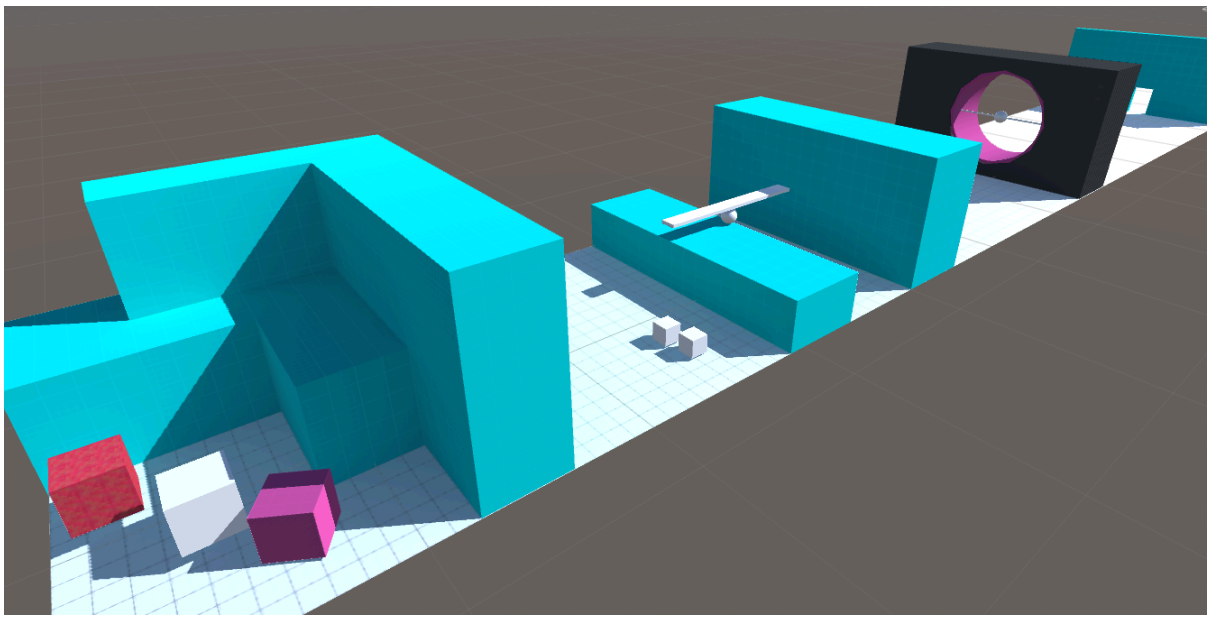
COMP4002/G54GAM Lab Exercise 04 – Playing with Physics

24/02/20

The next three labs consider three different forms of challenge and similarly explore different engine concepts:

- Inference and lateral thinking using the manipulation of physics objects
    - The physics engine
- AI Enemies that patrol then chase the player
    - NavMesh and routing
- An inventory of objects that can be used to open various locked-door puzzles
    - HUD widgets

Throughout, think about how the various elements of these and previous exercises might be sensibly combined into a more complicated game. You can keep track of time, so you can require the player to do various things against the clock. This resource mechanic could easily be translated to a notion of health. You can shoot at targets, so AI enemies can also be shot at, and events triggered when all of them are destroyed. Think about how each challenge might be made harder, or easier, and therefore enable a suitable progression dynamic.



This week's game is going to have the following features:

- A first-person character as in the previous game
- The ability to pick-up, carry, drop and throw objects using the mouse
- Four simple physics based puzzles that the player must solve
    - Stacking boxes to climb over a wall

- Placing a box on a seesaw that can then be climbed
- Using a box to stop a spinning blade
- Placing a box on a button to open a door

This exercise will involve extending the character to create a "gravity gun" style effect for manipulating physical objects, and experimenting with physics constraints to create various interactive features.

This game will again involve a first-person perspective – i.e. a moveable character with mouse look. You can either re-use the simple first-person controller that you made in the previous exercise, or use a ready-made one.

The Unity *Standard Assets* contain a variety of reusable components; scenes, scripts, materials etc., including a first-person character, but it is quite large:

https://assetstore.unity.com/packages/essentials/asset-packs/standard-assets-32351

The two first-person characters on their own have been added to the asset package on Moodle, and can be imported into the project via Assets->Import Package->Custom Package. Once imported, drag either the FPSController or RigidBodyFPSController into the hierarchy from the Standard Assets->Characters->FirestPersonCharacter->Prefabs folder

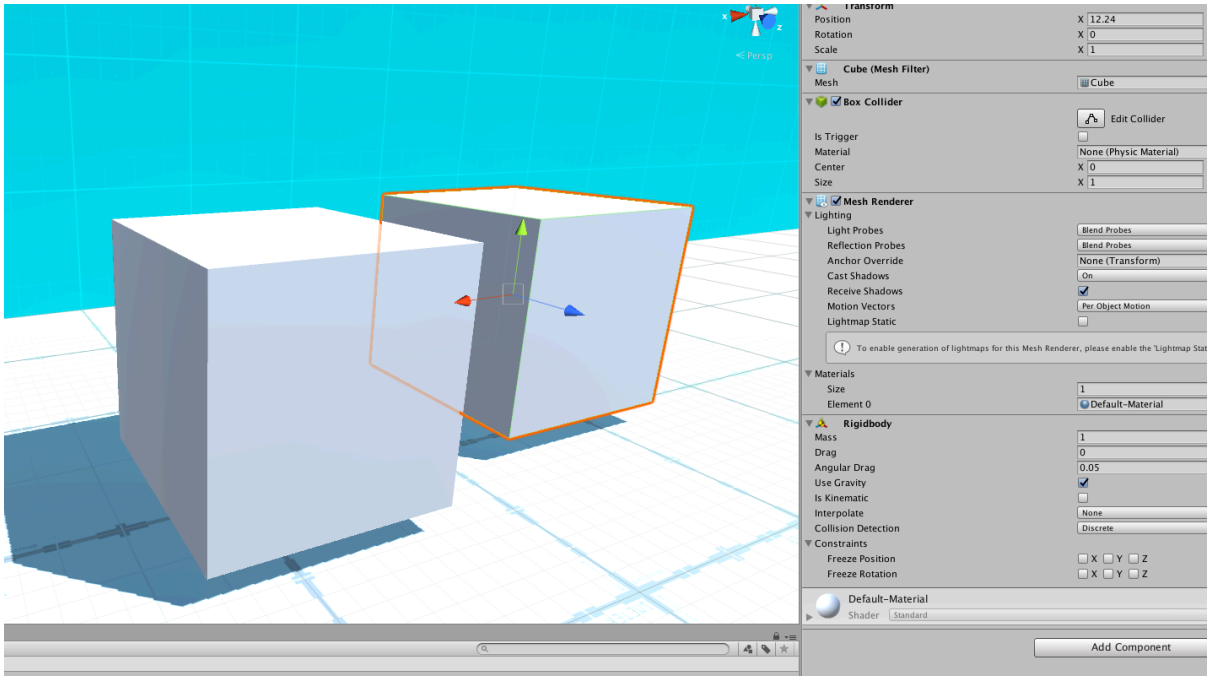
Gravity Gun

The "gravity gun" will allow the player to pick up, carry, drop and throw physics objects within the scene, and this will provide the basic mechanic for subsequent puzzles.

The gravity gun should have the following behaviour:

- When the player clicks the left mouse button, attempt to pick-up the nearest *physics* object that they are looking straight at.
- While the left mouse button is held, move the physics object with the player
- When the left mouse button is released, drop the physics object
- If the right mouse button is pressed while the player is holding a physics object, throw it forwards instead of dropping it.

Create a basic scene, either with a single plane or by using ProBuilder as before. Add some Cubes or other shapes to the scene (GameObject->3D Object->Cube) and make them moveable by adding a *Rigidbody* component to each one. You should be able to push them around using the player's character.



Add a new script to the first-person character – as with the previous exercise we want the player to be able to look straight at the object they wish to pick up, so either attach it to the MainCamera or to the FirstPersonCharacter component of the FPSController.

Modify the FixedUpdate() method for this script:

```
if (Input.GetButtonDown("Fire1"))
{
    if (heldObject == null)
    {
        RaycastHit colliderHit;

        if (Physics.Raycast(transform.position,
                transform.forward,
                out colliderHit,
                10.0f,
                layerMask))
        {
            heldObject = colliderHit.collider.gameObject;
            heldObject.GetComponent<Rigidbody>().useGravity = false;
        }
    }
}

if (heldObject != null)
{
    // move the thing we're holding
    heldObject.GetComponent<Rigidbody>().MovePosition(holdPosition.position);
    heldObject.GetComponent<Rigidbody>().MoveRotation(holdPosition.rotation);
}
```

As with testing for the ground in lab 02, when the player presses fire (by default the left mouse button), and if nothing is currently being held, we *raycast* forwards from the camera. If the ray hits an object within 10 units on the appropriate layer, then the Collider component of the object is returned via the variable *colliderHit*.

- Note that the *out* keyword in C# allows us to pass an argument by reference, meaning that the variable *colliderHit* can be modified by the function, not just used by it.

https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/out

If the raycast is successful, we keep a reference to the GameObject that the Collider component is attached to, so we can manipulate the object.

- Note that the *layerMask* is, as before, a left shifted integer that specifies layer 8:

```
int layerMask = 1 << 8;
```

- However you may find it easier to use the more user-friendly *LayerMask*, that provides an interface in the inspector for selecting multiple layers.

```
public LayerMask layerMask;
```

- Either way, the Cube objects need to be added to the layer in question, as in lab 02. The reason for this should be clear – we want to limit the objects that the user can interact with, as otherwise they would be able to pick up everything, including the floor.

Once an object is "held" – i.e. the GameObject heldObject is not null, each update we want to move the object to where the player has moved, essentially dragging the object around. Here *holdPosition* is, like *shotTransform* in the previous exercises, a *Transform* variable that is set to be a child GameObject somewhere just in front of the player's viewpoint.

https://docs.unity3d.com/ScriptReference/Rigidbody.MovePosition.html

As we want to move the object using physics – to achieve a relatively naturalistic behaviour – we don't just set the transform of the object being carried (teleport it to where it should be) but instead *move* the object smoothly by manipulating its physical manifestation, the *Rigidbody* component.

- Note that when the object is "held" we also disable gravity on the *Rigidbody*, to stop it falling downwards.

```
if (Input.GetButtonUp("Fire1"))
{
    // drop the object again
    if (heldObject!=null)
    {
        heldObject.GetComponent<Rigidbody>().useGravity = true;
        heldObject = null;
    }
}
```

When the player released the mouse button, and if they were holding something, we set our reference to it to null – i.e. we no longer want to move that object – and re-enable its gravity.

You might find that the object carries some momentum from being moved causing it to jump rather than fall. If you want to remove this behaviour then when dropping the object zero its linear and angular velocity.

```
heldObject.GetComponent<Rigidbody>().velocity = Vector3.zero;
heldObject.GetComponent<Rigidbody>().angularVelocity = Vector3.zero;
heldObject.GetComponent<Rigidbody>().ResetInertiaTensor();
```

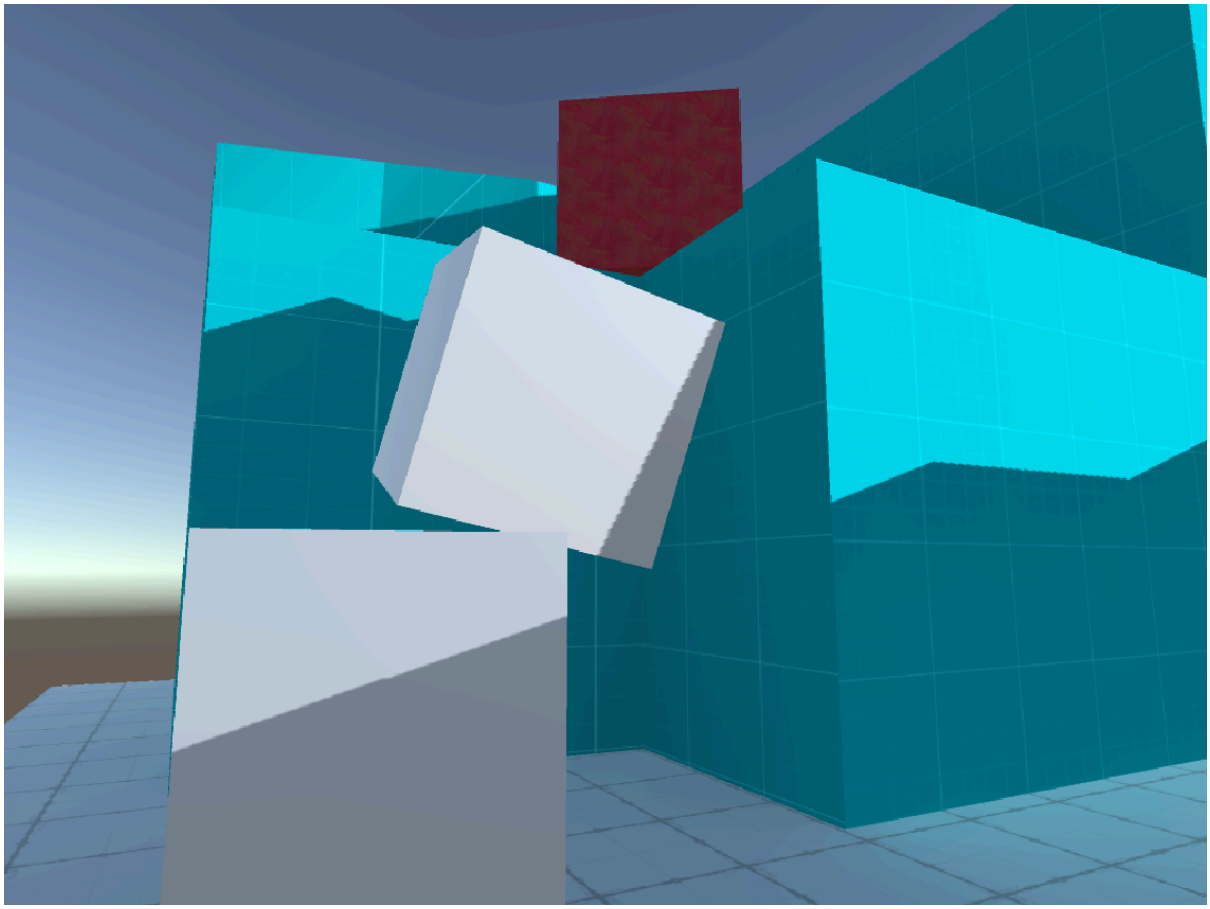
Throwing the object is much the same as dropping it. When the right mouse button (by default this is bound to “Fire2”) is pressed, enable gravity on the object again but also apply an *impulse force* – like a bat hitting a ball – to the Rigidbody component, where the impulse to be applied is a vector calculated by multiplying the *forward* vector of the camera (where we are looking) by some constant multiplyer. This means throw the object with a force of 10.0f in the forward direction.

```
AddForce(transform.forward * 10.0f, ForceMode.Impulse);
```

https://docs.unity3d.com/ScriptReference/Rigidbody.AddForce.html

A Climbing Puzzle

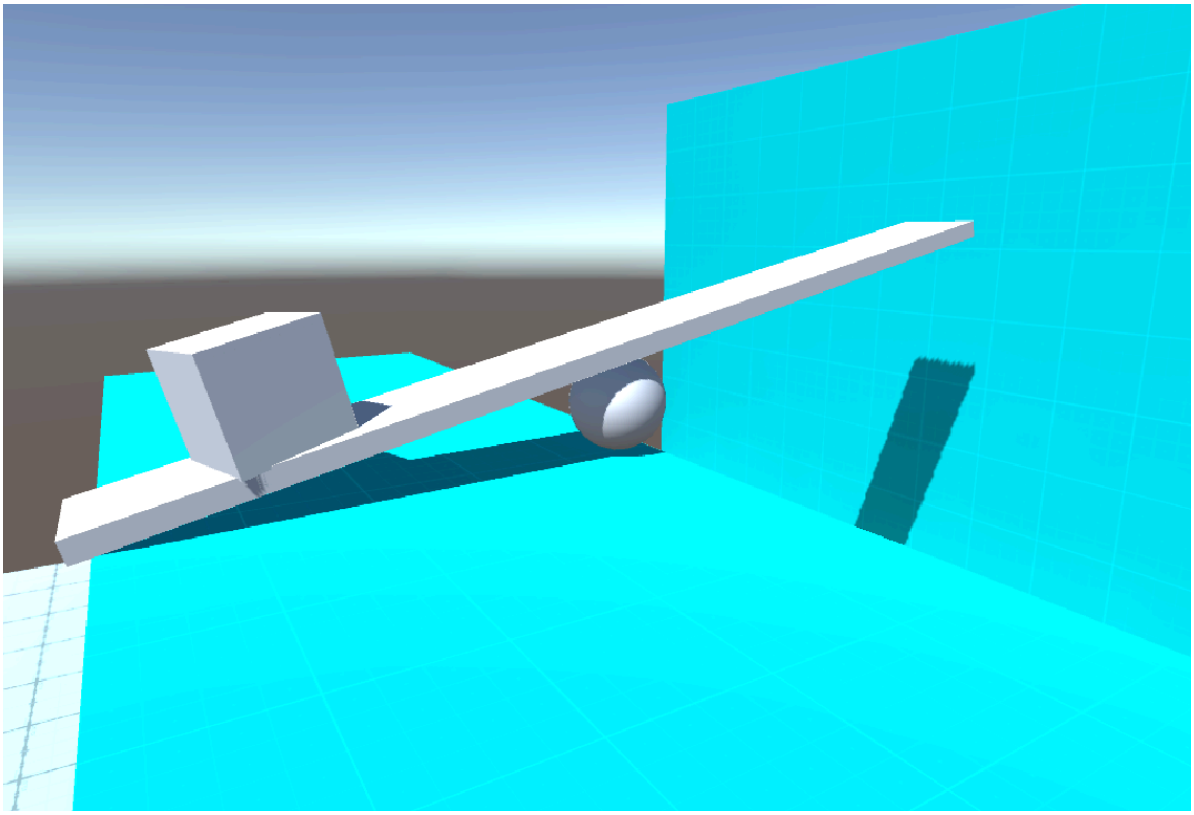
The beauty of the gravity gun / the manipulation of physics objects as a mechanic is that the physics engine can be exploited to naturalistically drive puzzles - stacking, throwing and placing things can be used to overcome obstacles as in “the real world”, as opposed to scripting many specific interactions, and this can engender a sense of exploration and lateral thinking for the player.



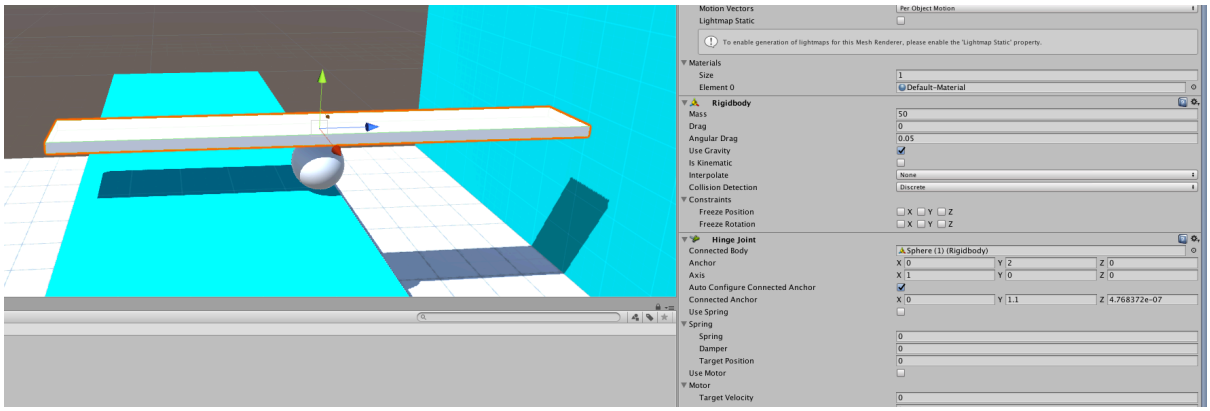
Having some sense of how high the player character can jump, construct a terrain geometry that serves as a simple climbing puzzle, and requires the player to use a moveable cube object to stand on to get to the next step, and then either must reuse the cube to climb up again, or find a second cube.

Seesaw

Next implement a simple physics puzzle in which there is a high wall blocking the progress of the player again. The player will need to find a heavy object to place on one end of a balancing seesaw, in order to raise it high enough that it can be used to jump on to the top of the wall.



This puzzle does not require scripting, but will make use of **Physics Constraints**, which allow constraining the movement of physical components within a given actor based on physical parameters.

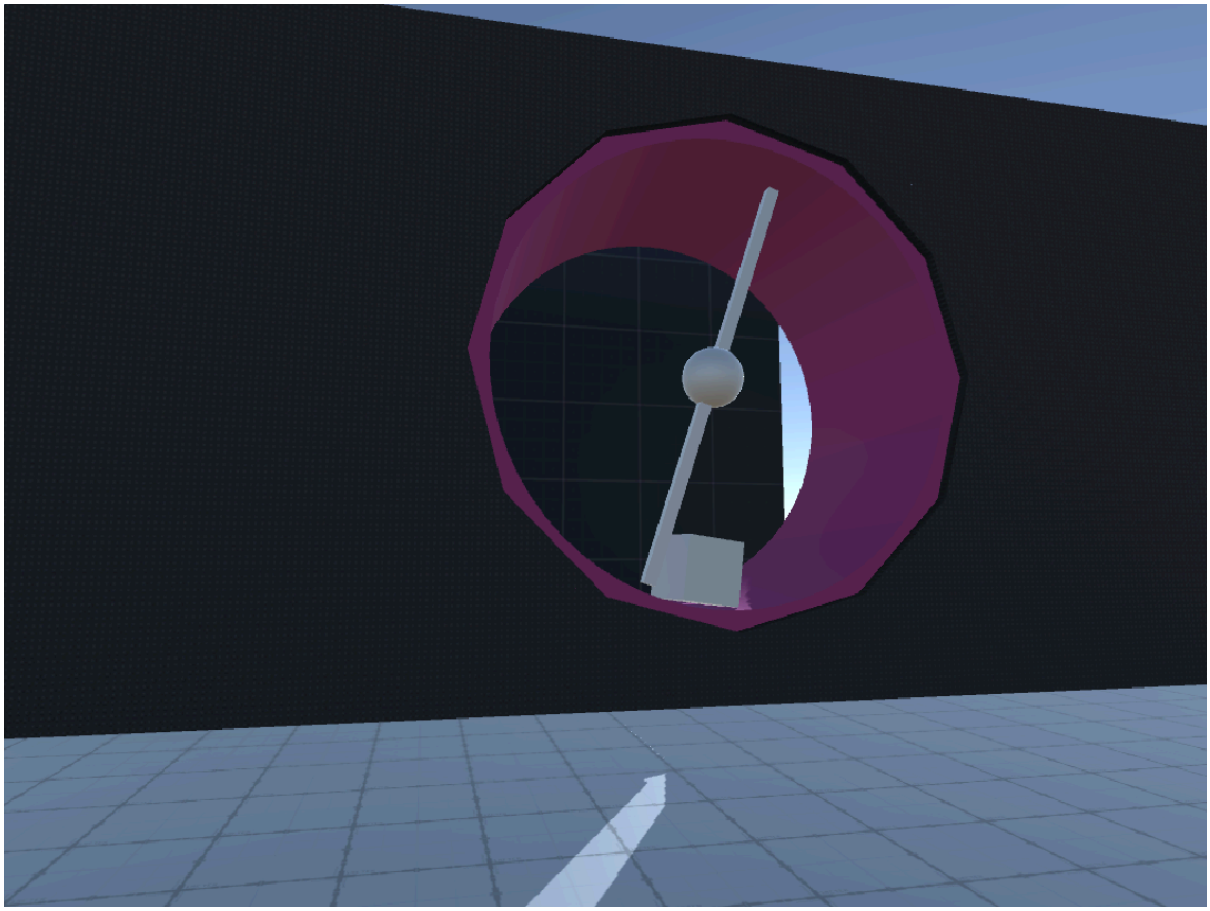


Create a Cube game object, again with a *Rigidbody* component, but scale it to be a thin and long plank, as in the screenshot, and place it on top of a sphere with a slight gap – the sphere provides the visual appearance of being the pivot of the seesaw.

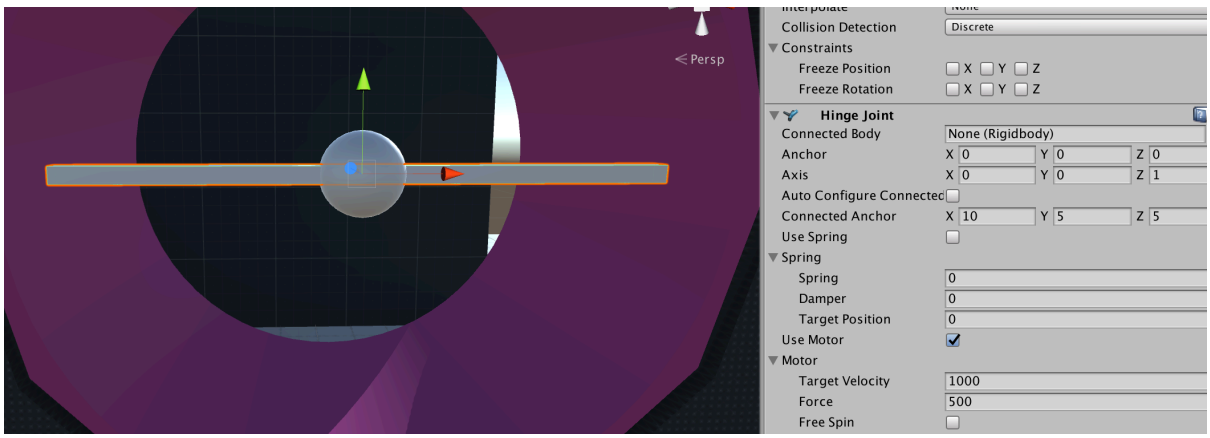
To make the seesaw work, add a *Hinge Joint* component to the flattened cube. This is a multipurpose joint that we can use to constrain the movement of the cube. The seesaw should only be allowed to rotate about the X axis (the red arrow) and this is achieved by setting the *axis* of the joint to be 1, 0, 0. The joint will connect the object to the world at the specified anchor point, or if a *Connected Body* is selected (i.e. the sphere) at an anchor point relative to this other object.

In the game, pick up and place a cube on one end of the seesaw to check that it works as expected. You may need to change the *mass* of the various Rigidbodies to be relatively similar to avoid a large amount of jitter.

Spinning Blade



The *Hinge Joint* can easily be adapted into a different kind of puzzle. Create a second jointed setup, but this time enable the *motor* for the joint – this will constantly drive the joint with the specified force to achieve the target velocity. The goal for the player here is to stop the motion of the joint by placing an appropriately heavy object in the way of the spinning fan blade so they can get through without being harmed.



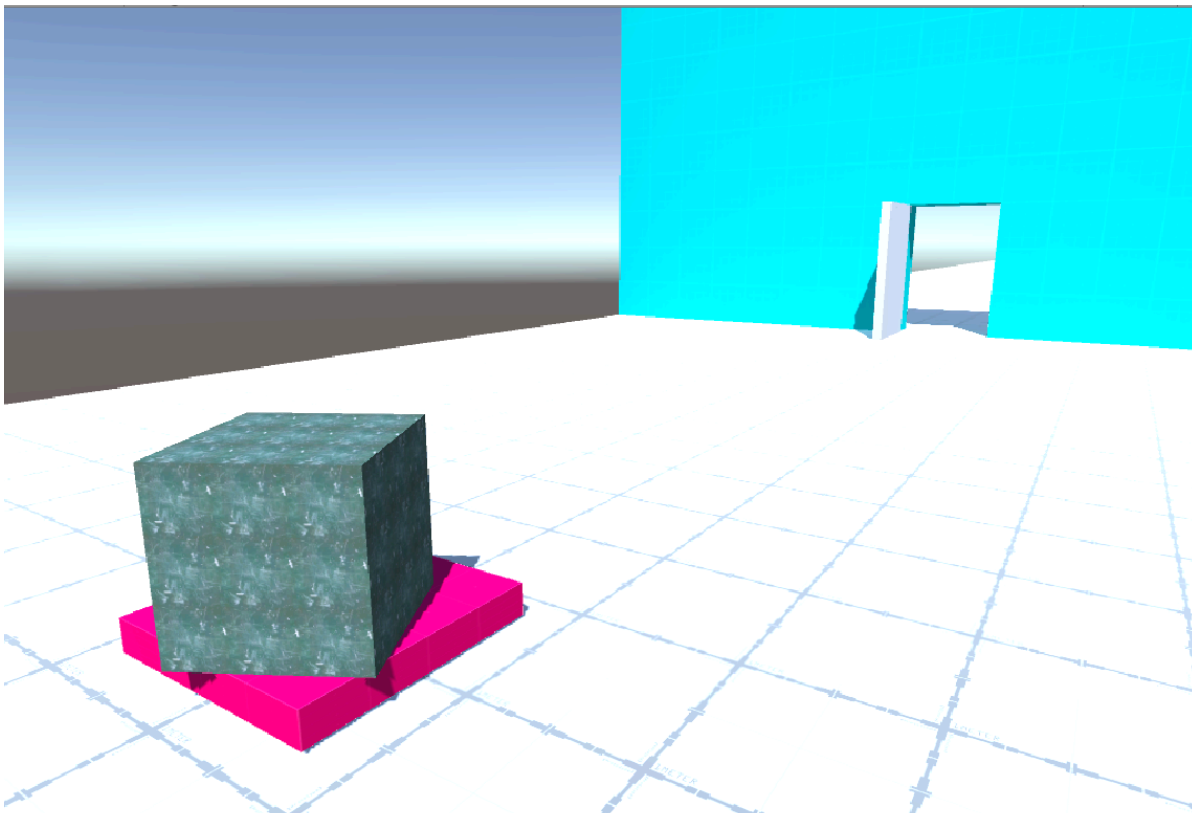
https://docs.unity3d.com/Manual/class-HingeJoint.html

You could easily add *wind* to the fan, by adding a *Trigger* collider that for each frame that an object remained in it resulted in force being applied to the object in a certain direction:

```
private void OnTriggerStay(Collider other)
{
    Rigidbody b = other.gameObject.GetComponent<Rigidbody>();
    b.AddForce(new Vector3(0, 50, 0), ForceMode.Force);
}
```

You might also experiment with the various *breaking* parameters of the joint. This will separate the object from its joint when a certain level of force is achieved as applied by the object that you are carrying.

Door Puzzle



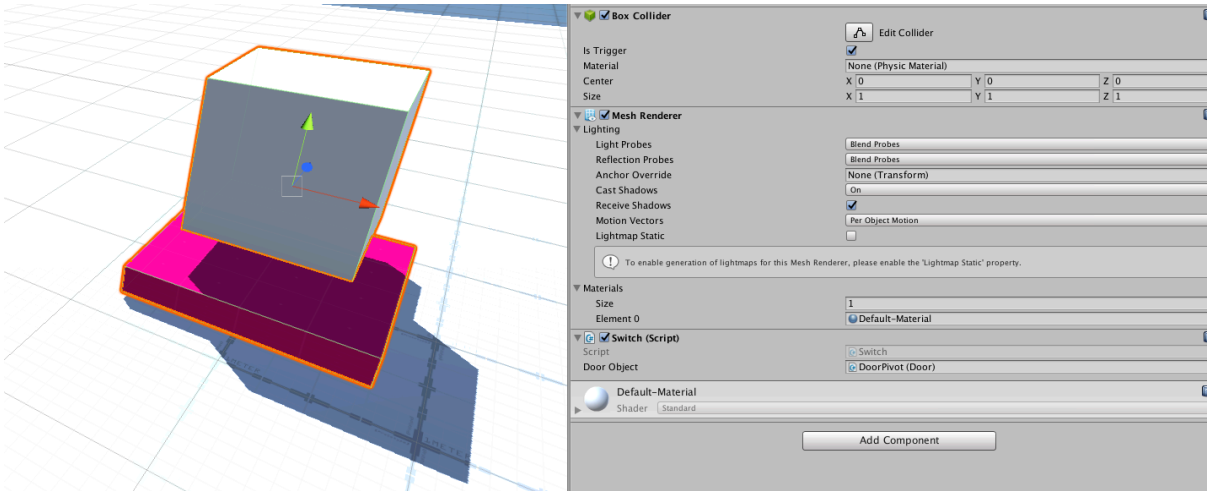
The final puzzle that the gravity gun will be used for is an inference challenge around a locked door. There is a *button* on the floor, which, when the player stands on it, opens a door. When the player leaves the button, the door closes again. The button is too far away from the door for the player to be able to run through it while it is opening. The challenge for the player is simply to find an object to place on the button that will hold the door open so that they can move through it.

There are three parts to creating this:

- Creating a button object that detects when something is placed upon it

- Creating a door that can open and close
- Connect the two together

Create a new Cube game object. The Box collider component of the cube will serve as the switch – when something enters the trigger, we will tell a door to open, and when the same thing leaves the trigger, we will tell the door to close.



Set the Box Collider to be a *Trigger*, and then disable the *Mesh Renderer* to render the collider invisible. You might find it useful to add a child cube to serve as the visual "switch".

```
public Door doorObject;

private void OnTriggerEnter(Collider other)
{
   doorObject.Open();
}

private void OnTriggerExit(Collider other)
{
    doorObject.Close();
}
```

Here *doorObject* refers to another game object, specified by a public variable, that serves as the door and has a script named *Door* attached to it with the appropriate functions:

```
public void Open()
{
    Debug.Log("opening");
}

public void Close()
{
    Debug.Log("closing");
}
```

How you choose to actually then open and close the door based on this is up to you – it could make use of an Animation, or an AnimationCurve, or you could start a *Coroutine* (remember lab 01) to change the rotation of the door object over time, as below, where *targetAngle* is the angle in degrees to aim for, and the animation speed is the number of frames this should take to happen:

```
private IEnumerator DoorAnimation(int targetAngle, int animationSpeed)
{
    for (int r = 0; r < animationSpeed; r += 1)
    {
        transform.localEulerAngles = new Vector3(0,
            Mathf.LerpAngle(transform.localEulerAngles.y,targetAngle,
                5f / animationSpeed),
            0);
        yield return null;
    }
}
```

- Note that this will cause the door to rotate around its centre Y axis, rather than if it were hinged on one side. You can change this behaviour by introducing an invisible parent object to act as the pivot, and it is this object that is rotated, allowing you to offset the transform of the visible child door.

Exercises

Meta-challenges

These four puzzles can be put together into a single level, with each presenting a subsequent challenge for the player.

Consider how the puzzles might make different use of the properties of the object being carried. For example, can you make one of the available objects in the seesaw puzzle *look* heavier than the rest, implying that this is the best one to use.

Consider how the four puzzles should be arranged. A common strategy is to present more than one puzzle that can be tackled at once, but only one correct approach. For example, require the player to first solve the locked door puzzle, and that unlocks the objects that can be used to climb the wall.