Exercises - Physics

Exercise 1

Create a new scene, add two spheres and a plane beneath them.

When you start the game, the position of the two spheres should be randomized.

When you press the space button, one sphere should be pushed (rigidbody and add force) towards the other sphere.

When they collide, print out a message.

Exercise 2, Whack a mole v2.0

This is again a whack a mole game.

Spawn spheres randomly.

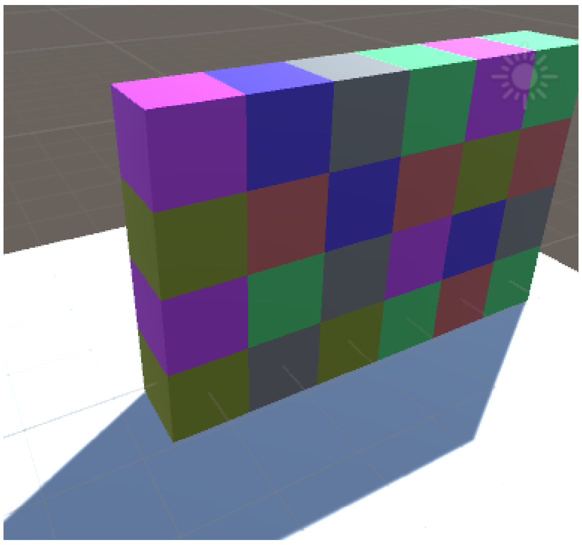
Use the following line of code to create a ray from the camera in the direction of the mouse. If you click an object in the game window, the ray should hit the object. Use ray casting to check if an object was hit.

```
Ray ray = Camera.main.ScreenPointToRay (Input.mousePosition);
```

Spawn the objects randomly, and destroy them when clicking on them.

Exercise 3, Push boxes

Create a new scene. Stack a bunch of boxes on top of each other:



Use the ray casting from the previous exercise.

When your ray hits a box, add a force to the box to push it away.

Exercise 4, Simple character controller

In this exercise you will set up a simple character controller using physics.

Create a GameObject, e.g. a sphere.

Create a script, and add it to your GameObject.

Use the input (“horizontal” and “vertical”) to add forces to the GameObject so that it can move around. Hint: you may want to just set the velocity directly.

When “jump” is pressed, add an upward force to the GameObject to make it jump.

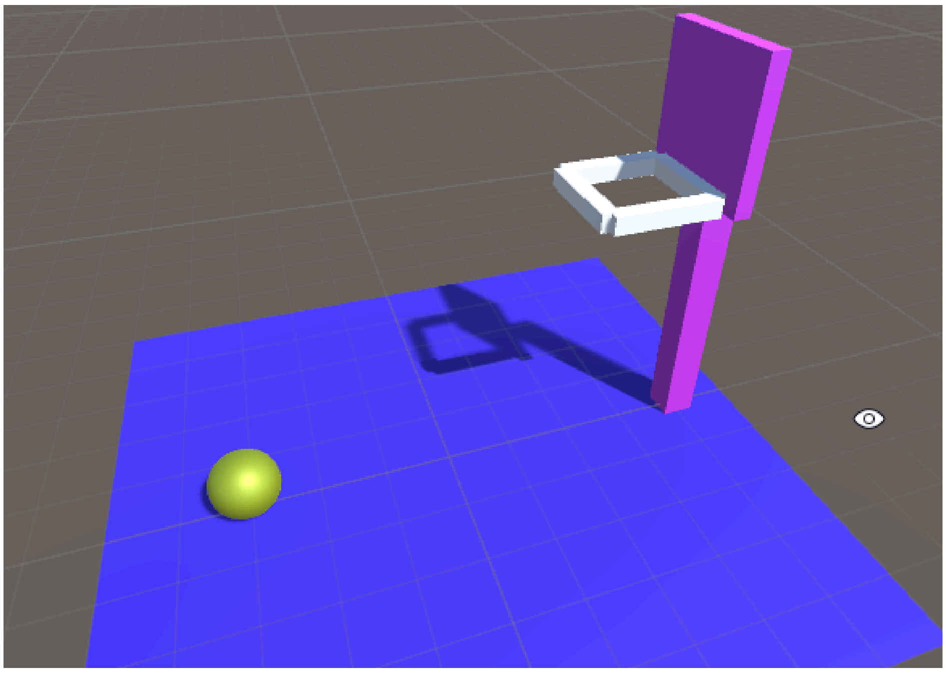
Add a few boxes you can jump up on.

You’ll notice that whenever space is pressed, you will jump, even though you’re not touching the ground.

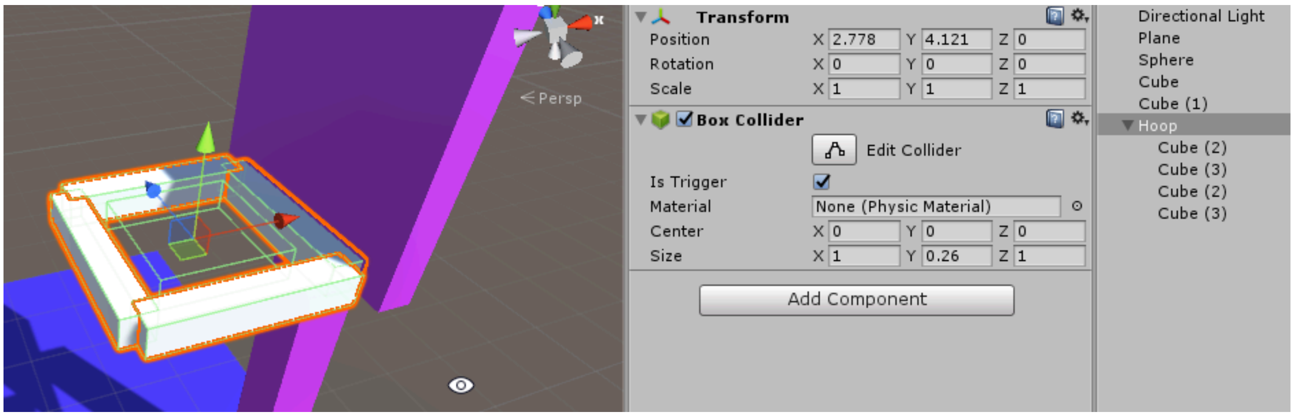
A simple fix to this, would be to check the “rigidBody.velocity.y” value, i.e. the vertical velocity. If it is 0, then you’re not moving vertically, and must be on the ground.

Exercise 5, Basketball

Create a scene similar to the one seen below:



In the ‘hoop’ there is a box collider, set as trigger:



```
Transform
Position     X 2.778   Y 4.121   Z 0
Rotation     X 0       Y 0       Z 0
Scale        X 1       Y 1       Z 1
Box Collider
             Edit Collider
Is Trigger   ✓
Material     None (Physic Material)
Center       X 0       Y 0       Z 0
Size         X 1       Y 0.26    Z 1
Add Component
```

```
Directional Light
Plane
Sphere
Cube
Cube (1)
Hoop
  Cube (2)
  Cube (3)
  Cube (2)
  Cube (3)
```

The idea is to hold down space to increase force, then release space to add the force to the ball. When it hits into the hoop, you get a point.

Use the Input.GetButtonDown method to register when space was pressed.

Use the Input.GetButtonUp method to register when space was released.

Keep track of the time between button down and button up, to figure out how long time space was held.

Define a direction for the force, make it a unit vector (Vector3::normalized), and then multiply with a constant, and then with the time the space was held.

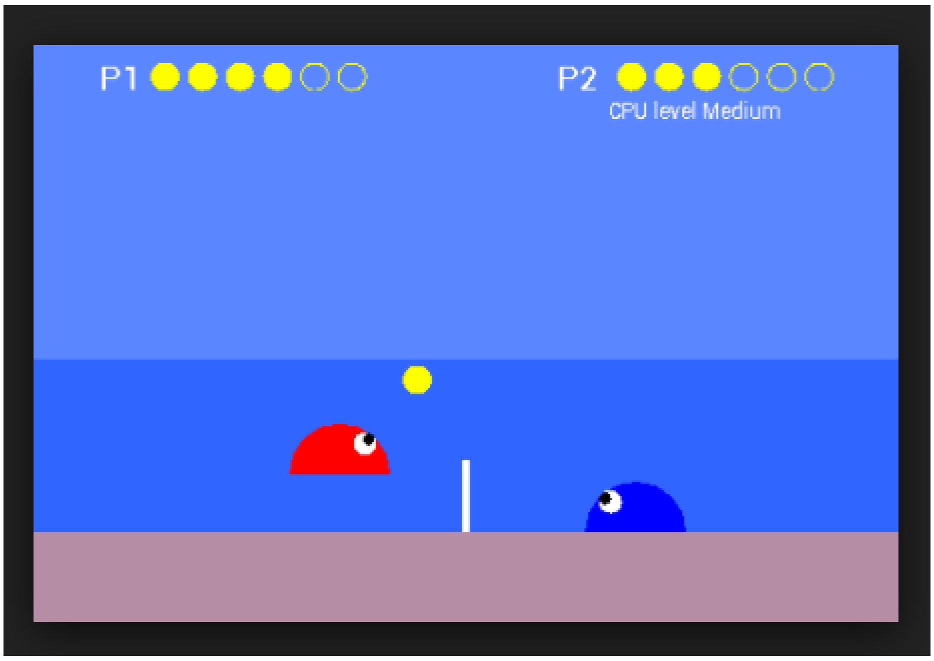
Aim the unit vector toward the hoop.

Create input to reset the ball after each throw, remember to set rigidbody.velocity = Vector3.zero, and rigidbody.angularVelocity = Vector3.zero, to reset the forces.

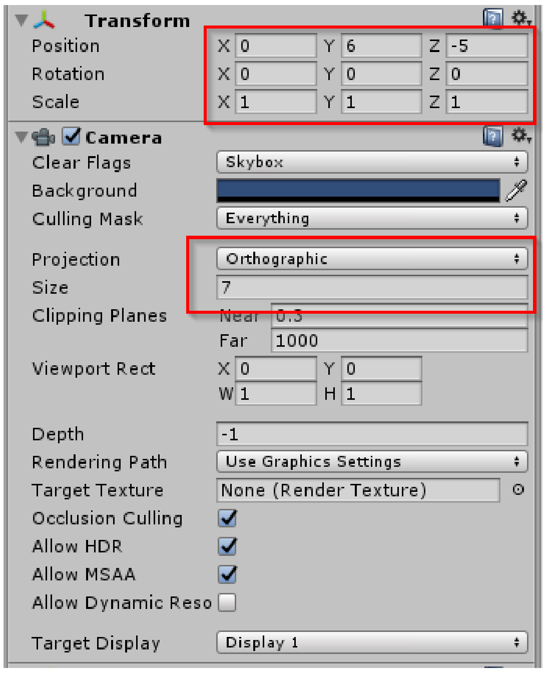
You can expand the above to be a multiplayer game. Instead of space, have player 1 use left control, and player 2 use right control. Add another hoop opposite the first. Keep track of both inputs now, and add force in the correct direction depending on whether player 1 or player 2 released the button.

Exercise 6, Volleyball

An homage to the old Slime Volley:



Create a new scene. This is a 2d game, so first fix the camera:



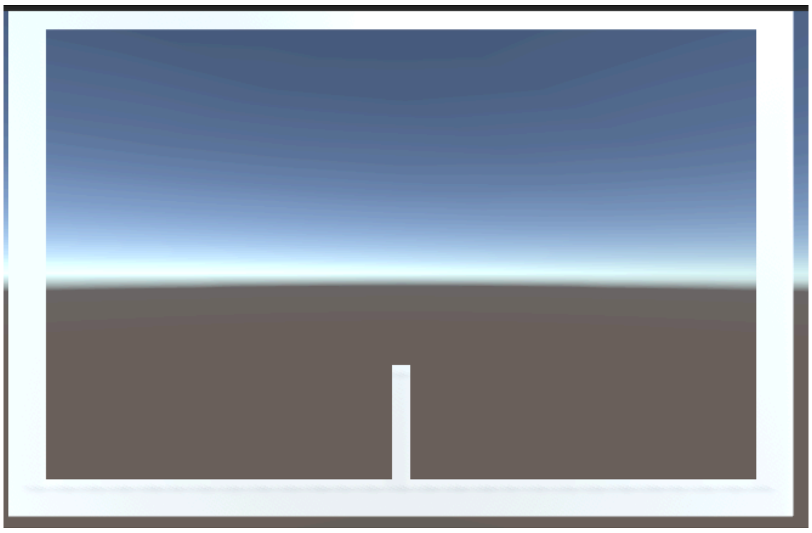
Transform

Position	X 0	Y 6	Z -5
Rotation	X 0	Y 0	Z 0
Scale	X 1	Y 1	Z 1

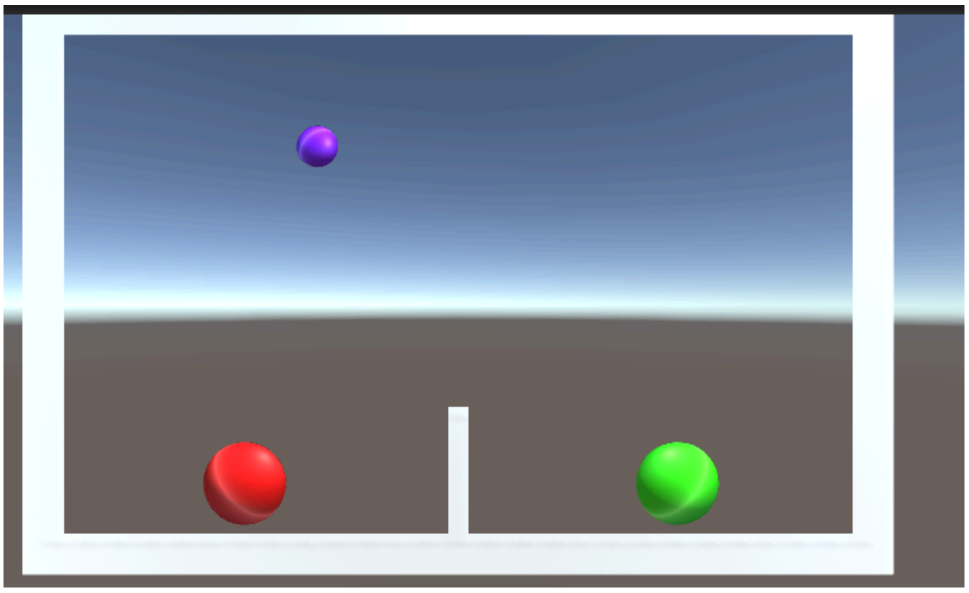
Camera

Clear Flags	Skybox
Background	
Culling Mask	Everything
Projection	Orthographic
Size	7
Clipping Planes	Near 0.3
	Far 1000
Viewport Rect	X 0 Y 0
	W 1 H 1
Depth	-1
Rendering Path	Use Graphics Settings
Target Texture	None (Render Texture)
Occlusion Culling	✓
Allow HDR	✓
Allow MSAA	✓
Allow Dynamic Reso	☐
Target Display	Display 1

Create a scene, similar to this:



Create three spheres:



Add rigid bodies, and lock the position in the z-direction on all three balls. You may need to lock rotations on the red and green sphere.

Create a script to control the red player, use ‘A’ and ‘D’ to move left and right. Use ‘W’ to add an upward force, to make the player jump.

Create a similar script to control the green ball, use e.g. “JIL” for controls.

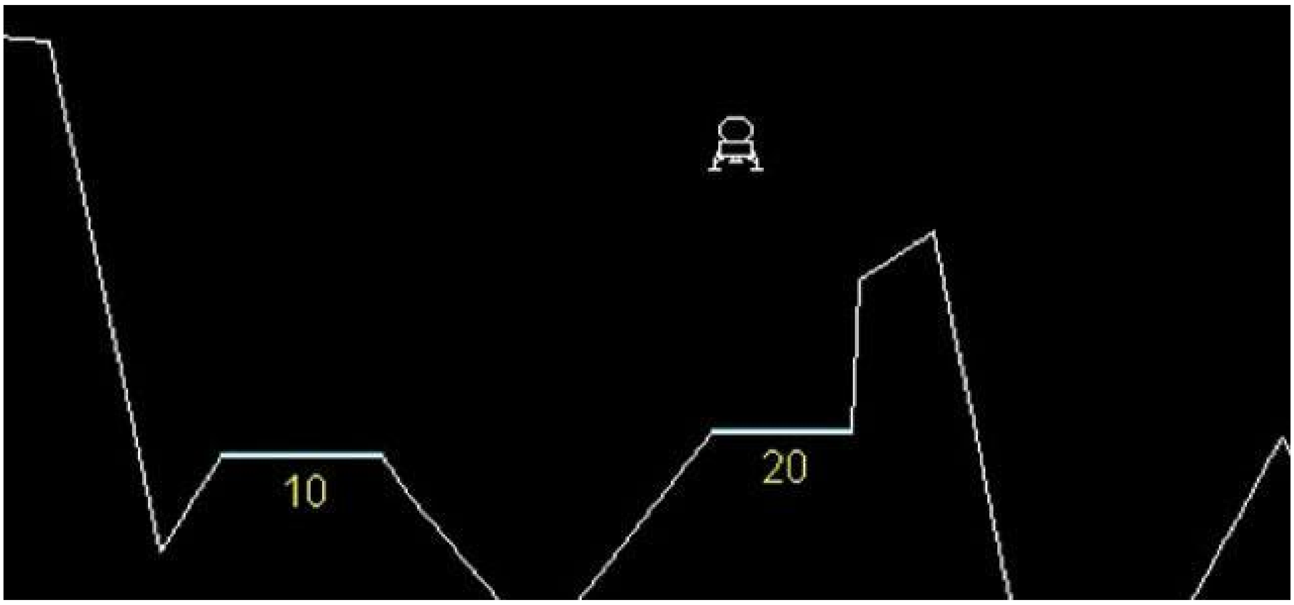
Add a physics material to the purple ball, make it bouncy. When the purple ball hits the red or green ball, it’s supposed to bounce back up in the air.

The floor is made of two boxes, one for the green side, one for the red side.

When the purple ball hits the ground, print out who won the point, then move the purple ball up in the air, and reset the velocity to start the game again.

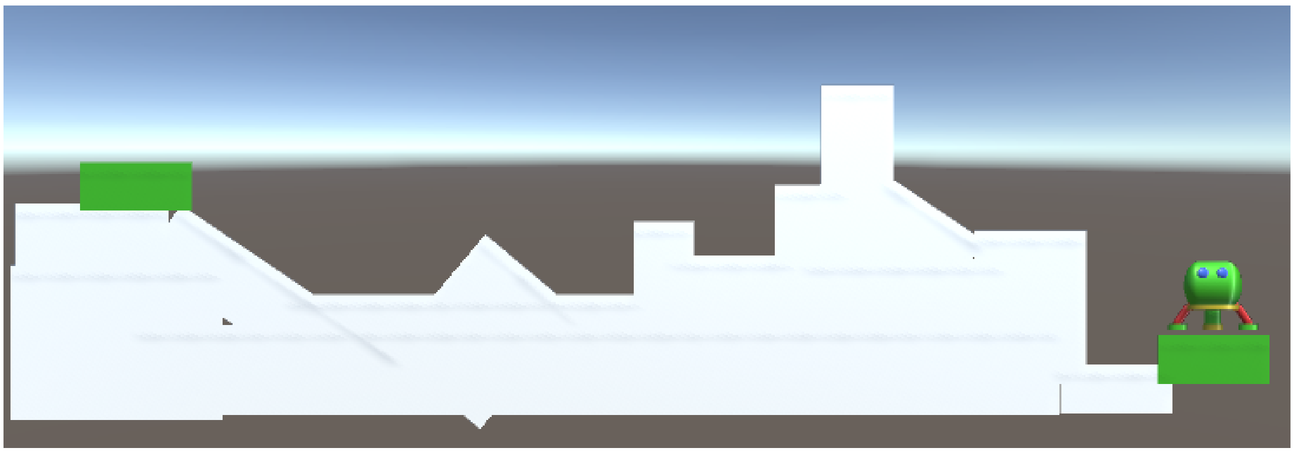
Exercise 7, Moon Lander

This exercise will have you remake the old classic:



This will be a 2d game. On the camera you can change 'perspective' to 'orthographic', similar to the previous exercise.

Create a scene and insert boxes, something like this:



You can import the uploaded Asset pack to get the beautiful Moon Lander model seen above, or you can just create your own.

Add a rigidbody to the MoonLander. In this game we need a little more control over the physics, so remove the Use Gravity tick from the rigid body, we will apply our own gravity.

You also need to set some constraints: All rotations, so that the Moon Lander cannot rotate. And lock one of the positions, probably z, but that depends on how you have positioned your stuff. The point is, we only want the Moon Lander to move up or down, and left or right, but not toward or away from the camera.

The rules: You use the arrows to apply force to the Moon Lander. The Moon Lander is only allowed to touch the green landing pads.

You will have to add colliders to your terrain, and check OnCollisionEnter, if the Moon Lander is touching terrain. You can use a tag for the terrain, and a tag for the landing pad, and in the OnCollisionEnter method check which you hit.

We're going to use the 'void FixedUpdate()' method, because we're going to apply forces.

First we need to add gravity: rigidBody.AddForce(Vector3.down * 30f).

Our gravitational acceleration is now 30, instead of Earth's 9.81. It will make the game feel a bit more snappy. You can play around with this.

Get input from the axes, in Update, and add force correspondingly in FixedUpdate to the MoonLanders rigidbody.

You should now be able to fly the Moon Lander to the other landing pad.

Increase difficulty:

Add fuel to your game. Have a float variable, which you decrease a little bit in the FixedUpdate method, whenever you're adding force to the Moon Lander.

You are not allowed to crash onto the landing pad, so we're going to check the velocity on impact.

You can do that like so:

```
void OnCollisionEnter(Collision col) {
    Debug.Log (col.relativeVelocity.magnitude);
}
```

This will print out the velocity of the Moon Lander upon impact. If it's larger than some threshold, say 5, you lose the game.

Exercise 8, Billiard

Create a billiard game. Build the billiard table out of boxes and planes.

Add triggers to the 'holes' to figure out when a ball goes in the hole.

Add bouncy physics materials to the table and the balls.

Create a top-down view, with an orthogonal camera.

The idea is you press and hold the mouse on a ball, then drag the mouse somewhere, release the button, and the ball will be pushed in the direction you dragged the mouse.

Use ray casting to check when you click on a ball. On mouse button down, get the mouse position:

Vector2 v = Input.mousePosition

On mouse button up, get the mouse position again. You now have two points. Calculate the direction as a vector, add that vector as a force to the ball's rigidbody. You may need to scale with some constant.

Destroy, or disable (gameObject.SetActive(false)), when they go into a hole.

Make sure you can only shoot the white ball. You can e.g. use tags.

Or you can look into the LayerMask. When you do a Physics.Raycast(...), one of the method headers takes a LayerMask. This is a list of Layers, which the ray can intersect. You create a public field variable of type LayerMask. Then in the inspector, you're able to select which layers you want.