

You should be able to apply these techniques to your own project.

If you have any questions, please ask your tutor.

If you like, you can also complete the **GADV Unity 3D Physics Game Tutorial** worksheet. This will help you to put what you’ve learned to good use in making a simple physics-based game.

Well done for getting this far! 😊