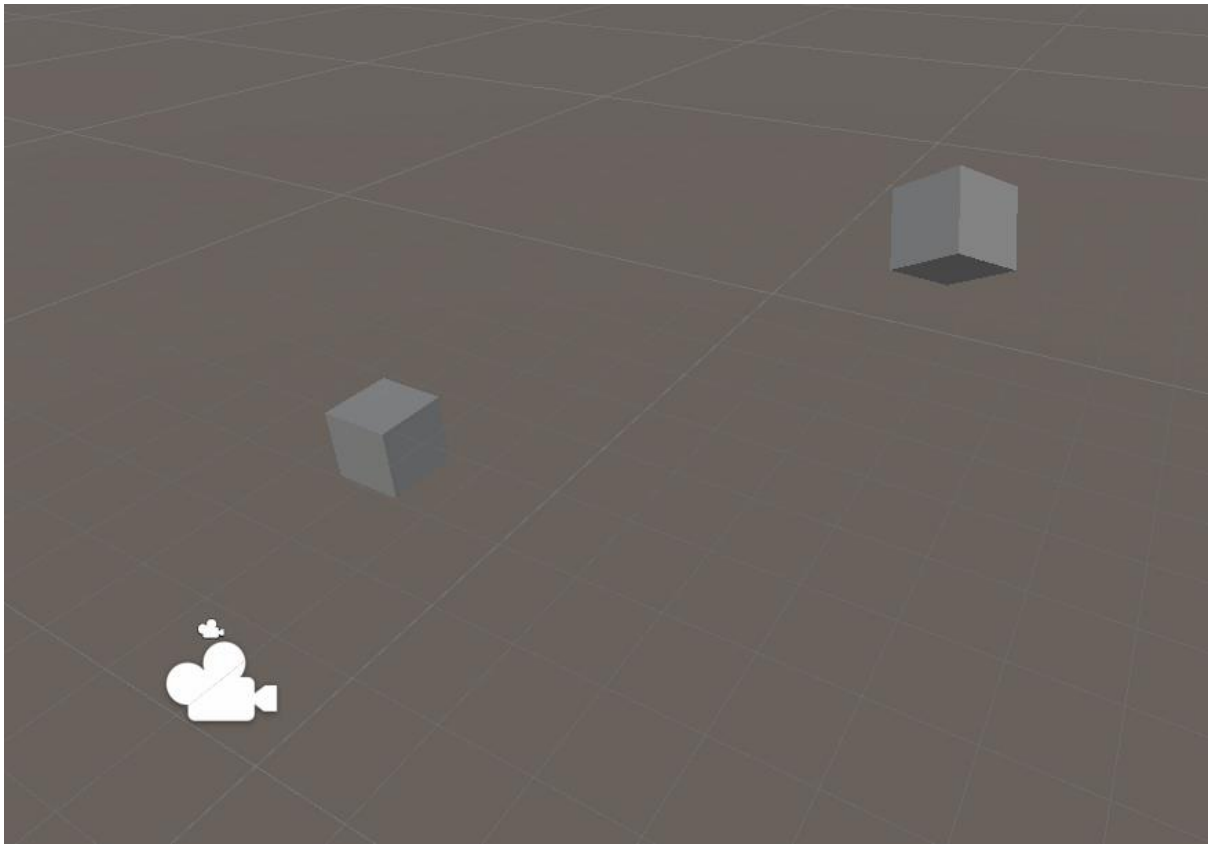
Lerp functions

In this tutorial we will learn how to use linear interpolation properly in order to smoothly change values, such as positions or rotations to give the movement in our games a more polished feel.

First, create two cubes in a 3D scene, and space them apart. Rotate one of the cubes so that it is oriented differently to the other.



Next, insert a script in the object you wish to move, for this example, we will move the first cube to the second rotated one.

There are two main ways of smoothly moving objects with lerp, one may be considered more desirable, but for the purpose of this tutorial I will show both ways.

Method 1

Start by creating a public transform to reference the lerp destination:

```
public Transform destination;
```

Next, create a function and call it “Lerp” as so:

```
0 references
void Lerp()
{

}
```

Inside this function is where our movement will take place. Since we are essentially changing the position based on distance to the destination and the time elapsed, we will be modifying transform.Position. Here we can use Vector3.Lerp as so:

```
0 references
void Lerp()
{
    transform.position = Vector3.Lerp(transform.position, destination.position, Time.deltaTime);
}
```

Vector3.Lerp takes three values: start value, end value and the fraction to move by between 0 and 1, 0 being not at all and 1 being all the way. For now, we will use Time.deltaTime, which is equivalent to the amount of time since the last frame was rendered, equivalent to approximately 0.016 seconds.

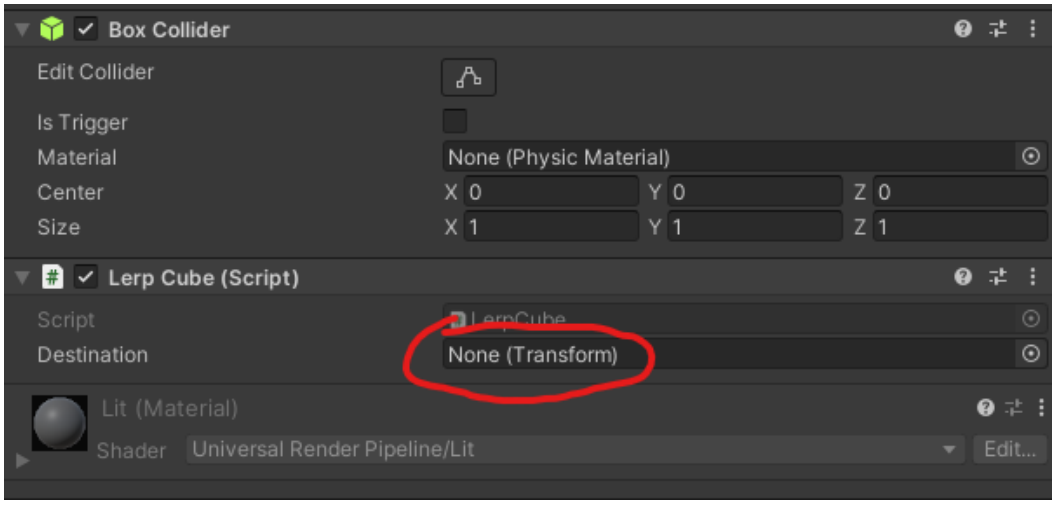
Because we want this to run smoothly, we will call the Lerp function every frame, in void Update.

```
public Transform destination;

Unity Message | 0 references
void Update()
{
    Lerp();
}

1 reference
void Lerp()
{
    transform.position = Vector3.Lerp(transform.position, destination.position, Time.deltaTime);
}
```

Finally, we need to set our reference to the target destination from earlier. Look for the script in the inspector to find the empty field, and drag the gameobject you want to move to.



Upon pressing play, you will notice that the cube smoothly moves to the destination position, but does not rotate to match it. You may also notice that it is gradually slowing down, and if you select the moving cube you will find that it never actually reaches the target, slowing exponentially! If this is not your desired lerp, I will show off a better way.

Method 2

Firstly, we will need to make sure that our lerp function receives only one start and end point, rather than the using the current position which is constantly changing. We will set these up in a Start function:

```
Vector3 startPos;
Vector3 endPos;

Unity Message | 0 references
private void Start()
{
    startPos = transform.position;
    endPos = destination.position;
}
```

This will run only once at the start, so the values we give our lerp will not change, regardless of if the gameobjects are. Next, we need to change our lerp function to incorporate this.

```
transform.position = Vector3.Lerp(startPos, endPos, Time.deltaTime);
```

This won't work on its own, as with the start position now locked, our gameobject will only travel by a fraction of Time.deltaTime toward the target. To fix this, we will need to create a new value for the fraction we want to move each frame. Start by creating a public float for the time we want it to take, and a private float for the elapsed time:

```
public float duration = 3;
float elapsedTime = 0;
```

We will use the private float to count the time since we started lerping like so:

```
void Lerp()
{
    elapsedTime += Time.deltaTime;
```

We then use this value to calculate the fraction of time we have remaining in the lerp

```
float fraction = elapsedTime / duration;
```

Finally, we will use a second interpolation, known as SmoothStep. If you recall earlier where lerps take three values, we want to lerp the third value or the fraction of the distance by which we want to lerp, which is a value between 0 and 1. We couple this with our fraction of the time that has elapsed so far and we get this for our final line of code.

```
1 reference
void Lerp()
{
    elapsedTime += Time.deltaTime;
    float fraction = elapsedTime / duration;
    transform.position = Vector3.Lerp(startPos, endPos, Mathf.SmoothStep(0, 1, fraction));
}
```

Enter play mode, and you should see that the gameobject smoothly moves to its destination, speeding up and slowing down in a predictable and tidy way.

If you want to lerp rotation too, we need to add some new variables to account for start and end:

```
Quaternion startRotation;
Quaternion endRotation;
```

```
Unity Message | 0 references
private void Start()
{
    startPos = transform.position;
    endPos = destination.position;

    startRotation = transform.rotation;
    endRotation = destination.rotation;
}
```

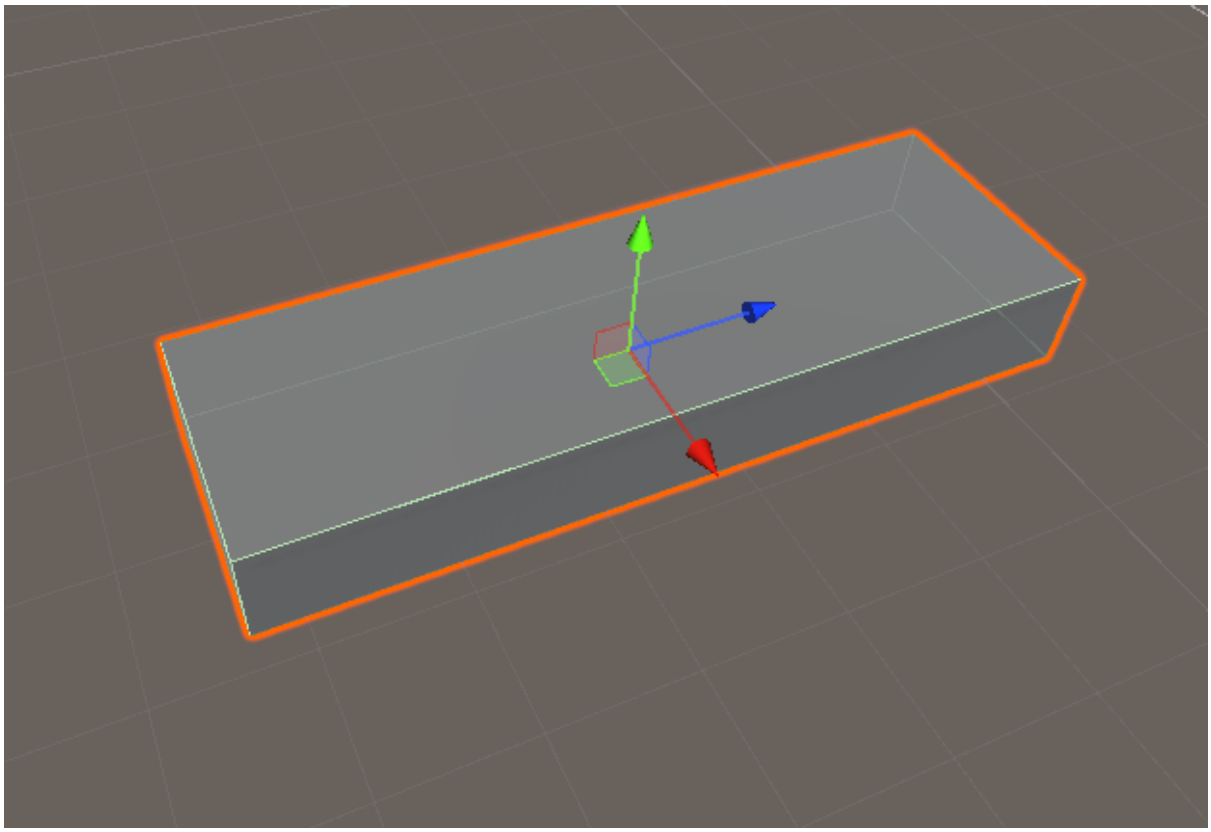
We can use the function transform.SetPositionAndRotation(), and simply copy our earlier lerp code to a Quaternion.Lerp and replace the positions with our rotations.

```
transform.SetPositionAndRotation(Vector3.Lerp(startPos, endPos, Mathf.SmoothStep(0, 1, fraction)), Quaternion.Lerp(startRotation, endRotation, Mathf.SmoothStep(0, 1, fraction)));
```

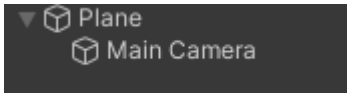
Flight controls

In this tutorial I will show you how to make a simple physics-based plane, using a simplified classic aeroplane pitch/roll flight model.

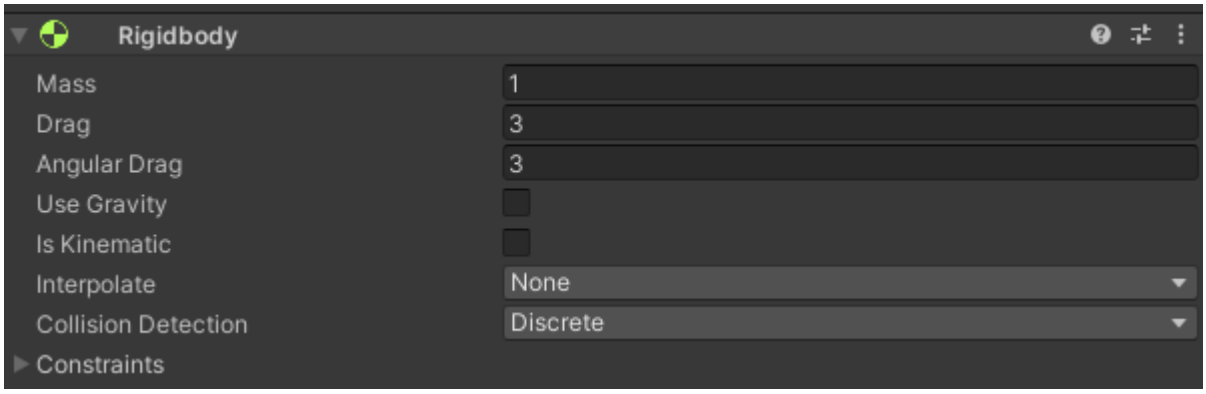
Firstly, create an object to be the body of the plane. In this example I will use an elongated cube. Ensure that when clicking on the object in local transform mode, the blue arrow points in the plane's forward direction.



Next drag the camera inside of the object in the hierarchy, and position it behind and slightly above it.



This will lock the camera to the plane to allow us to see where we are going as we fly. Since the plane will be physics based, attach a rigidbody to the gameobject, and set a drag and angular drag value as shown (you may tweak these to your liking later) and ensure "Use Gravity" is unchecked.



Next, create a script and add it into the gameobject. We will start by setting up all of the parameters for the controls and speeds.

```
public float maxSpeed = 15;
public float acceleration = 10;

public float pitchSpeed = 3;
public float rollSpeed = 1;

private float speed;
```

These public variables will allow easy adjustments to the plane's controls on the fly (ha), but for now I have selected some sensible default values. To save time, make a shorthand reference to the gameobject's rigidbody, since we will be referencing it a lot.

```
private Rigidbody rb;

Unity Message | 0 references
private void Start()
{
    rb = GetComponent<Rigidbody>();
}
```

Since we are working with unity physics, the plane's movement code will go inside a FixedUpdate rather than an Update. While Update runs every frame, it is best not to use it for physics calculations, since physics runs at a different rate to rendering, and using update may cause some stuttering. Inside FixedUpdate, create an if statement to check for one of two keypresses, and add to the speed value.

```
void FixedUpdate()
{
    if (Input.GetKey(KeyCode.E))
    {
        speed += acceleration * Time.fixedDeltaTime;
    }
    else if (Input.GetKey(KeyCode.Q))
    {
        speed -= acceleration * Time.fixedDeltaTime;
    }
```

Here I am using Q and E to accelerate and decelerate. By adding (or subtracting) our acceleration every physics update (using Time.fixedDeltaTime to get the time since the last physics update), we are able to customise the rate at which we speed up or slow down the plane.

To apply this speed to the plane, simply write the following line below the if statements:

```
rb.AddForce(speed * transform.forward);
```

You may find this allows the plane to accelerate forever, and even fly in reverse. Since we don't want this, a simple way to stop both of these issues using MathF.Clamp(). Put the following line of code just before when the force is added:

```
speed = Mathf.Clamp(speed, 0, maxSpeed);
```

Clamp takes three parameters, a variable and two floats, which then locks said variable's value to within the minimum and maximum of said floats. This will prevent the plane exceeding maxSpeed and also going backwards.

Next, we need to implement pitch controls. The easiest way to do this is by using axis controls, as this condenses multiple inputs on one axis to one value that can be positive or negative.

```
float pitch = Input.GetAxis("Vertical");
```

This will read W and S (or up and down on a joystick) as 1 and -1 respectively. Now we just need to apply the force to the rigidbody using RigidBody.AddTorque.

```
rb.AddTorque(transform.right * pitch * pitchSpeed);
```

Finally, for roll, we can re-use our code, but change the axis to horizontal, and use transform.forward as our pivot axis and -roll (so the controls aren't inverted)

```
Unity Message | 0 references
void FixedUpdate()
{
    if (Input.GetKey(KeyCode.E))
    {
        speed += acceleration * Time.fixedDeltaTime;
    }
    else if (Input.GetKey(KeyCode.Q))
    {
        speed -= acceleration * Time.fixedDeltaTime;
    }

    float pitch = Input.GetAxis("Vertical");
    float roll = Input.GetAxis("Horizontal");

    rb.AddTorque(transform.right * pitch * pitchSpeed);
    rb.AddTorque(transform.forward * -roll * rollSpeed);

    speed = Mathf.Clamp(speed, 0, maxSpeed);

    rb.AddForce(speed * transform.forward);
}
```

We can add a simple collision detection function to add challenge to our game, using OnCollisionEnter() to trigger a function to cause an end condition such as a game over script, or simply delete the plane.

```
Unity Message | 0 references
private void OnCollisionEnter(Collision collision)
{
    destroyPlane();
}

1 reference
void destroyPlane()
{
    Destroy(gameObject);
}
```

Guided missiles

This tutorial will teach you how to create a deadly guided projectile that can be used in conjunction with the plane from the previous tutorial to create a fun challenge, combining knowledge of the previous two tutorials.

This example won't include target selection, so for demonstration we will just drag the object we wish to track into a public variable. Add some other variables for tweaking the missile's turn and fly speeds.

```
public Transform target;
public float turnSpeed = 1;
public float flySpeed = 2;
```

We can make the missile track towards the player with Quaternion.LookRotation, by giving it a vector direction. Since we don't know the direction to the player, we can find it out by finding the difference between the two vector3 positions.

```
transform.rotation = Quaternion.LookRotation(target.transform.position - transform.position);
```

Next, add code to translate the plane forward locally by flySpeed, not forgetting to multiply by time.deltatime to ensure it is counted in seconds.

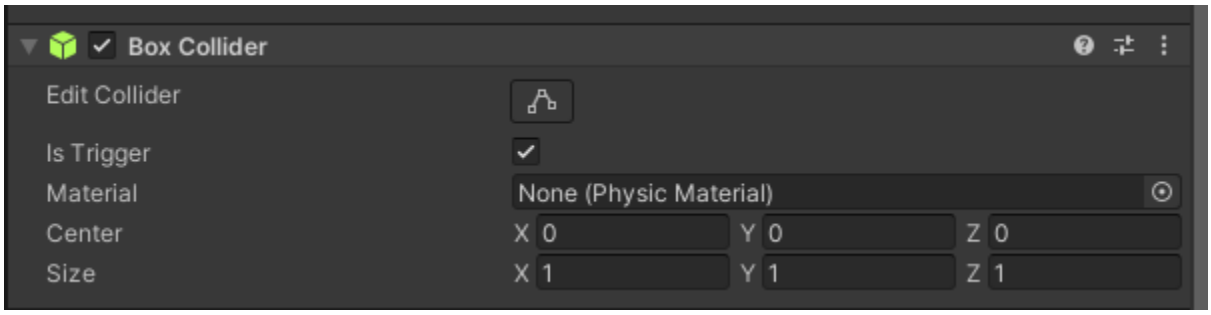
```
transform.Translate(transform.forward * flySpeed * Time.deltaTime);
```

When hitting play, you may notice that the missile is deadly accurate! Since this isn't fun to play against as it is impossible to avoid, we will need to add a delay to its tracking. We can use Quaternion.Lerp from before, giving current rotation, target look rotation and turn speed as its parameters.

```
Unity Message | 0 references
void Update()
{
    targetRotation = Quaternion.LookRotation(target.transform.position - transform.position);

    transform.rotation = Quaternion.Slerp(transform.rotation, targetRotation, turnSpeed * Time.deltaTime);
    transform.Translate(transform.forward * flySpeed * Time.deltaTime);
}
```

Finally, we can trigger behaviours in objects it hit, such as destroying the plane from earlier. For this I will make sure that the collider is set to a trigger, and use SendMessage inside OnTriggerEnter to Invoke the plane's destruction function.



Using an If statement to check that the script is present, we can simply execute the plane's destruction function from inside the missile when it hits.

```
Unity Message | 0 references
private void OnTriggerEnter(Collider other)
{
    if (other.GetComponent<FlyPlane>())
    {
        other.SendMessage("destroyPlane");
    }
}
```