CW1 Tutorials

Tutorial 1: Edge Detection AI

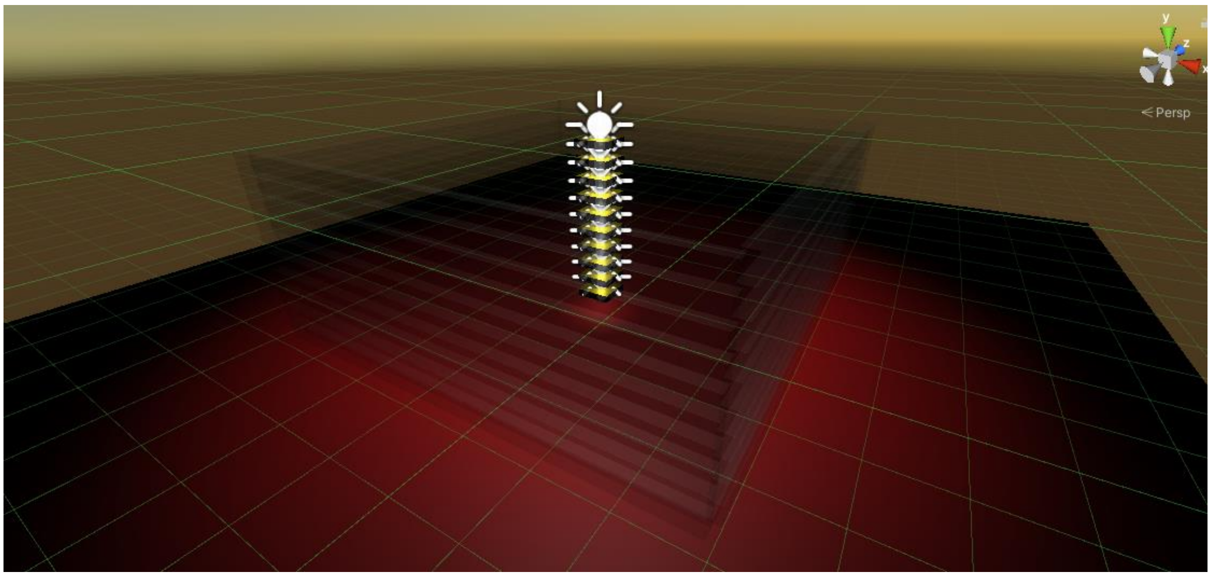
Difficulty: Hard

This tutorial is for AI simple movement, rotation and more importantly giving the AI an ability to detect if it will fall. The edge detection is rather simple, however, if you want to make your own pathing code this may be useful as starting code.

Specific program info:

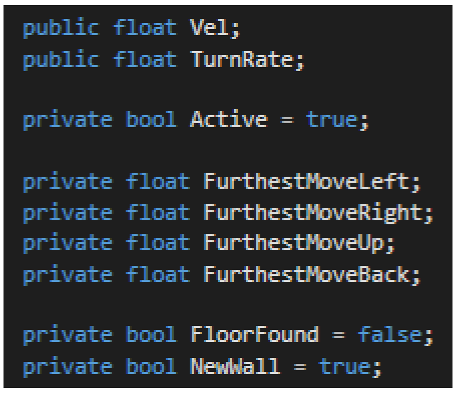
- AI Movement
- Edge detection when reaching a side of the object
- Changing which object to check for fall when landing on a new object

Firstly, we must make the simple AI object. For this we take a cube, add an Rigidbody (RB), and make sure it has the Box Collider. As well as this, create a large flat plane using another box which will be the floor. This large plane will be what the AI uses to move on. This object also needs a Box Collider, but not an RB as it does not contain physics. For this project there is no need to use the RB physics as we are not having any physics as a mechanic. I have attached an RB to the AI object only so that we can test that it will not fall, if it had no RB then there would not be any gravity or mass to make it fall.



You only need 1 movement object and 1 plane, but for me I have stacked many on top to see the effect of the random movement.

After setting up the scene like this, we need to make the simple AI movement. First, we define the "speeds" for movement, make 2 public (this can be private, public only so you can edit in inspector) float variables: 1 for forward velocity and 1 for turn rate. We also need 3 boolean variables, an "Active" variable (to run/cancel the script) a "FoundFloor" variable used to see if the object is touching the floor, and finally a "NewWall" variable used to see if the object is touching a new side of the floor. We also need 4 floats to define the ends of the floor.



```
public float Vel;
public float TurnRate;

private bool Active = true;

private float FurthestMoveLeft;
private float FurthestMoveRight;
private float FurthestMoveUp;
private float FurthestMoveBack;

private bool FloorFound = false;
private bool NewWall = true;
```

Now to make the simple movement. Create 2 private void functions, RotateObject and MoveInDirection. Both functions should inherit a Vector3 called Direction.

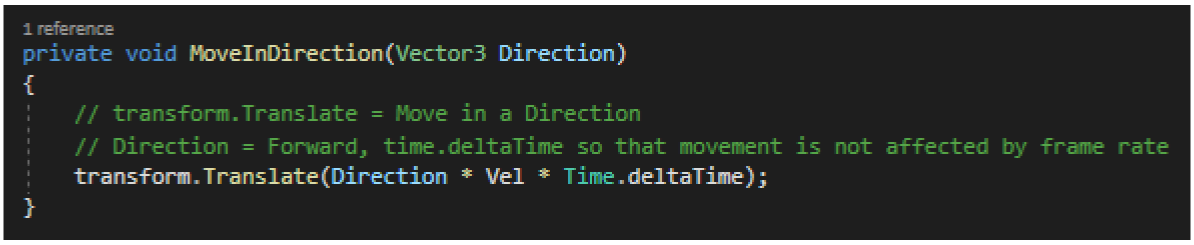
We will code the movement function first. For this project I am using transform movement as it is very precise, although if you wanted you could use RB.AddForce(). Specifically, I used transform.Translate in this function. Translate here means to move in a direction, you -could- use transform.position = new Vector3(…) however this will be highly complicated due to the rotation of the AI object. Inside MoveInDirection, write: transform.Translate(Direction * Vel * Time.deltaTime);

transform = object positioning variables

Direction = Vector3 direction, ie: transform.forward

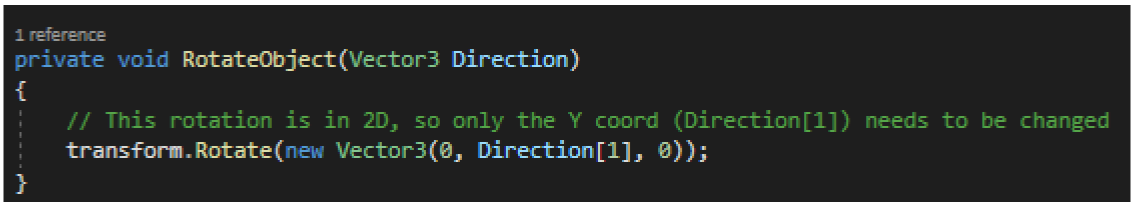
Vel = Speed

Time.deltaTime = makes sure the movement is not affected by frame rate



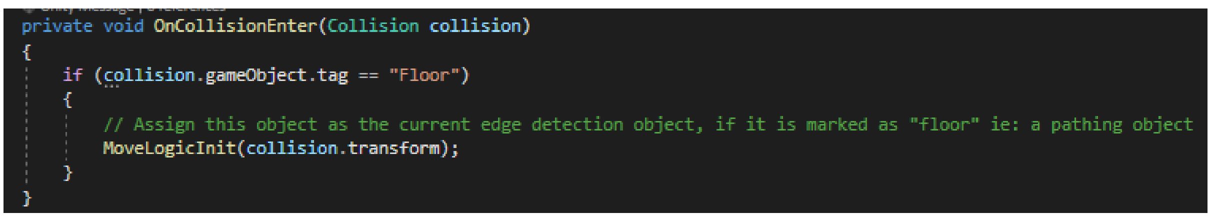
```
1 reference
private void MoveInDirection(Vector3 Direction)
{
    // transform.Translate = Move in a Direction
    // Direction = Forward, time.deltaTime so that movement is not affected by frame rate
    transform.Translate(Direction * Vel * Time.deltaTime);
}
```

Now to code the RotateObject. This is much more simple, I used the simple built in rotation code for Unity, transform.Rotate(). We only need to change rotation in 1 axis, the y axis for this tutorial as we are only doing edge detection on a flat surface. Simply write the code: transform.Rotate(new Vector3(0, Direction[1], 0));



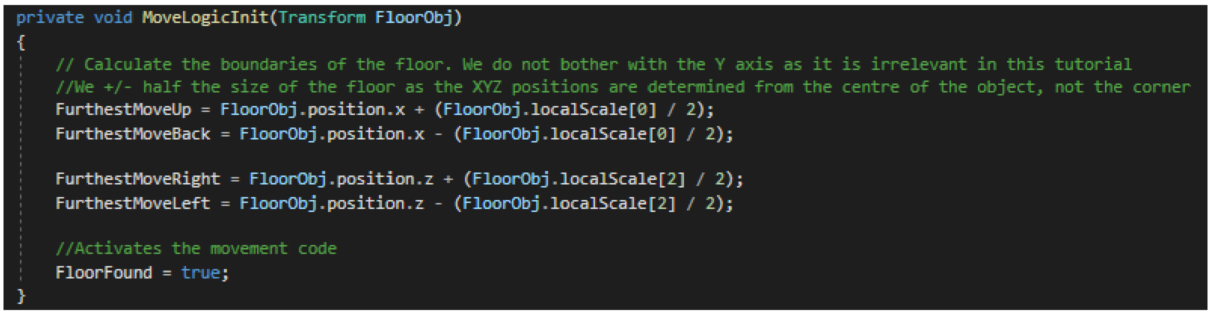
```
1 reference
private void RotateObject(Vector3 Direction)
{
    // This rotation is in 2D, so only the Y coord (Direction[1]) needs to be changed
    transform.Rotate(new Vector3(0, Direction[1], 0));
}
```

Detection of if the AI is touching the floor is rather simple, use the OnCollisionEnter function and check that the collided object has the “floor” tag, this marks it as a valid pathing object. After colliding with a floor, the code calls the MoveLogicInit function. This immediately calculates and stores the boundaries of the object with 4 variables to determine where not to move the AI past.



```
private void OnCollisionEnter(Collision collision)
{
    if (collision.gameObject.tag == "Floor")
    {
        // Assign this object as the current edge detection object, if it is marked as "floor" ie: a pathing object
        MoveLogicInit(collision.transform);
    }
}
```

The move logic function simply decides which direction to turn the layer and decides if the players current rotation is invalid and resets it (> 356 or < 0). The in-code comments are more useful to explain this part of the code.

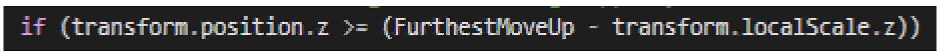


```
private void MoveLogicInit(Transform FloorObj)
{
    // Calculate the boundaries of the floor. We do not bother with the Y axis as it is irrelevant in this tutorial
    //We +/- half the size of the floor as the XYZ positions are determined from the centre of the object, not the corner
    FurthestMoveUp = FloorObj.position.x + (FloorObj.localScale[0] / 2);
    FurthestMoveBack = FloorObj.position.x - (FloorObj.localScale[0] / 2);

    FurthestMoveRight = FloorObj.position.z + (FloorObj.localScale[2] / 2);
    FurthestMoveLeft = FloorObj.position.z - (FloorObj.localScale[2] / 2);

    //Activates the movement code
    FloorFound = true;
}
```

This function assigns the values of the Furthest Move variables used to find the edges of the floor and prevent the AI falling off. Also, now that the floor is found movement can begin, this is done simply by enabling the movement function by setting “FloorFound” to true.

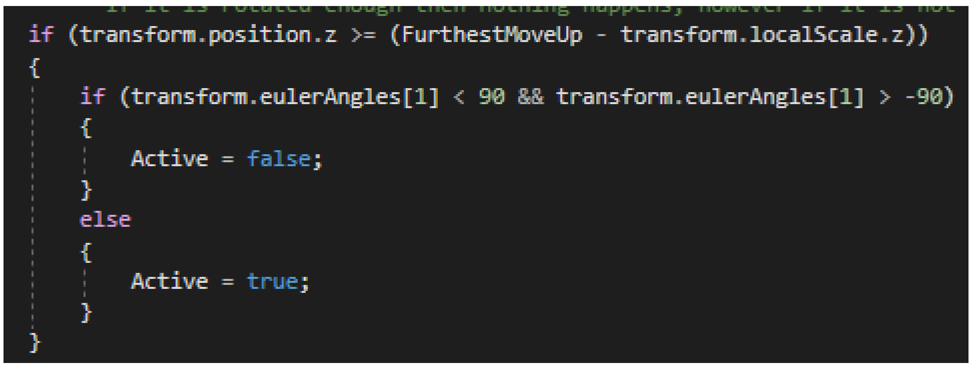
Now that we’ve found the boundaries, we should create the function used to check the object’s position compared to the floor. Create a function called “CheckBoundaries”. In this function we will simply compare the player’s position to the boundary, and then check if the player’s rotation will make them fall off the boundary. I will explain one of these if statements as all 4 directions are the same code with different numbers.

First we check if the player’s position is greater than the boundary of the floor (if moving up or right, otherwise we check to see if the position is less than as moving up and down is increasing the value of the coordinate, whilst moving left and down is decreasing it) while taking into account the size of the player object. To do this we do:



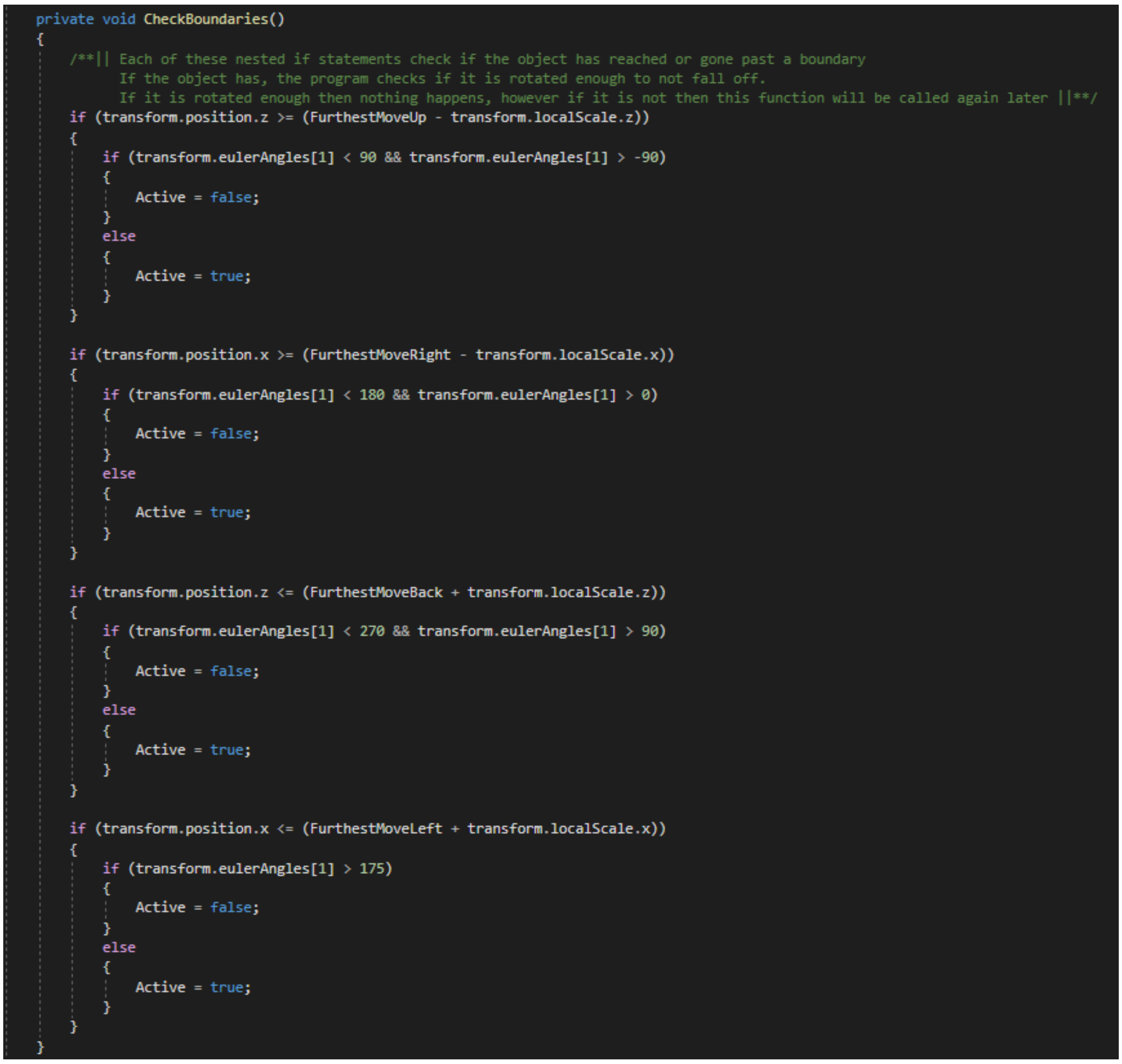
```
if (transform.position.z >= (FurthestMoveUp - transform.localScale.z))
```

If this is true, we know the AI is at a boundary, and if it continues moving it might fall off. We must next check if the rotation of the object will lead it to fall if it continues to move. If so then we must stop the object, if false we can reenable movement. For this we nest an if statement inside this previous statement comparing the player rotation (eulerAngles to use inspector values) to the min and max required rotation, if it is less than the min the player can move and if it is greater than max then it can move too, otherwise movement is disabled by setting Active to false.



```
    If it is rotated enough then nothing happens, however if it is not
if (transform.position.z >= (FurthestMoveUp - transform.localScale.z))
{
    if (transform.eulerAngles[1] < 90 && transform.eulerAngles[1] > -90)
    {
        Active = false;
    }
    else
    {
        Active = true;
    }
}
```

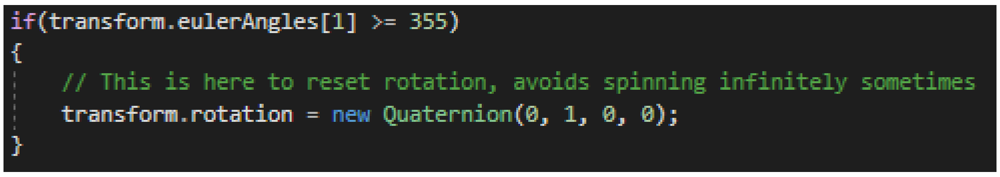
And here is how the function will look when all cords are checked. Active is returned



```
private void CheckBoundaries()
{
    /**|| Each of these nested if statements check if the object has reached or gone past a boundary
          If the object has, the program checks if it is rotated enough to not fall off.
          If it is rotated enough then nothing happens, however if it is not then this function will be called again later ||**/
    if (transform.position.z >= (FurthestMoveUp - transform.localScale.z))
    {
        if (transform.eulerAngles[1] < 90 && transform.eulerAngles[1] > -90)
        {
            Active = false;
        }
        else
        {
            Active = true;
        }
    }

    if (transform.position.x >= (FurthestMoveRight - transform.localScale.x))
    {
        if (transform.eulerAngles[1] < 180 && transform.eulerAngles[1] > 0)
        {
            Active = false;
        }
        else
        {
            Active = true;
        }
    }

    if (transform.position.z <= (FurthestMoveBack + transform.localScale.z))
    {
        if (transform.eulerAngles[1] < 270 && transform.eulerAngles[1] > 90)
        {
            Active = false;
        }
        else
        {
            Active = true;
        }
    }

    if (transform.position.x <= (FurthestMoveLeft + transform.localScale.x))
    {
        if (transform.eulerAngles[1] > 175)
        {
            Active = false;
        }
        else
        {
            Active = true;
        }
    }
}
```

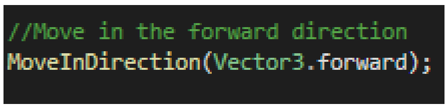
With these out of the way, the brain of the AI is essentially created, now we create the movement. It is somewhat strange to code the brain before the actually movement, but it works for this script quite well. We will create a function called “MoveLogic”. Initially, have the MoveLogic run the CheckBoundaries function. This checks every time the player wants to move if they’re at a boundary.

After this, as a small bug fix we check if the rotation of the AI is nearly at max value, if so reset it back to 1 degree (the code can become buggy if the object is near 360 degrees or 0 degrees). The rotation change is so minor it wont be noticed by the player.



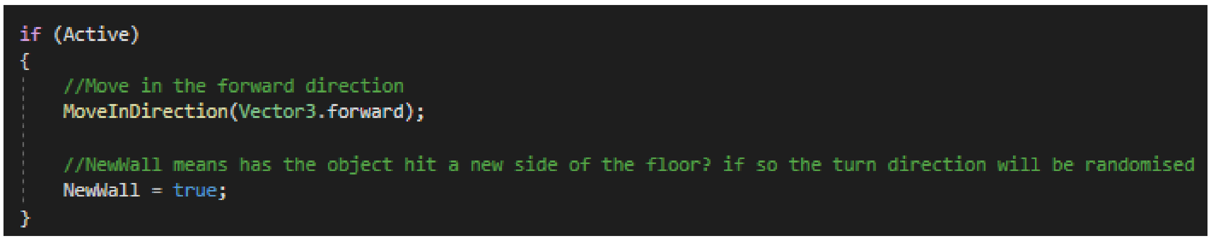
```
if(transform.eulerAngles[1] >= 355)
{
    // This is here to reset rotation, avoids spinning infinitely sometimes
    transform.rotation = new Quaternion(0, 1, 0, 0);
}
```

The next part of the code is the movement code, we will encase it in if (Active) so that if movement is disabled by boundaries it will not run. We simply have the object move forward using:



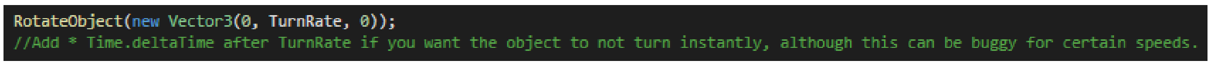
```
//Move in the forward direction
MoveInDirection(Vector3.forward);
```

After this we also set NewWall to true to show the player is now not touching a boundary. Here is how the if statement will look when done:



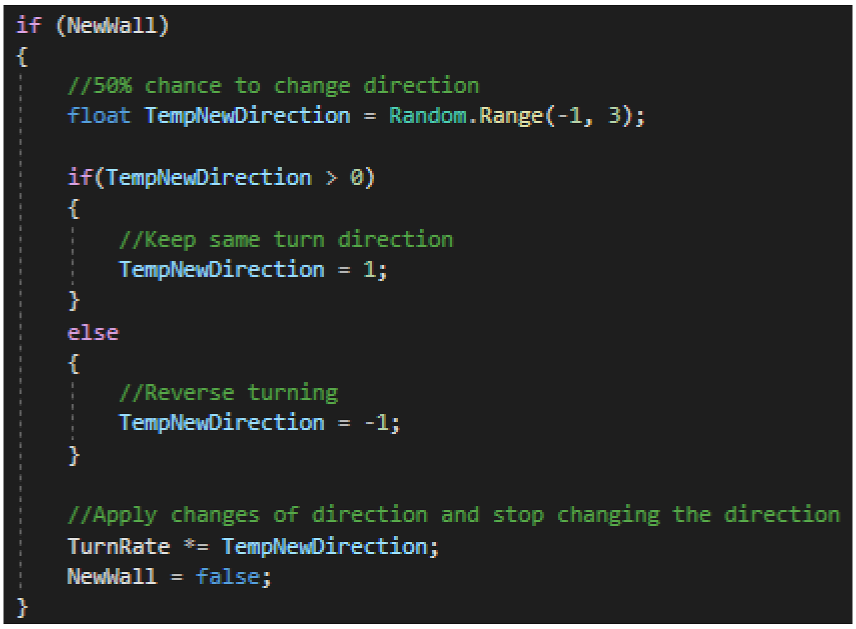
```
if (Active)
{
    //Move in the forward direction
    MoveInDirection(Vector3.forward);

    //NewWall means has the object hit a new side of the floor? if so the turn direction will be randomised
    NewWall = true;
}
```

When the object is touching the boundary we want to rotate it. For this we add an else statement to the if statement, which will run our RotateObject function. Use a new Vector3 with TurnRate as the y coord to do this.



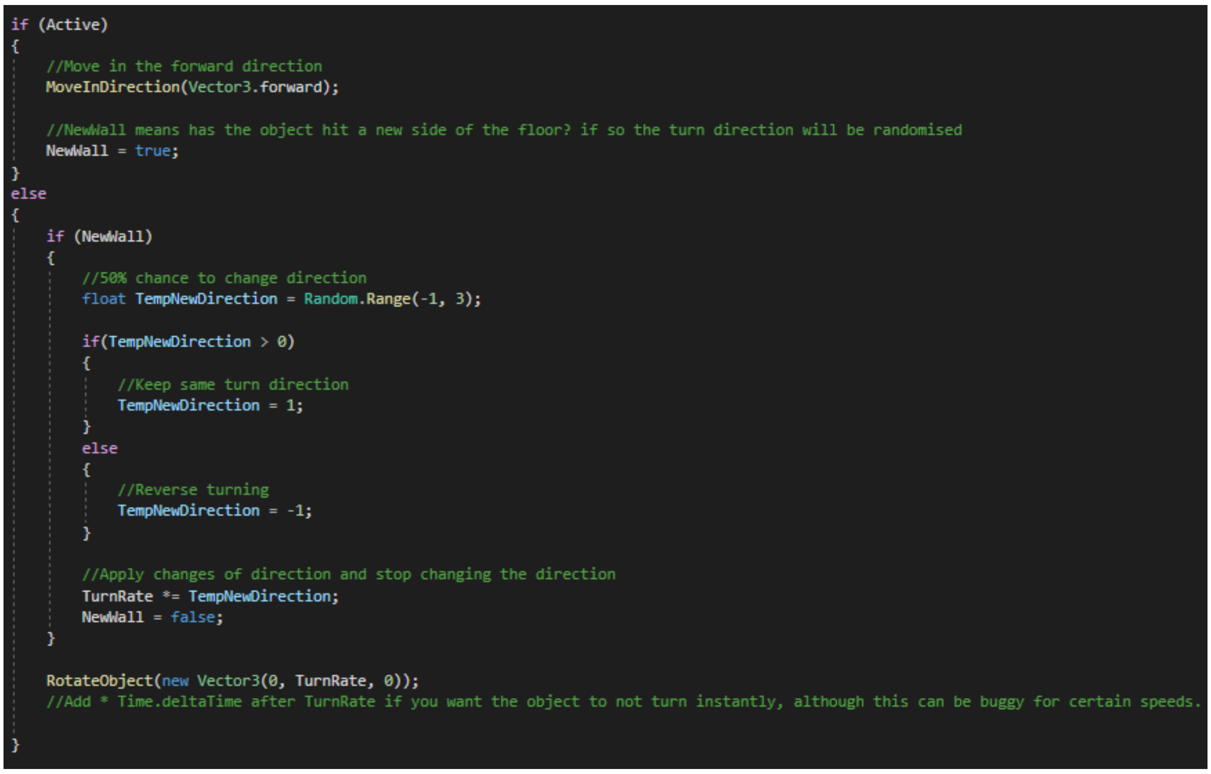
```
RotateObject(new Vector3(0, TurnRate, 0));
//Add * Time.deltaTime after TurnRate if you want the object to not turn instantly, although this can be buggy for certain speeds.
```

However, we also want to add a nested if statement in the else before the rotate function to randomise the turn direction. To do this we ask if (NewWall), and if true we create a new random direction using a temporary float and depending on the float's value we either turn left or right. Use Random.Range to get a random number.



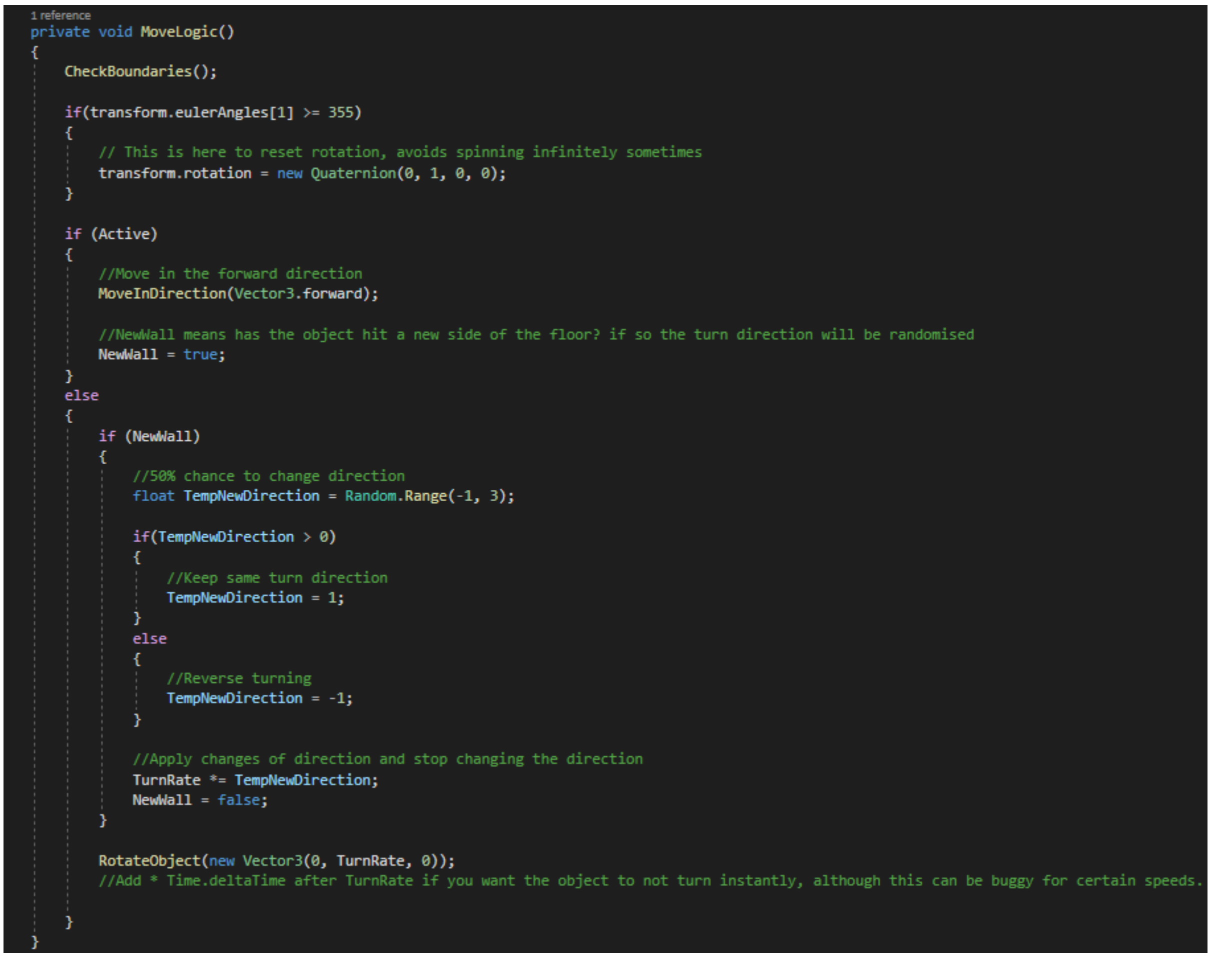
```
if (NewWall)
{
    //50% chance to change direction
    float TempNewDirection = Random.Range(-1, 3);

    if(TempNewDirection > 0)
    {
        //Keep same turn direction
        TempNewDirection = 1;
    }
    else
    {
        //Reverse turning
        TempNewDirection = -1;
    }

    //Apply changes of direction and stop changing the direction
    TurnRate *= TempNewDirection;
    NewWall = false;
}
```

We of course set NewWall to false to stop this checking. Here is how the whole if steatemtne looks together:



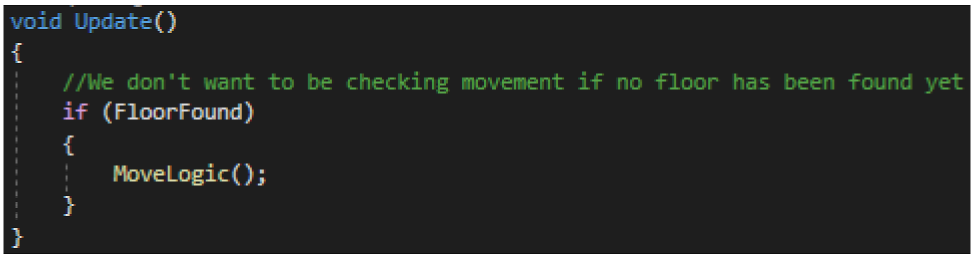
```
if (Active)
{
    //Move in the forward direction
    MoveInDirection(Vector3.forward);

    //NewWall means has the object hit a new side of the floor? if so the turn direction will be randomised
    NewWall = true;
}
else
{
    if (NewWall)
    {
        //50% chance to change direction
        float TempNewDirection = Random.Range(-1, 3);

        if(TempNewDirection > 0)
        {
            //Keep same turn direction
            TempNewDirection = 1;
        }
        else
        {
            //Reverse turning
            TempNewDirection = -1;
        }

        //Apply changes of direction and stop changing the direction
        TurnRate *= TempNewDirection;
        NewWall = false;
    }

    RotateObject(new Vector3(0, TurnRate, 0));
    //Add * Time.deltaTime after TurnRate if you want the object to not turn instantly, although this can be buggy for certain speeds.

}
```

And finally, here is the whole function:



```
1 reference
private void MoveLogic()
{
    CheckBoundaries();

    if(transform.eulerAngles[1] >= 355)
    {
        // This is here to reset rotation, avoids spinning infinitely sometimes
        transform.rotation = new Quaternion(0, 1, 0, 0);
    }

    if (Active)
    {
        //Move in the forward direction
        MoveInDirection(Vector3.forward);

        //NewWall means has the object hit a new side of the floor? if so the turn direction will be randomised
        NewWall = true;
    }
    else
    {
        if (NewWall)
        {
            //50% chance to change direction
            float TempNewDirection = Random.Range(-1, 3);

            if(TempNewDirection > 0)
            {
                //Keep same turn direction
                TempNewDirection = 1;
            }
            else
            {
                //Reverse turning
                TempNewDirection = -1;
            }

            //Apply changes of direction and stop changing the direction
            TurnRate *= TempNewDirection;
            NewWall = false;
        }

        RotateObject(new Vector3(0, TurnRate, 0));
        //Add * Time.deltaTime after TurnRate if you want the object to not turn instantly, although this can be buggy for certain speeds.

    }
}
```

After this we simply add MoveLogic into the Update, with an if statement to check if (FloorFound) and the whole code is good to go.



```
void Update()
{
    //We don't want to be checking movement if no floor has been found yet
    if (FloorFound)
    {
        MoveLogic();
    }
}
```

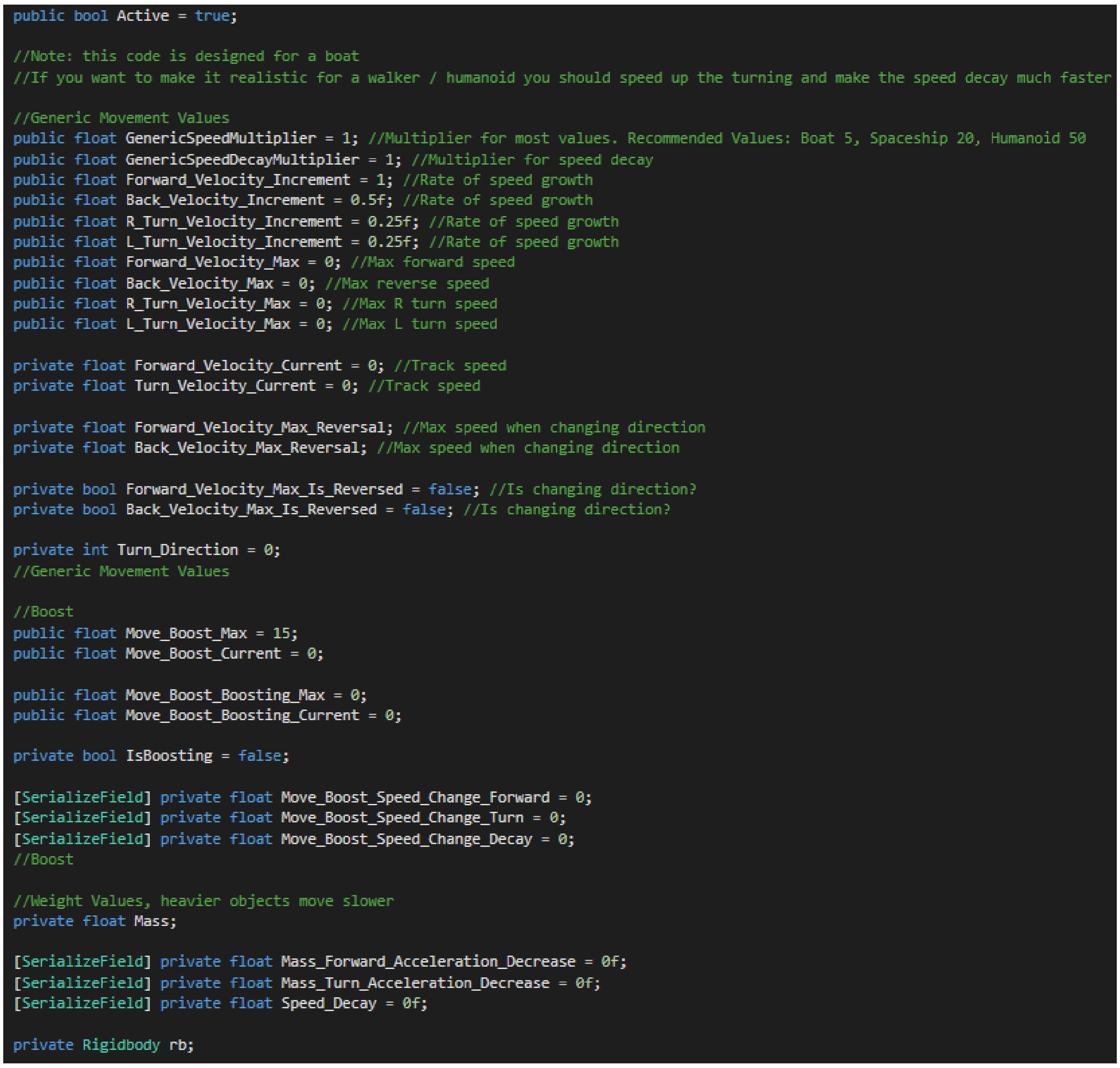
Note: If you're using RB.AddForce instead of transform, use FixedUpdate not Update.

Tutorial 2: Advanced Movement Tutorial

Difficulty: Very Hard

This tutorial covers a more complicated way to simulate movement in unity. This program is essentially what the "complex joint" is to joints in unity, for movement. You can simulate any real-world object's movement in the 2D plane using this program, and it can be adapted simply into 3D and 2.5D although the version I do a tutorial on here is for 2D. This program can work with the unity physics or on its own completely. This code is incredibly "precise" with the movement so there are a lot of variables used.

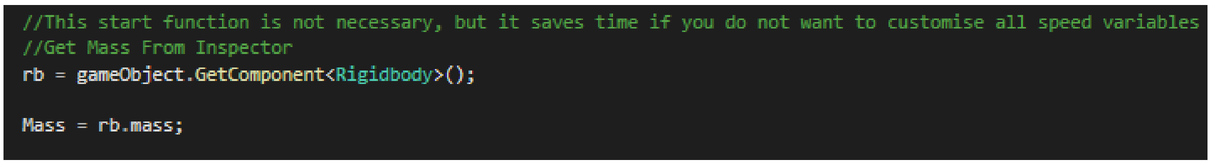
First, we must define a long list of variables. Here is a screenshot of all the variables with explanations on what they are for:



```
public bool Active = true;

//Note: this code is designed for a boat
//If you want to make it realistic for a walker / humanoid you should speed up the turning and make the speed decay much faster

//Generic Movement Values
public float GenericSpeedMultiplier = 1; //Multiplier for most values. Recommended Values: Boat 5, Spaceship 20, Humanoid 50
public float GenericSpeedDecayMultiplier = 1; //Multiplier for speed decay
public float Forward_Velocity_Increment = 1; //Rate of speed growth
public float Back_Velocity_Increment = 0.5f; //Rate of speed growth
public float R_Turn_Velocity_Increment = 0.25f; //Rate of speed growth
public float L_Turn_Velocity_Increment = 0.25f; //Rate of speed growth
public float Forward_Velocity_Max = 0; //Max forward speed
public float Back_Velocity_Max = 0; //Max reverse speed
public float R_Turn_Velocity_Max = 0; //Max R turn speed
public float L_Turn_Velocity_Max = 0; //Max L turn speed

private float Forward_Velocity_Current = 0; //Track speed
private float Turn_Velocity_Current = 0; //Track speed

private float Forward_Velocity_Max_Reversal; //Max speed when changing direction
private float Back_Velocity_Max_Reversal; //Max speed when changing direction

private bool Forward_Velocity_Max_Is_Reversed = false; //Is changing direction?
private bool Back_Velocity_Max_Is_Reversed = false; //Is changing direction?

private int Turn_Direction = 0;
//Generic Movement Values

//Boost
public float Move_Boost_Max = 15;
public float Move_Boost_Current = 0;

public float Move_Boost_Boosting_Max = 0;
public float Move_Boost_Boosting_Current = 0;

private bool IsBoosting = false;

[SerializeField] private float Move_Boost_Speed_Change_Forward = 0;
[SerializeField] private float Move_Boost_Speed_Change_Turn = 0;
[SerializeField] private float Move_Boost_Speed_Change_Decay = 0;
//Boost

//Weight Values, heavier objects move slower
private float Mass;

[SerializeField] private float Mass_Forward_Acceleration_Decrease = 0f;
[SerializeField] private float Mass_Turn_Acceleration_Decrease = 0f;
[SerializeField] private float Speed_Decay = 0f;

private Rigidbody rb;
```

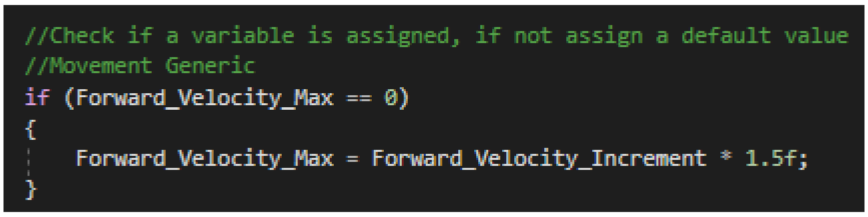
I will explain these more when they come up later in the code.

Firstly, in the Start function we need to find the object's RigidBody and assign the mass value, we do this simply like this:



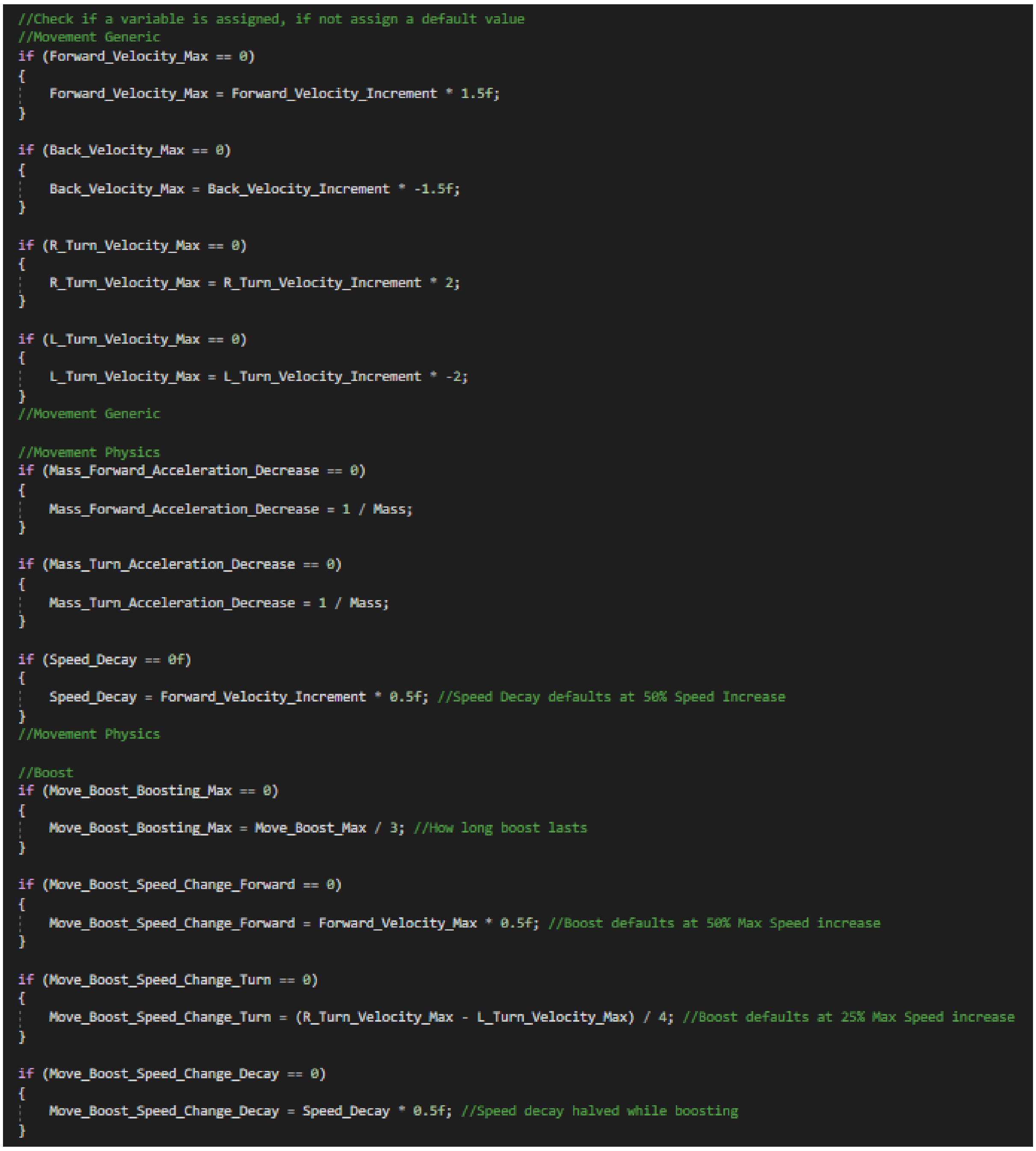
```
//This start function is not necessary, but it saves time if you do not want to customise all speed variables
//Get Mass From Inspector
rb = gameObject.GetComponent<Rigidbody>();

Mass = rb.mass;
```

After this, in the start function we should write a long list of checkers to see if the variables have been assigned in the inspector, if they have no value then default values for a boat are assigned. This is not necessary but is excellent for making sure there is always a value assigned by default. This is good when you must create the script onto a new object rather than importing a prefab. Here is an example of how to do this for one variable:



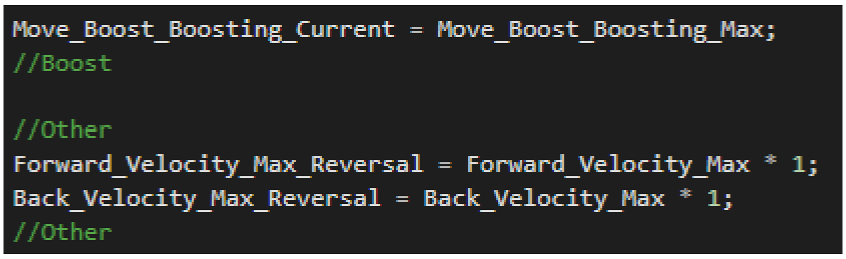
```
//Check if a variable is assigned, if not assign a default value
//Movement Generic
if (Forward_Velocity_Max == 0)
{
    Forward_Velocity_Max = Forward_Velocity_Increment * 1.5f;
}
```

First you check if the value equals 0 (it has not been edited in inspector) then assign a value for it if this is true. Here are the default values I have chosen for all the variables, although you can customise this as you will:



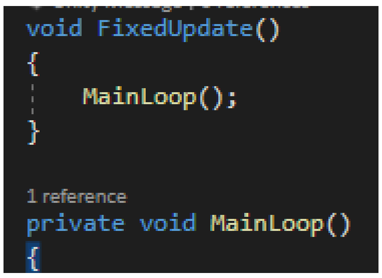
```
//Check if a variable is assigned, if not assign a default value
//Movement Generic
if (Forward_Velocity_Max == 0)
{
    Forward_Velocity_Max = Forward_Velocity_Increment * 1.5f;
}

if (Back_Velocity_Max == 0)
{
    Back_Velocity_Max = Back_Velocity_Increment * -1.5f;
}

if (R_Turn_Velocity_Max == 0)
{
    R_Turn_Velocity_Max = R_Turn_Velocity_Increment * 2;
}

if (L_Turn_Velocity_Max == 0)
{
    L_Turn_Velocity_Max = L_Turn_Velocity_Increment * -2;
}
//Movement Generic

//Movement Physics
if (Mass_Forward_Acceleration_Decrease == 0)
{
    Mass_Forward_Acceleration_Decrease = 1 / Mass;
}

if (Mass_Turn_Acceleration_Decrease == 0)
{
    Mass_Turn_Acceleration_Decrease = 1 / Mass;
}

if (Speed_Decay == 0f)
{
    Speed_Decay = Forward_Velocity_Increment * 0.5f; //Speed Decay defaults at 50% Speed Increase
}
//Movement Physics

//Boost
if (Move_Boost_Boosting_Max == 0)
{
    Move_Boost_Boosting_Max = Move_Boost_Max / 3; //How long boost lasts
}

if (Move_Boost_Speed_Change_Forward == 0)
{
    Move_Boost_Speed_Change_Forward = Forward_Velocity_Max * 0.5f; //Boost defaults at 50% Max Speed increase
}

if (Move_Boost_Speed_Change_Turn == 0)
{
    Move_Boost_Speed_Change_Turn = (R_Turn_Velocity_Max - L_Turn_Velocity_Max) / 4; //Boost defaults at 25% Max Speed increase
}

if (Move_Boost_Speed_Change_Decay == 0)
{
    Move_Boost_Speed_Change_Decay = Speed_Decay * 0.5f; //Speed decay halved while boosting
}
```

The default variables are setup for a boat’s movement system. If you do not want to assign the variables here then you must write them in the inspector, although it will save you a LOT of time assigning the values here instead. Finally, we must assign a couple other values which are dependant on the variables we just assigned. Notably the reversed speed for turning and the boost cooldown tracker. This goes at the end of the Start function:



```
Move_Boost_Boosting_Current = Move_Boost_Boosting_Max;
//Boost

//Other
Forward_Velocity_Max_Reversal = Forward_Velocity_Max * 1;
Back_Velocity_Max_Reversal = Back_Velocity_Max * 1;
//Other
```

Next, replace the Update function with a FixedUpdate function, we will be using Unity physics so Fixed Update is superior to the regular update. As well as this create a new function called MainLoop() and have this run every tick in the FixedLoop.

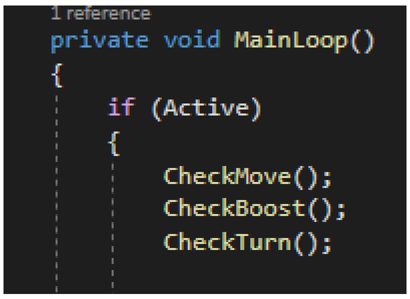


```
void FixedUpdate()
{
    MainLoop();
}

1 reference
private void MainLoop()
{
```

The MainLoop will begin empty but we will add to it over the course of the tutorial.

Now that we have prepared the global variables and the mainloop, we can begin creating our functions. For this module we will create 3 main functions and then bring them together into the MainLoop. Let us start by creating 3 private void functions called CheckMove(), CheckBoost() and CheckTurn(). For now they will be empty.

After this, return to the MainLoop function. Add a simple if statement that asks if (Active) and if this condition is true run the 3 functions we just created. The active variable is useful to disable movement instantly and allows other modules to interact with this object quickly and simply.



```
1 reference
private void MainLoop()
{
    if (Active)
    {
        CheckMove();
        CheckBoost();
        CheckTurn();
```

Underneath this we also want to add the 2 lines of code that edit movement and rotation. Firstly movement, write this line of code:

rb.velocity = transform.forward * Forward_Velocity_Current * GenericSpeedMultiplier * Time.deltaTime;

Rb.velocity: speed of our object, using rigidbody physica

Transform.forward: move in the forward direction

Fwd_vel_current: our current speed (vector3), multiplied by the global speed bonus and multiplied by deltaTime so it is unaffected by framerate.

After this we need to add the code for turning. This is slightly more complicated. Write:

transform.Rotate(new Vector3(0, -Turn_Direction, 0) * Turn_Velocity_Current * GenericSpeedMultiplier * Time.deltaTime, Space.World);

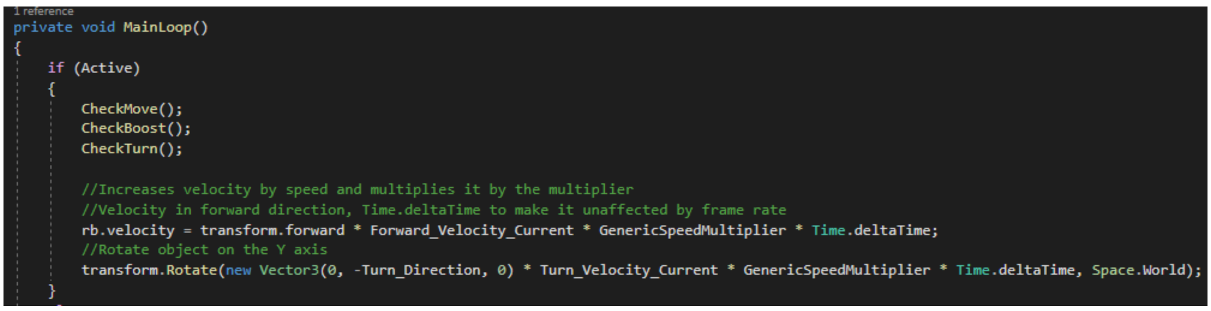
transform.Rotate: turn this object's transform

new Vector3: create a new vector, we set the first and last value to 0 because we don't need to turn in the x or z direction. We set the Y to the opposite of the turn direction, this will either be 1 or -1.

Then we multiply the whole vector 3 by the turn rate, the global speed multiplier and the deltaTime.

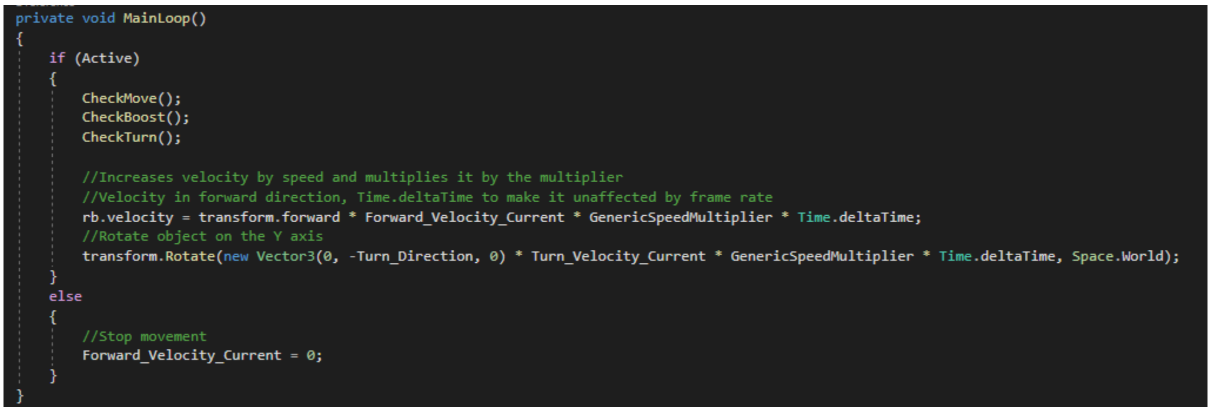
Finally Space.World simply indicates the turning to be done in world space.

Here is how your code should look after this is done:



```
1 reference
private void MainLoop()
{
    if (Active)
    {
        CheckMove();
        CheckBoost();
        CheckTurn();

        //Increases velocity by speed and multiplies it by the multiplier
        //Velocity in forward direction, Time.deltaTime to make it unaffected by frame rate
        rb.velocity = transform.forward * Forward_Velocity_Current * GenericSpeedMultiplier * Time.deltaTime;
        //Rotate object on the Y axis
        transform.Rotate(new Vector3(0, -Turn_Direction, 0) * Turn_Velocity_Current * GenericSpeedMultiplier * Time.deltaTime, Space.World);
    }
```

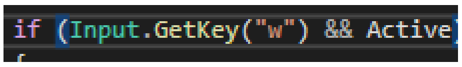
Finally, we want to add an else statement to our if statement that simply sets the Forward_Velocity_Current to 0 to instantly stop the object if its movement is disabled.



```
private void MainLoop()
{
    if (Active)
    {
        CheckMove();
        CheckBoost();
        CheckTurn();

        //Increases velocity by speed and multiplies it by the multiplier
        //Velocity in forward direction, Time.deltaTime to make it unaffected by frame rate
        rb.velocity = transform.forward * Forward_Velocity_Current * GenericSpeedMultiplier * Time.deltaTime;
        //Rotate object on the Y axis
        transform.Rotate(new Vector3(0, -Turn_Direction, 0) * Turn_Velocity_Current * GenericSpeedMultiplier * Time.deltaTime, Space.World);
    }
    else
    {
        //Stop movement
        Forward_Velocity_Current = 0;
    }
}
```

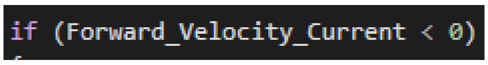
Let's look at creating the code in order we have written them here. The CheckMove function is rather simple, it checks first if the player wants to move up or down. The movement here is however more complex than simply adding or reversing speed. If the player is moving forward, the speed is added on using the player speed value. If the player was previously moving in the opposite direction the max speed is altered to simulate the drag from the previous movement. This should be less extreme for humanoid movement, but much more extreme for boat or spaceship movement due to the greater drag. This can be edited in the public variables in the inspector. If the player has not pressed any key the object will slow down naturally. This is not immediate but somewhat gradual (again can be edited) to simulate drag.

To get this effect we need to first add checkers to see if the user is inputing movement. To do this we add an if statement.



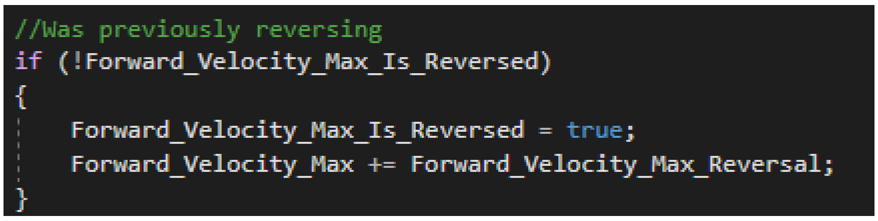
```
if (Input.GetKey("w") && Active)
```

The "and Active" part is less necessary because the mainloop function checks this, I thought to include it however just in case. Inside the if statement we add a nested if statement.



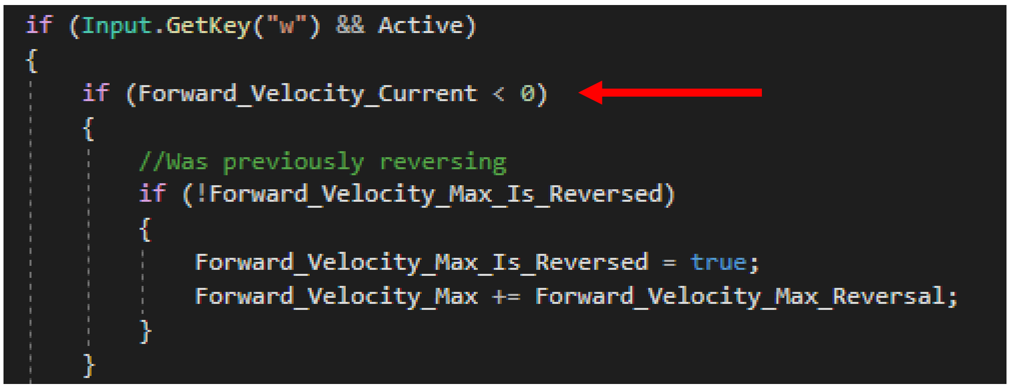
```
if (Forward_Velocity_Current < 0)
```

This checks if the player was previously reversing, if true it means that the player has now changed direction and the speed should be increased using the reversal speed instead of the forward speed. To do this we have to reassign the main movement speed with the reversal speed, which can be done inside the final nested if statement.



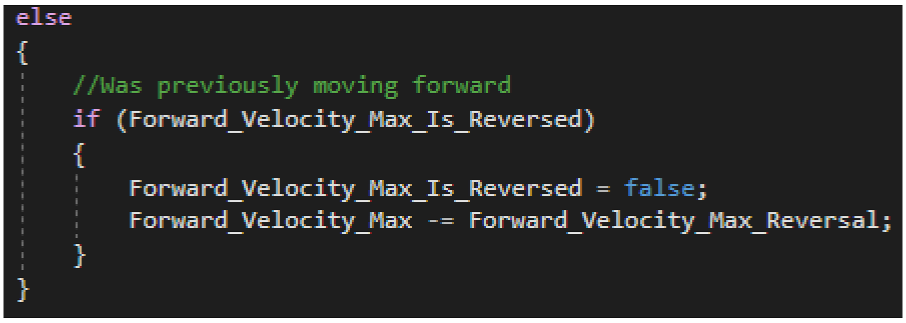
```
//Was previously reversing
if (!Forward_Velocity_Max_Is_Reversed)
{
    Forward_Velocity_Max_Is_Reversed = true;
    Forward_Velocity_Max += Forward_Velocity_Max_Reversal;
}
```

Here is how your if statement should look when completed:



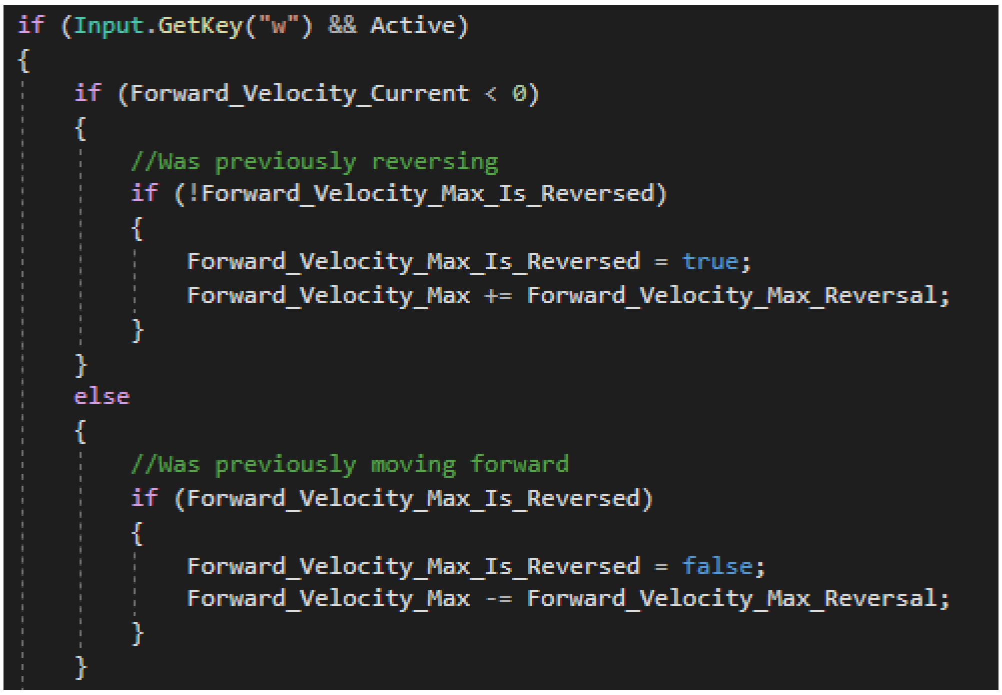
```
if (Input.GetKey("w") && Active)
{
    if (Forward_Velocity_Current < 0)
    {
        //Was previously reversing
        if (!Forward_Velocity_Max_Is_Reversed)
        {
            Forward_Velocity_Max_Is_Reversed = true;
            Forward_Velocity_Max += Forward_Velocity_Max_Reversal;
        }
    }
```

After this we need to add the else statement for the 2nd if statement, the one with a red arrow pointing at it. Inside this function n we simply do the reversal of the above if statement.



```
else
{
    //Was previously moving forward
    if (Forward_Velocity_Max_Is_Reversed)
    {
        Forward_Velocity_Max_Is_Reversed = false;
        Forward_Velocity_Max -= Forward_Velocity_Max_Reversal;
    }
}
```

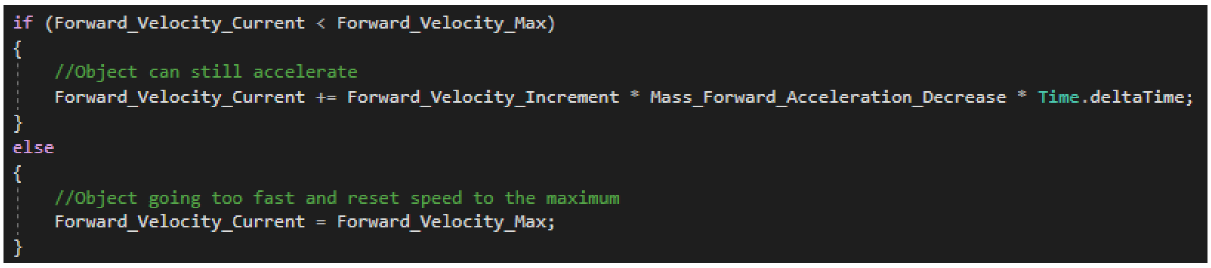
This is done if the player was always moving forward, it is to reset the forward velocity back to its original speed rather than the reversal speed. Here is how your if statement should look up to now:



```
if (Input.GetKey("w") && Active)
{
    if (Forward_Velocity_Current < 0)
    {
        //Was previously reversing
        if (!Forward_Velocity_Max_Is_Reversed)
        {
            Forward_Velocity_Max_Is_Reversed = true;
            Forward_Velocity_Max += Forward_Velocity_Max_Reversal;
        }
    }
    else
    {
        //Was previously moving forward
        if (Forward_Velocity_Max_Is_Reversed)
        {
            Forward_Velocity_Max_Is_Reversed = false;
            Forward_Velocity_Max -= Forward_Velocity_Max_Reversal;
        }
    }
```

To understand this system, it might be good to look at how movement works in other games. Sometimes when you start moving the player will accelerate, turning direction may be faster or slower depending on your main speed, and the player could stumble if you try to move backwards from a fast-running speed. These interactions can be simulated rather well using this code system.

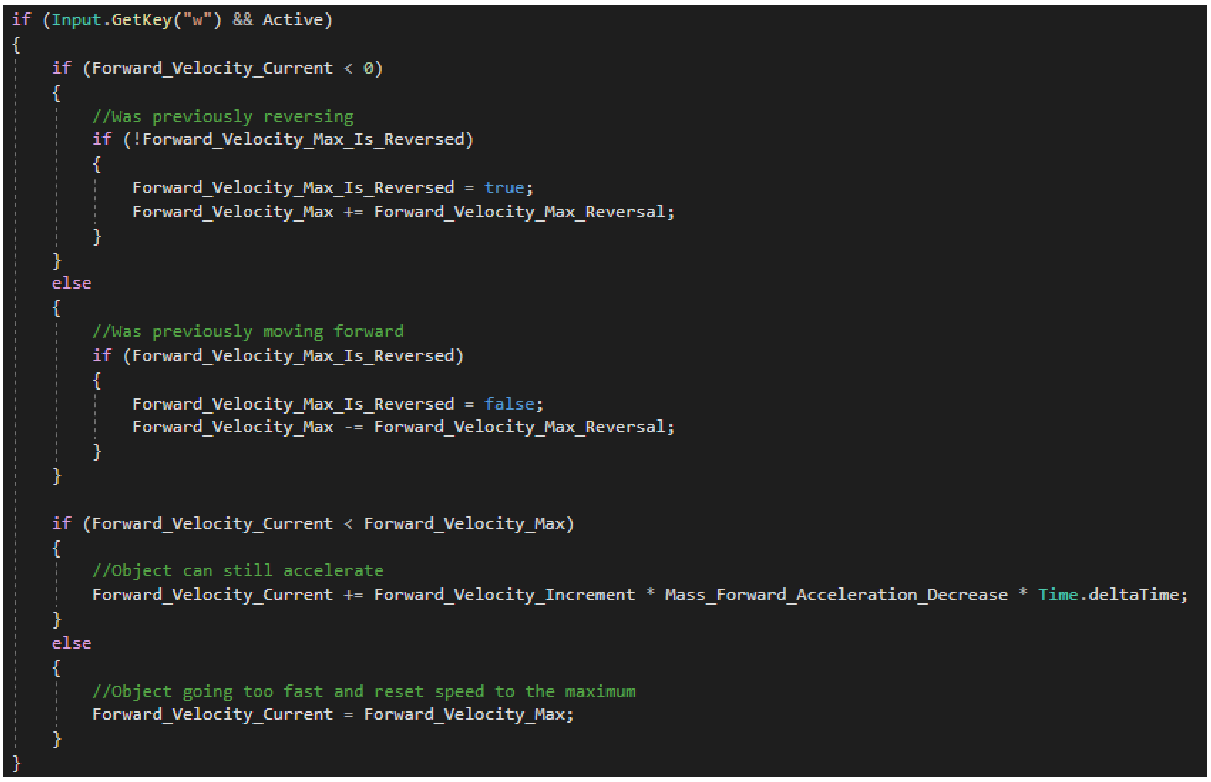
After we have done this part we must code the final part of this input if statement. After the end of the else write a new if-else statement. This is rather simple so I will explain it in one go.



```
if (Forward_Velocity_Current < Forward_Velocity_Max)
{
    //Object can still accelerate
    Forward_Velocity_Current += Forward_Velocity_Increment * Mass_Forward_Acceleration_Decrease * Time.deltaTime;
}
else
{
    //Object going too fast and reset speed to the maximum
    Forward_Velocity_Current = Forward_Velocity_Max;
}
```

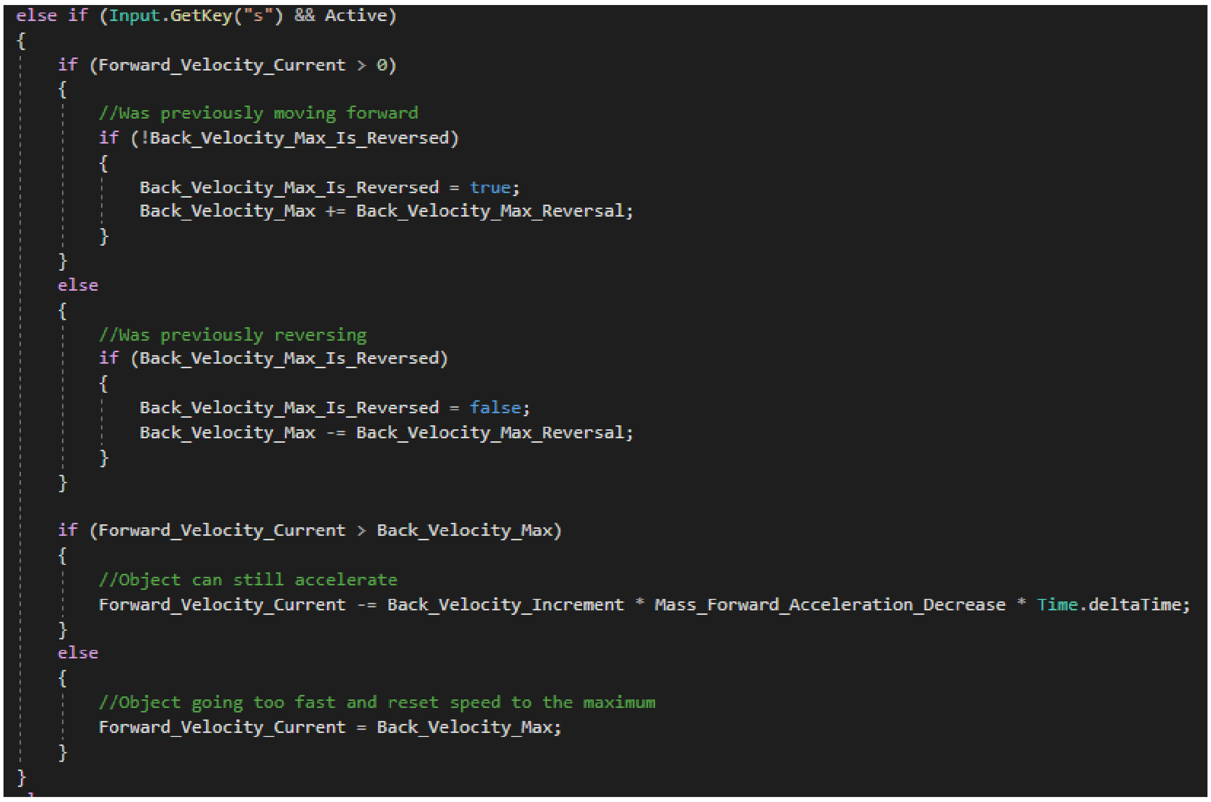
Firstly we check if the current speed is lower than the maximum speed. If this is true then the player should still accelerate. This can be done by adding the current speed with the speed increment (how much speed is gained per tick) multiplied by deltaTime (frame rate does not affect speed). I also added the Mass variable for forward velocity to make heavier objects gain speed slower. The else statement is simply there to make sure the player does not exceed the max speed by resetting the speed back to the max speed.

Now your logic for velocity increase is done for now, here is how the if statement should look in its entirety.



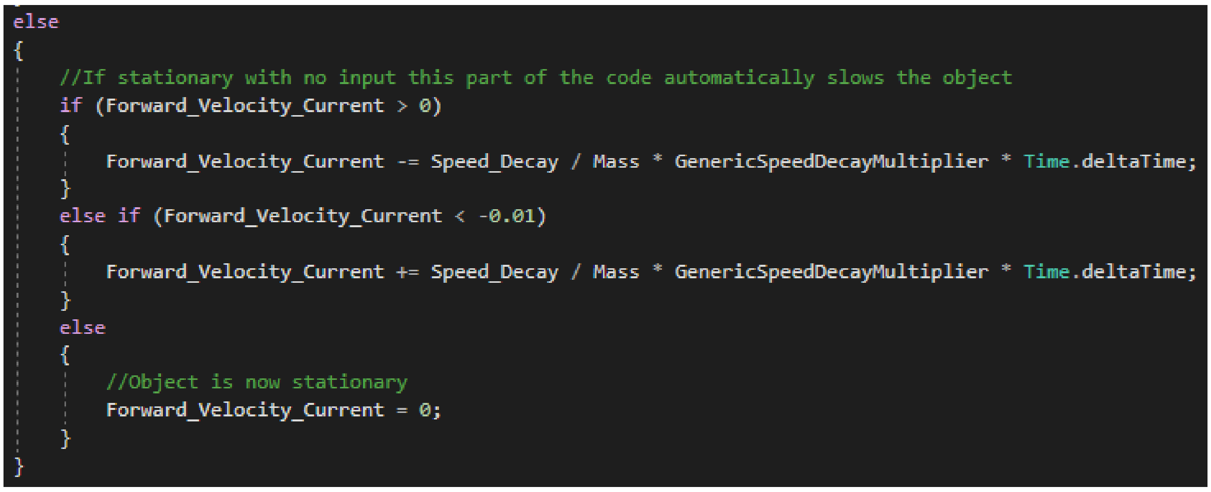
```
if (Input.GetKey("w") && Active)
{
    if (Forward_Velocity_Current < 0)
    {
        //Was previously reversing
        if (!Forward_Velocity_Max_Is_Reversed)
        {
            Forward_Velocity_Max_Is_Reversed = true;
            Forward_Velocity_Max += Forward_Velocity_Max_Reversal;
        }
    }
    else
    {
        //Was previously moving forward
        if (Forward_Velocity_Max_Is_Reversed)
        {
            Forward_Velocity_Max_Is_Reversed = false;
            Forward_Velocity_Max -= Forward_Velocity_Max_Reversal;
        }
    }

    if (Forward_Velocity_Current < Forward_Velocity_Max)
    {
        //Object can still accelerate
        Forward_Velocity_Current += Forward_Velocity_Increment * Mass_Forward_Acceleration_Decrease * Time.deltaTime;
    }
    else
    {
        //Object going too fast and reset speed to the maximum
        Forward_Velocity_Current = Forward_Velocity_Max;
    }
}
```

Next, we have to do the backwards movement, this is done the exact same way as the forward movement but with different variables. Add this code after your main if statement.



```
else if (Input.GetKey("s") && Active)
{
    if (Forward_Velocity_Current > 0)
    {
        //Was previously moving forward
        if (!Back_Velocity_Max_Is_Reversed)
        {
            Back_Velocity_Max_Is_Reversed = true;
            Back_Velocity_Max += Back_Velocity_Max_Reversal;
        }
    }
    else
    {
        //Was previously reversing
        if (Back_Velocity_Max_Is_Reversed)
        {
            Back_Velocity_Max_Is_Reversed = false;
            Back_Velocity_Max -= Back_Velocity_Max_Reversal;
        }
    }

    if (Forward_Velocity_Current > Back_Velocity_Max)
    {
        //Object can still accelerate
        Forward_Velocity_Current -= Back_Velocity_Increment * Mass_Forward_Acceleration_Decrease * Time.deltaTime;
    }
    else
    {
        //Object going too fast and reset speed to the maximum
        Forward_Velocity_Current = Back_Velocity_Max;
    }
}
```

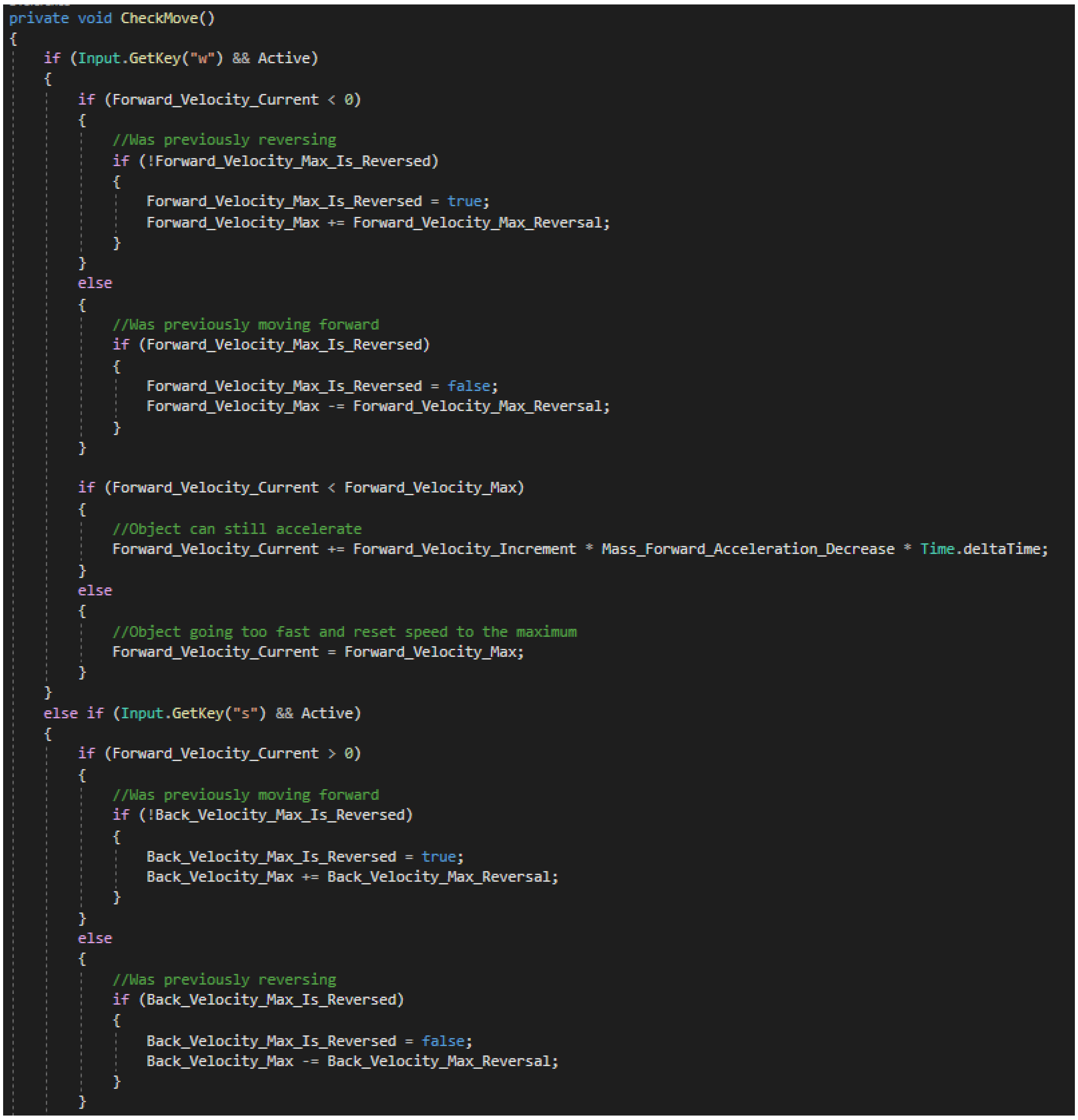
So now we have the movement in the forward and backwards direction, however if we stay still nothing will change. For this we need to add an else to our large if statement. Again this is rather simple so I shall explain it in one go.



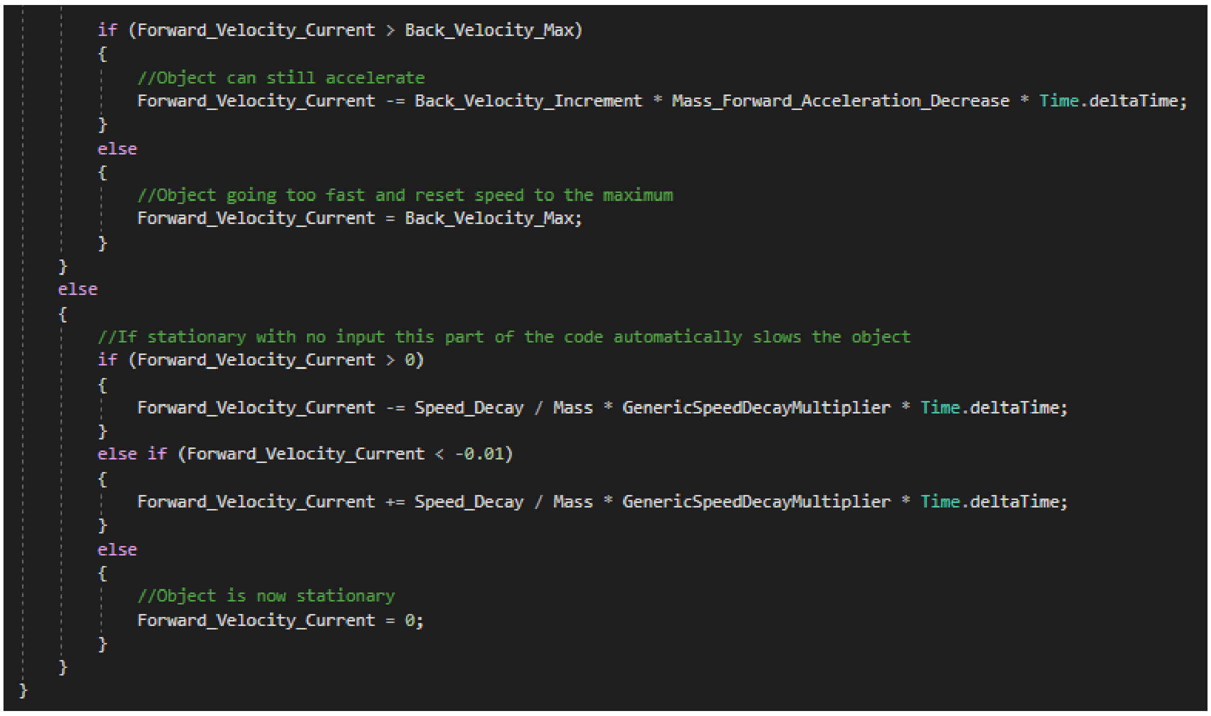
```
else
{
    //If stationary with no input this part of the code automatically slows the object
    if (Forward_Velocity_Current > 0)
    {
        Forward_Velocity_Current -= Speed_Decay / Mass * GenericSpeedDecayMultiplier * Time.deltaTime;
    }
    else if (Forward_Velocity_Current < -0.01)
    {
        Forward_Velocity_Current += Speed_Decay / Mass * GenericSpeedDecayMultiplier * Time.deltaTime;
    }
    else
    {
        //Object is now stationary
        Forward_Velocity_Current = 0;
    }
}
```

We check if the current speed is greater than 0, which means the objects speed needs to be decreased. If so we decrease the velocity gradually by taking away the speed_decay rate, divided by mass (heavier objects slow down slower), multiplied by the speed decay (to simulate a boat will have a lower multiply than say, a humanoid figure) then multiplied by deltaTime.

In the elseif we check if the speed is less than 0 and if so the speed needs to be increased to stop the object. We do the opposite and simply add the speed decay rather than take away. Finally we add an else to stop the object fully. With that done, the CheckMove function is complete!



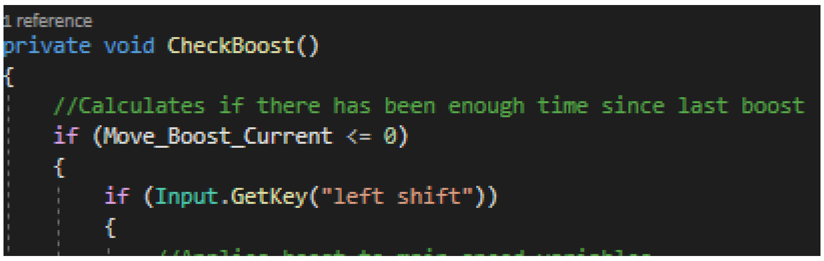
```
private void CheckMove()
{
    if (Input.GetKey("w") && Active)
    {
        if (Forward_Velocity_Current < 0)
        {
            //Was previously reversing
            if (!Forward_Velocity_Max_Is_Reversed)
            {
                Forward_Velocity_Max_Is_Reversed = true;
                Forward_Velocity_Max += Forward_Velocity_Max_Reversal;
            }
        }
        else
        {
            //Was previously moving forward
            if (Forward_Velocity_Max_Is_Reversed)
            {
                Forward_Velocity_Max_Is_Reversed = false;
                Forward_Velocity_Max -= Forward_Velocity_Max_Reversal;
            }
        }

        if (Forward_Velocity_Current < Forward_Velocity_Max)
        {
            //Object can still accelerate
            Forward_Velocity_Current += Forward_Velocity_Increment * Mass_Forward_Acceleration_Decrease * Time.deltaTime;
        }
        else
        {
            //Object going too fast and reset speed to the maximum
            Forward_Velocity_Current = Forward_Velocity_Max;
        }
    }
    else if (Input.GetKey("s") && Active)
    {
        if (Forward_Velocity_Current > 0)
        {
            //Was previously moving forward
            if (!Back_Velocity_Max_Is_Reversed)
            {
                Back_Velocity_Max_Is_Reversed = true;
                Back_Velocity_Max += Back_Velocity_Max_Reversal;
            }
        }
        else
        {
            //Was previously reversing
            if (Back_Velocity_Max_Is_Reversed)
            {
                Back_Velocity_Max_Is_Reversed = false;
                Back_Velocity_Max -= Back_Velocity_Max_Reversal;
            }
        }
```


```
        if (Forward_Velocity_Current > Back_Velocity_Max)
        {
            //Object can still accelerate
            Forward_Velocity_Current -= Back_Velocity_Increment * Mass_Forward_Acceleration_Decrease * Time.deltaTime;
        }
        else
        {
            //Object going too fast and reset speed to the maximum
            Forward_Velocity_Current = Back_Velocity_Max;
        }
    }
    else
    {
        //If stationary with no input this part of the code automatically slows the object
        if (Forward_Velocity_Current > 0)
        {
            Forward_Velocity_Current -= Speed_Decay / Mass * GenericSpeedDecayMultiplier * Time.deltaTime;
        }
        else if (Forward_Velocity_Current < -0.01)
        {
            Forward_Velocity_Current += Speed_Decay / Mass * GenericSpeedDecayMultiplier * Time.deltaTime;
        }
        else
        {
            //Object is now stationary
            Forward_Velocity_Current = 0;
        }
    }
}
```

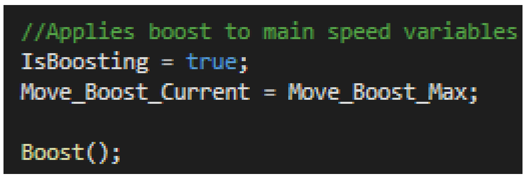
Remember the Mainloop, it changes the speed using Forward_Velocity_Current, so this is the most important part of the CheckMove function.

Next we will look at the CheckBoost function. The CheckBoost function checks first if the player can boost (if it has cooled down), if the player is not ready then the cooldown continues decreasing. If the player is ready, and presses shift then the boost happens which runs the Boost function. Let us see how this is done.

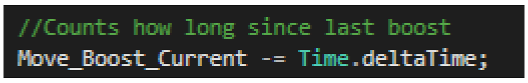
First create an if-else statement in the CheckBoost function with the conditions if (Move_Boost_Current <= 0). If this is true it means the boost can be performed, so we must check for user input using a nested if statement that checks if (Input.GetKey("left shift")).


```
1 reference
private void CheckBoost()
{
    //Calculates if there has been enough time since last boost
    if (Move_Boost_Current <= 0)
    {
        if (Input.GetKey("left shift"))
        {
```

Inside the nested if statement we want to start boosting. We have not coded the boost function yet but we will get to it soon. Write these lines of code and run the boost function if the player is pressing left shift:

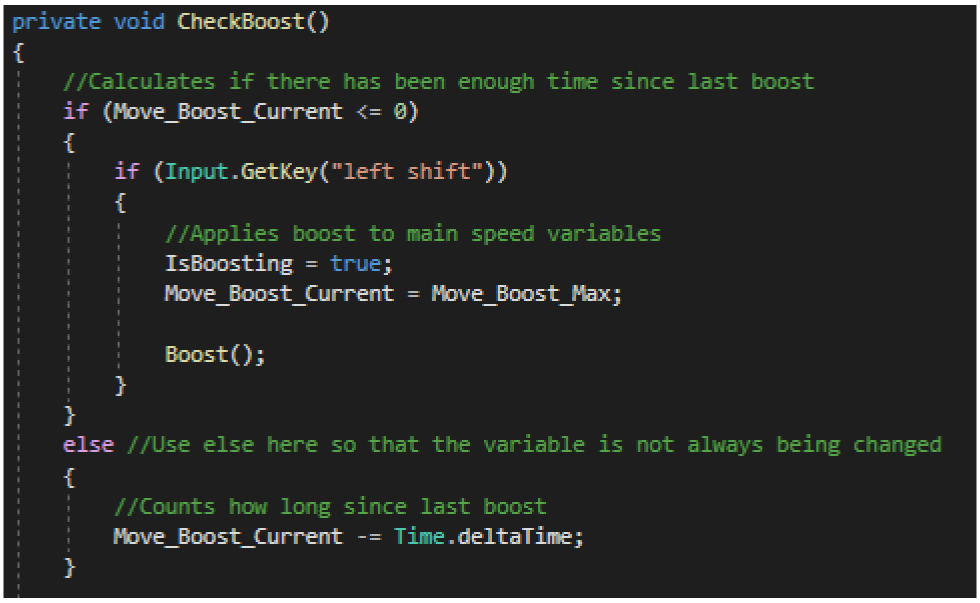

```
//Applies boost to main speed variables
IsBoosting = true;
Move_Boost_Current = Move_Boost_Max;

Boost();
```

Now, in the else statement for the initial if statement write:



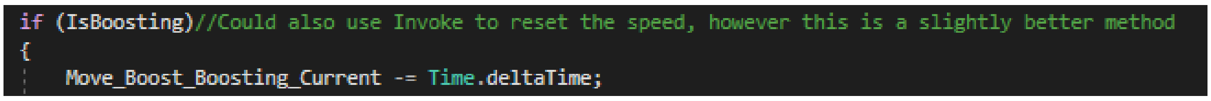
```
//Counts how long since last boost
Move_Boost_Current -= Time.deltaTime;
```

This acts as a countdown to 0. Notice if the player presses shift the countdown sets itself to the max value for boost cooldown, so this is used to count down. Here is how the if-else should look now:



```
private void CheckBoost()
{
    //Calculates if there has been enough time since last boost
    if (Move_Boost_Current <= 0)
    {
        if (Input.GetKey("left shift"))
        {
            //Applies boost to main speed variables
            IsBoosting = true;
            Move_Boost_Current = Move_Boost_Max;

            Boost();
        }
    }
    else //Use else here so that the variable is not always being changed
    {
        //Counts how long since last boost
        Move_Boost_Current -= Time.deltaTime;
    }
```

Finally we need to reset the boost speed if the player is boosting. For this write a second if statement underneat the first, if(IsBoosting). If this is true, do Move_Boost_Boosting_Current -= Time.DeltaTime.



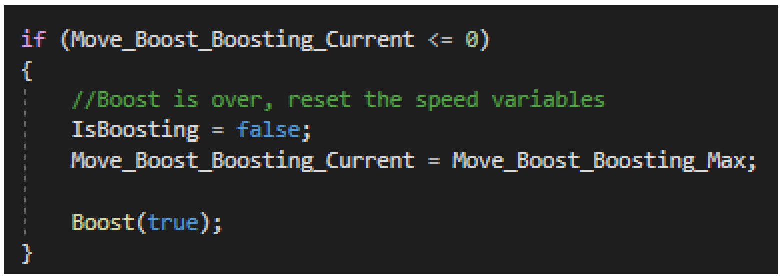
```
if (IsBoosting)//Could also use Invoke to reset the speed, however this is a slightly better method
{
    Move_Boost_Boosting_Current -= Time.deltaTime;
```

This is similar to the above method but we must use different variables. This is because sometimes you want the cooldown of the boost to be greater than the actual boost duration.

Move_Boost_Max = cooldown of boosting

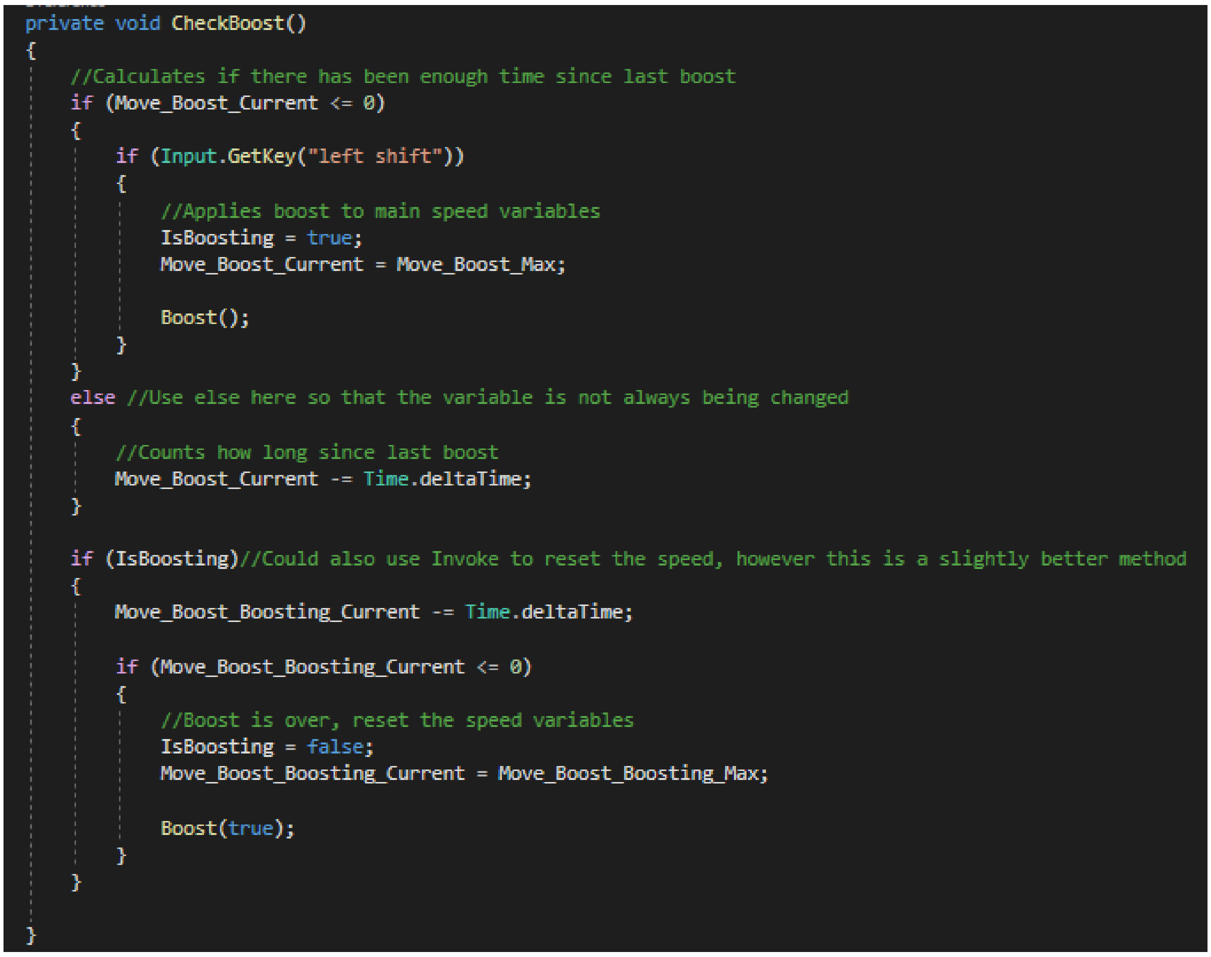
Move_Boost_Boosting_Max = duration of boost

Finally we create a similar nested if statement as above to reset the boost if the duration is over.



```
if (Move_Boost_Boosting_Current <= 0)
{
    //Boost is over, reset the speed variables
    IsBoosting = false;
    Move_Boost_Boosting_Current = Move_Boost_Boosting_Max;

    Boost(true);
}
```

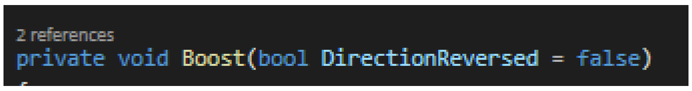
Notice we use true as an attribute to Boost, I will explain this when we write the Boost function. Here is how your finished function should look now:



```
private void CheckBoost()
{
    //Calculates if there has been enough time since last boost
    if (Move_Boost_Current <= 0)
    {
        if (Input.GetKey("left shift"))
        {
            //Applies boost to main speed variables
            IsBoosting = true;
            Move_Boost_Current = Move_Boost_Max;

            Boost();
        }
    }
    else //Use else here so that the variable is not always being changed
    {
        //Counts how long since last boost
        Move_Boost_Current -= Time.deltaTime;
    }

    if (IsBoosting)//Could also use Invoke to reset the speed, however this is a slightly better method
    {
        Move_Boost_Boosting_Current -= Time.deltaTime;

        if (Move_Boost_Boosting_Current <= 0)
        {
            //Boost is over, reset the speed variables
            IsBoosting = false;
            Move_Boost_Boosting_Current = Move_Boost_Boosting_Max;

            Boost(true);
        }
    }

}
```

Now onto the final function, the boost function. This function is rather short. The Boost function uses a Boolean system to see if the player speed needs to be increased or decreased. This allows the same function to perform 2 purposes. The boost increases the max speed and turn speed by a set amount, but if the bool is set to true then it is reset back down to the default speeds. After the player has completed the boost (boost is set to a limited time) then the Boost function is called again in the CheckBoost function, but this time set to true so the speed is reset. Let’s see how this is done.

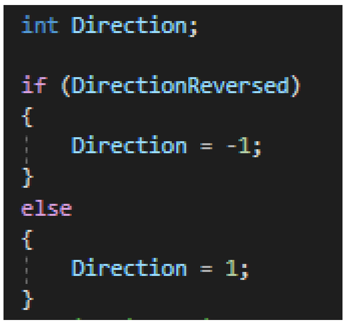
First we create a function called Boost, that inherits a Boolean.



```
2 references
private void Boost(bool DirectionReversed = false)
```

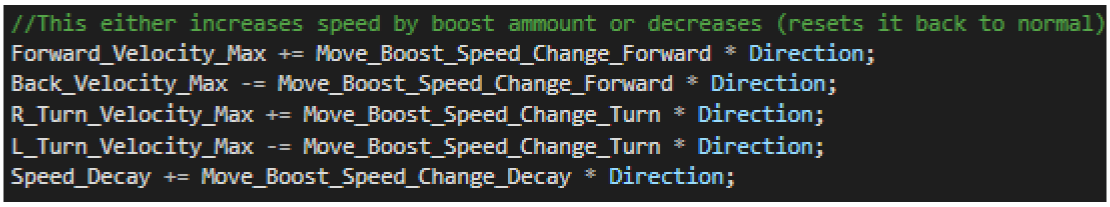
Notice it is set by default to false so that you don’t have to write false every time you run the function.

Next we create a local integer variable called Direction, and if DirectionReversed is true we set this to -1, else to 1.



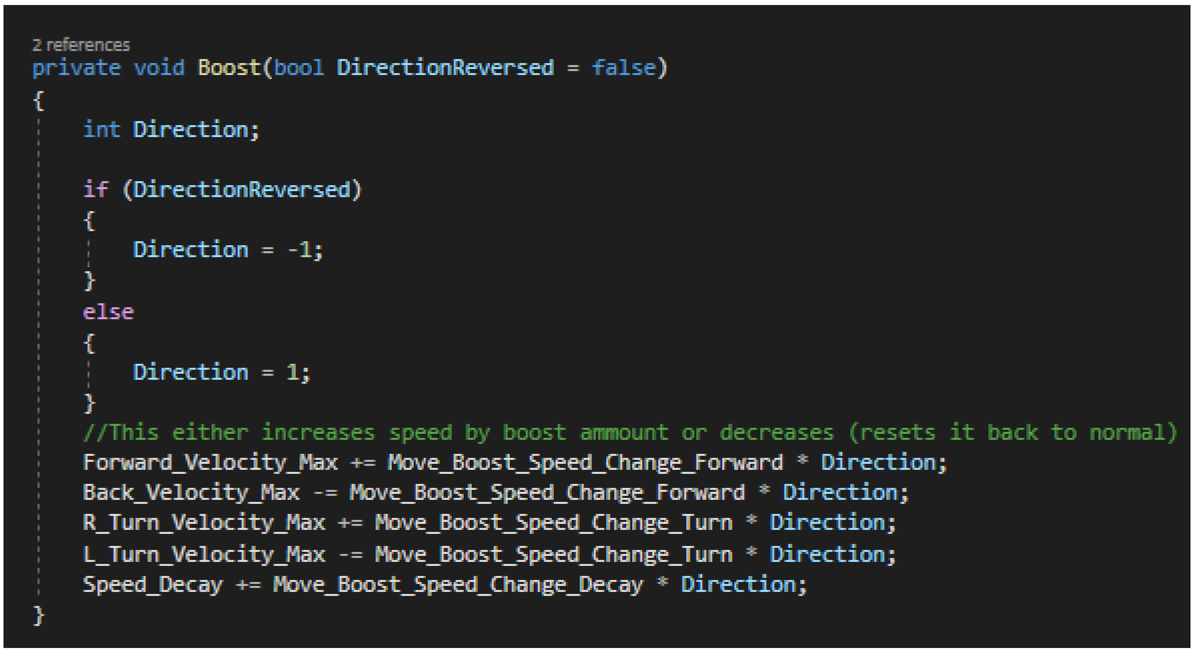
```
int Direction;

if (DirectionReversed)
{
    Direction = -1;
}
else
{
    Direction = 1;
}
```

Finally we assign the boost multiplied by the Direction to a set of variables that we want to be affected by boosting.



```
//This either increases speed by boost ammount or decreases (resets it back to normal)
Forward_Velocity_Max += Move_Boost_Speed_Change_Forward * Direction;
Back_Velocity_Max -= Move_Boost_Speed_Change_Forward * Direction;
R_Turn_Velocity_Max += Move_Boost_Speed_Change_Turn * Direction;
L_Turn_Velocity_Max -= Move_Boost_Speed_Change_Turn * Direction;
Speed_Decay += Move_Boost_Speed_Change_Decay * Direction;
```

Notice we use – instead of plus for things moving left or backwards. This is because to move to the left is in the negative x direction and to move backwards is in the negative y direction. Here is how the finished function should look:



```
2 references
private void Boost(bool DirectionReversed = false)
{
    int Direction;

    if (DirectionReversed)
    {
        Direction = -1;
    }
    else
    {
        Direction = 1;
    }
    //This either increases speed by boost ammount or decreases (resets it back to normal)
    Forward_Velocity_Max += Move_Boost_Speed_Change_Forward * Direction;
    Back_Velocity_Max -= Move_Boost_Speed_Change_Forward * Direction;
    R_Turn_Velocity_Max += Move_Boost_Speed_Change_Turn * Direction;
    L_Turn_Velocity_Max -= Move_Boost_Speed_Change_Turn * Direction;
    Speed_Decay += Move_Boost_Speed_Change_Decay * Direction;
}
```

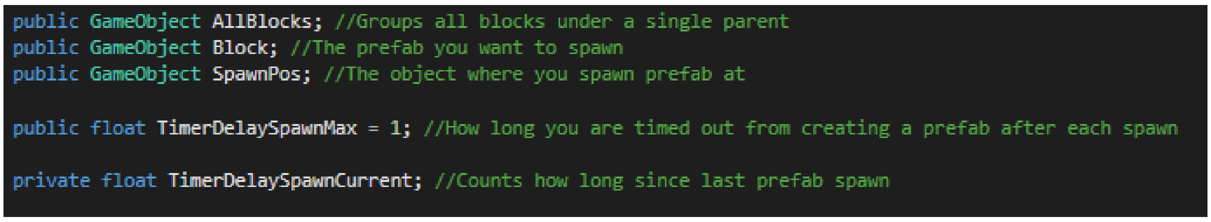
Congratulations with this done your code is finished! It should be around 300 to 350 lines of code.

<u>Tutorial 3: Block Placer Tutorial</u>

<u>Difficulty: Easy</u>

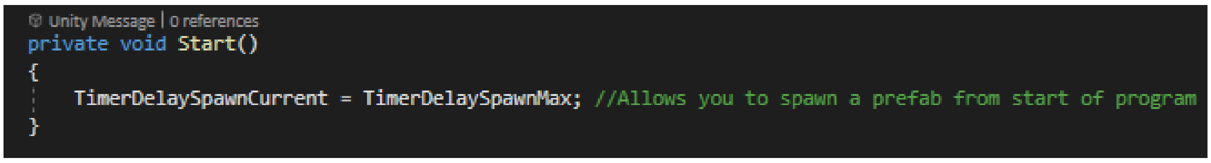
Here is a simple tutorial on how to instantiate a prefab at a given location correctly, and to group it into a parent object for good practise. You can put this on any object and assign its values in the inspector. I recommend using a rigidbody on the Block object and remove the collider from the spawner. This works for Unity3D and 2D

Firstly, create the global variables. The in-code comments explain these quite well



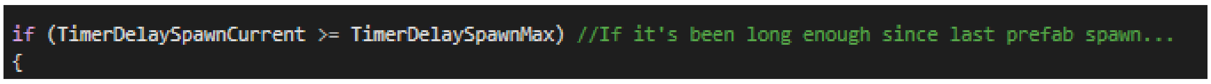
```
public GameObject AllBlocks; //Groups all blocks under a single parent
public GameObject Block; //The prefab you want to spawn
public GameObject SpawnPos; //The object where you spawn prefab at

public float TimerDelaySpawnMax = 1; //How long you are timed out from creating a prefab after each spawn

private float TimerDelaySpawnCurrent; //Counts how long since last prefab spawn
```

Next, in the Star function, rese the TimerDelaySpawnCurrent to the Max, this is optional but allows you to spawn an object straight after loading the scene.



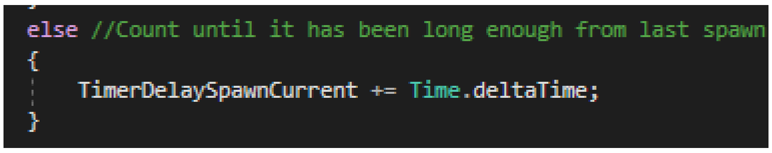
```
Unity Message | 0 references
private void Start()
{
    TimerDelaySpawnCurrent = TimerDelaySpawnMax; //Allows you to spawn a prefab from start of program
}
```

Next in the update function we need to check if it is possible to drop an object, to do this we use an if-else statement.



```
if (TimerDelaySpawnCurrent >= TimerDelaySpawnMax) //If it's been long enough since last prefab spawn...
{
```

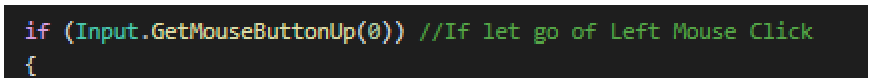
And in the else:



```
else //Count until it has been long enough from last spawn
{
    TimerDelaySpawnCurrent += Time.deltaTime;
}
```

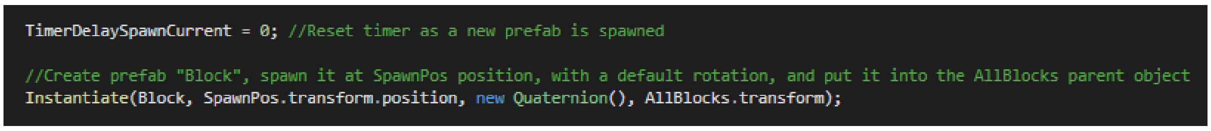
This asks if it’s been another time since the last drop, and if not then add the time.

Inside the if statement we need to check if the player wants to drop the object, to do this we check the input.



```
if (Input.GetMouseButtonUp(0)) //If let go of Left Mouse Click
{
```

And finally we add the drop code. We need to reset the timer to stop multiple drops and then instatnatie the object.



```
TimerDelaySpawnCurrent = 0; //Reset timer as a new prefab is spawned

//Create prefab "Block", spawn it at SpawnPos position, with a default rotation, and put it into the AllBlocks parent object
Instantiate(Block, SpawnPos.transform.position, new Quaternion(), AllBlocks.transform);
```

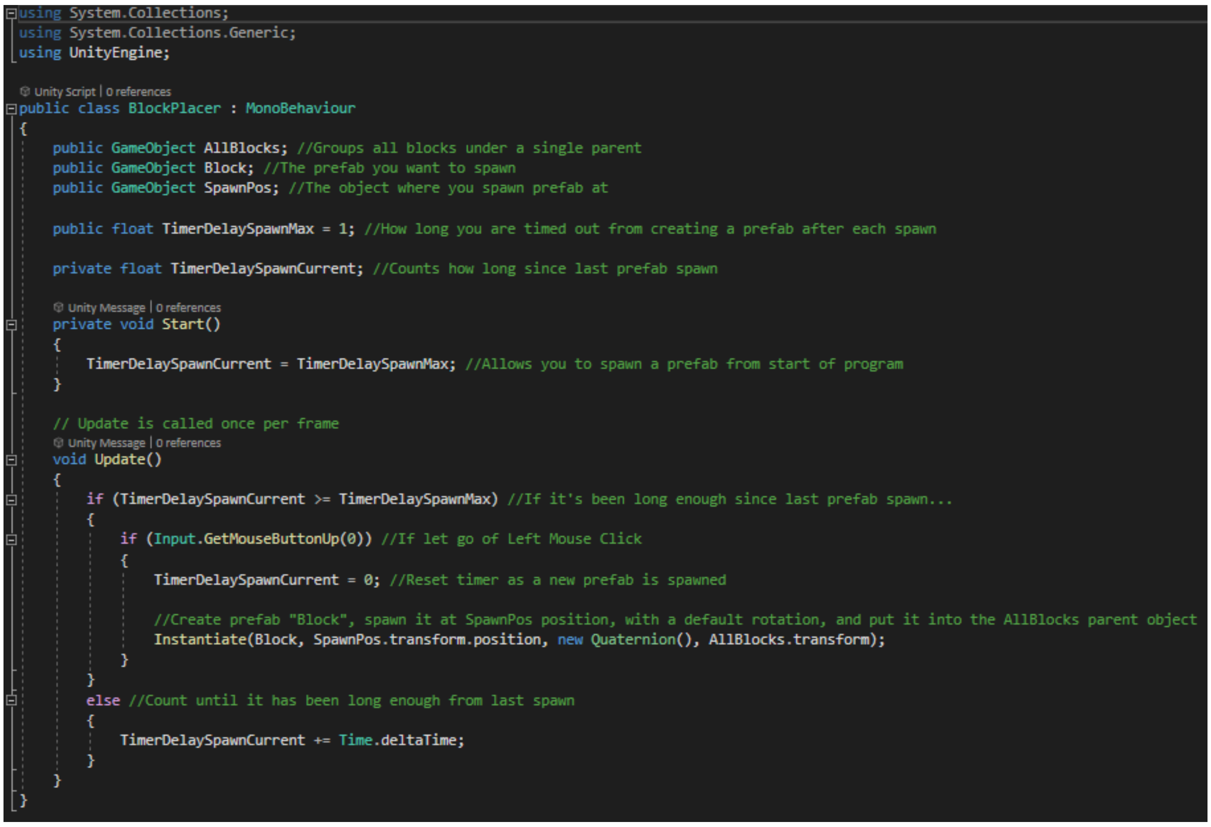
Block = object we are dropping

SpawnPos.transform.postion = get the spawn position

new Quanternion() = default rotation (0,0,0,0)

AllBlocks.transform = make the parent of this object the AllBlocks.

Here is how your finished code should look:



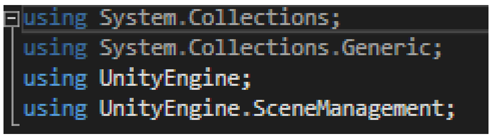
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

Unity Script | 0 references
public class BlockPlacer : MonoBehaviour
{
    public GameObject AllBlocks; //Groups all blocks under a single parent
    public GameObject Block; //The prefab you want to spawn
    public GameObject SpawnPos; //The object where you spawn prefab at

    public float TimerDelaySpawnMax = 1; //How long you are timed out from creating a prefab after each spawn

    private float TimerDelaySpawnCurrent; //Counts how long since last prefab spawn

    Unity Message | 0 references
    private void Start()
    {
        TimerDelaySpawnCurrent = TimerDelaySpawnMax; //Allows you to spawn a prefab from start of program
    }

    // Update is called once per frame
    Unity Message | 0 references
    void Update()
    {
        if (TimerDelaySpawnCurrent >= TimerDelaySpawnMax) //If it's been long enough since last prefab spawn...
        {
            if (Input.GetMouseButtonUp(0)) //If let go of Left Mouse Click
            {
                TimerDelaySpawnCurrent = 0; //Reset timer as a new prefab is spawned

                //Create prefab "Block", spawn it at SpawnPos position, with a default rotation, and put it into the AllBlocks parent object
                Instantiate(Block, SpawnPos.transform.position, new Quaternion(), AllBlocks.transform);
            }
        }
        else //Count until it has been long enough from last spawn
        {
            TimerDelaySpawnCurrent += Time.deltaTime;
        }
    }
}
```

Tutorial 4: Death Layer

Difficulty: Easy/Medium

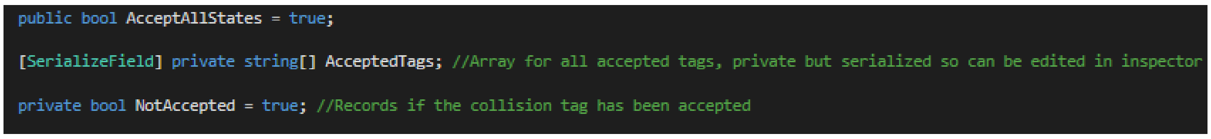
I used this in all my tutorials as a simple fail-state prefab in case the scene is no longer playable, or to clean up the scene when an object falls. It is not necessary to read the other tutorials, but I thought I should make this tutorial on this simple prefab anyway.

You must use UnityEngine.SceneManagement for this module.



```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;
```

Firstly create 3 global variables.



```
public bool AcceptAllStates = true;

[SerializeField] private string[] AcceptedTags; //Array for all accepted tags, private but serialized so can be edited in inspector

private bool NotAccepted = true; //Records if the collision tag has been accepted
```

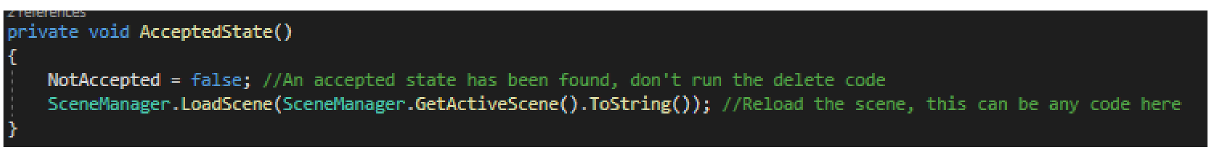
Accept all states allows any object to be accepted by the object, this will also ignore the AcceptedTags if set to true.

AcceptedTags is a private SerializeField. This is so it can be accessed in the inspector but is not public so it is localised only to this object. This somewhat makes this variable a constant, although it could be edited via script if need be. This constant is used if the first Boolean is set to false, and only objects with the tags inside this list can be accepted by the object.

Finally, the private NotAccepted is used to check if the checked object has been accepted using iteration of the AcceptedTags list.

In this code, if an object is accepted it will restart the scene (could be used when the player hits the DeathLayer) otherwise the object is simply deleted.

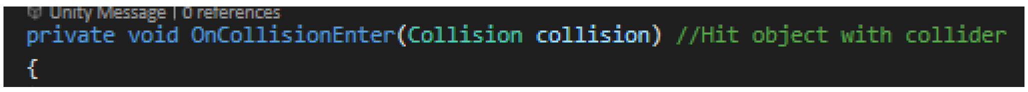
Let us first code the Accepted state, for this you can put anything you want inside it depending on what interaction you want to happen when a special object collides with the DeathLayer. For now here is the code for making the level restart:



```
2 references
private void AcceptedState()
{
    NotAccepted = false; //An accepted state has been found, don't run the delete code
    SceneManager.LoadScene(SceneManager.GetActiveScene().ToString()); //Reload the scene, this can be any code here
}
```

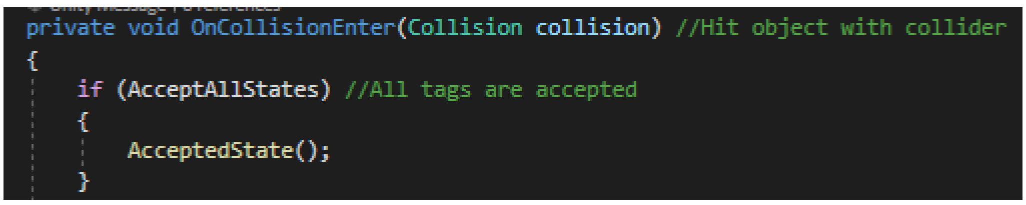
The in-code comments explain this part well.

Now we must code the collision. Firstly, make an OnCollisionEnter function.



```
Unity Message | 0 references
private void OnCollisionEnter(Collision collision) //Hit object with collider
{
```

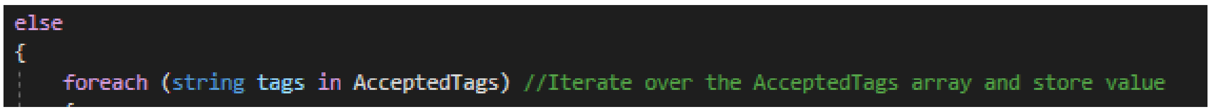
After this simply check if(AcceptedAllStates) and then run the function.



```
private void OnCollisionEnter(Collision collision) //Hit object with collider
{
    if (AcceptAllStates) //All tags are accepted
    {
        AcceptedState();
    }
```

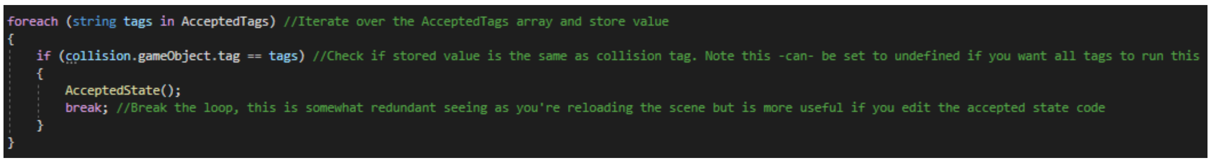
At this point you can check if the AcceptedStates function is working by colliding an object with the DeathLayer.

Next we create an else statement. This statement must iterate over all the tags inside the AcceptedTags list.



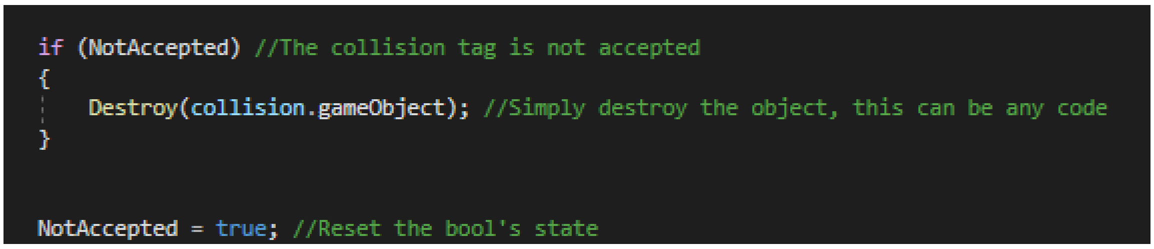
```
else
{
    foreach (string tags in AcceptedTags) //Iterate over the AcceptedTags array and store value
```

This foreach statement stores the values of the list temporarily into the local tags variable. We use this to check wether the collision object has the same tag as an accepted tag. If true we run the accepted state and break the for loop.



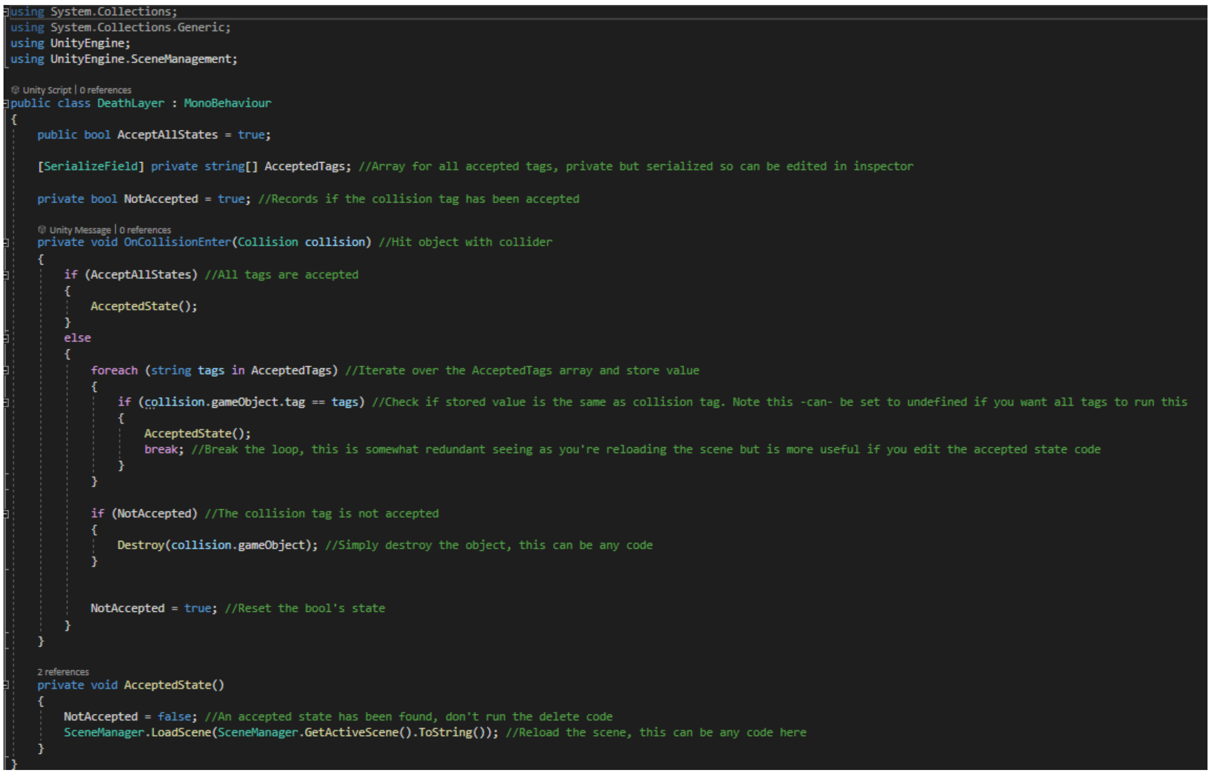
```
foreach (string tags in AcceptedTags) //Iterate over the AcceptedTags array and store value
{
    if (collision.gameObject.tag == tags) //Check if stored value is the same as collision tag. Note this -can- be set to undefined if you want all tags to run this
    {
        AcceptedState();
        break; //Break the loop, this is somewhat redundant seeing as you're reloading the scene but is more useful if you edit the accepted state code
    }
}
```

After this we see what happens if the object is not accepted, to do thie we simply do:



```
if (NotAccepted) //The collision tag is not accepted
{
    Destroy(collision.gameObject); //Simply destroy the object, this can be any code
}


NotAccepted = true; //Reset the bool's state
```

You can put any code here instead of Destroy, but I add this just to delete the object. Finally we rest the NotAccepted tag to true to bring the module back to its starting value (although this is only important if you are allowing an object to collide multiple times with the death layer, ie if it teleports the player). Here is how your whole code should look now



```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;

Unity Script | 0 references
public class DeathLayer : MonoBehaviour
{
    public bool AcceptAllStates = true;

    [SerializeField] private string[] AcceptedTags; //Array for all accepted tags, private but serialized so can be edited in inspector

    private bool NotAccepted = true; //Records if the collision tag has been accepted

    Unity Message | 0 references
    private void OnCollisionEnter(Collision collision) //Hit object with collider
    {
        if (AcceptAllStates) //All tags are accepted
        {
            AcceptedState();
        }
        else
        {
            foreach (string tags in AcceptedTags) //Iterate over the AcceptedTags array and store value
            {
                if (collision.gameObject.tag == tags) //Check if stored value is the same as collision tag. Note this -can- be set to undefined if you want all tags to run this
                {
                    AcceptedState();
                    break; //Break the loop, this is somewhat redundant seeing as you're reloading the scene but is more useful if you edit the accepted state code
                }
            }

            if (NotAccepted) //The collision tag is not accepted
            {
                Destroy(collision.gameObject); //Simply destroy the object, this can be any code
            }


            NotAccepted = true; //Reset the bool's state
        }
    }

    2 references
    private void AcceptedState()
    {
        NotAccepted = false; //An accepted state has been found, don't run the delete code
        SceneManager.LoadScene(SceneManager.GetActiveScene().ToString()); //Reload the scene, this can be any code here
    }
}
```