COMP2007 - Game Development

Week 4 - Code session

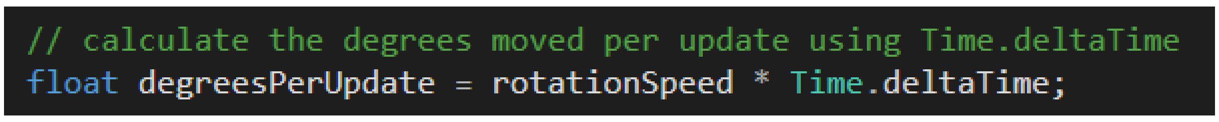
Rotate

Three strategies for rotating transforms, Local, world and Orbit.

Local rotation

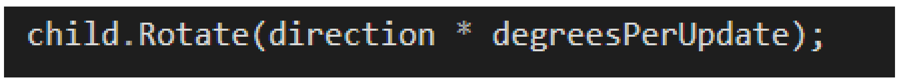
Best used when the transform rotates as part of a hierarchy, which itself may rotate
For example a gun turret on a vehicle, particularly a flying vehicle!

Rotate a transform along its LOCAL axis using the Transform.Rotate method



```
// calculate the degrees moved per update using Time.deltaTime
float degreesPerUpdate = rotationSpeed * Time.deltaTime;
```

NOTE: the Rotate method will default to local space if not specified by a second parameter

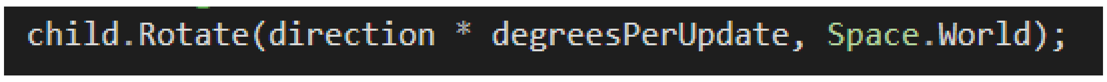


```
child.Rotate(direction * degreesPerUpdate);
```

World rotation

Best used when the transform is a parent of a hierarchy
For example a vehicle or character root transform

Rotate a transform along the WORLD axis using the Transform.Rotate method
NOTE: the second parameter "Space.World" specifies the rotation in world coordinates



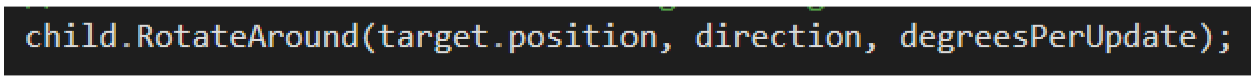
```
child.Rotate(direction * degreesPerUpdate, Space.World);
```

Orbiting a target

Useful for orbiting objects such as planets, magical force fields or transforms with a spherical offset from a target
For example a solar system with a sun, planets and moons

RotateAround will orbit a target position on a specified axis at a set speed

- Target position - Vector3 position
- Direction - Vector3 as an axis (like transform.forward etc)
    - Normalised this before use for stable results
- Degrees per update - rotate speed

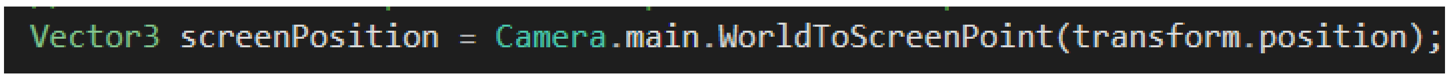


```
child.RotateAround(target.position, direction, degreesPerUpdate);
```

Follow - face the mouse

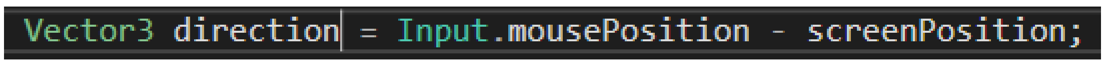
A simple example to rotate a transform in the direction of the mouse cursor

STEP 1: We get the screen position from our transform and the camera



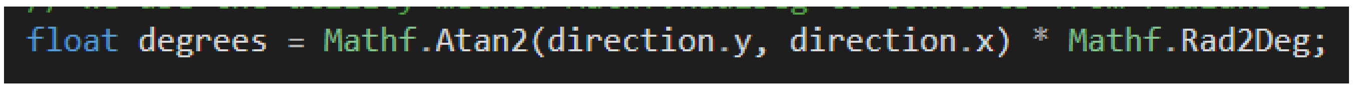
```
Vector3 screenPosition = Camera.main.WorldToScreenPoint(transform.position);
```

STEP 2: get a heading or direction from the mouse position to the screen position



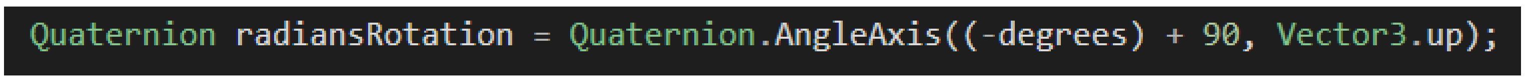
```
Vector3 direction = Input.mousePosition - screenPosition;
```

STEP 3: calculate the Z angle from the X and Y of the direction
NOTE: Mathf.Atan2 uses trigonometry to calculate the angle in radians
We multiply the result of Atan2 with Mathf.Rad2Deg to convert the radian angle to degrees



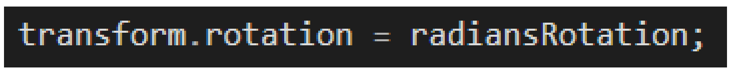
```
float degrees = Mathf.Atan2(direction.y, direction.x) * Mathf.Rad2Deg;
```

STEP 4: create a quaternion to apply the rotation to our transform
Quaternion.AngleAxis returns our radians based rotation from a single angle in degrees and an axis
NOTE we adjust the degrees to a 3-quarter turn (270 degrees) to face the mouse, this is due to unity rotation actually facing to the right!



```
Quaternion radiansRotation = Quaternion.AngleAxis((-degrees) + 90, Vector3.up);
```

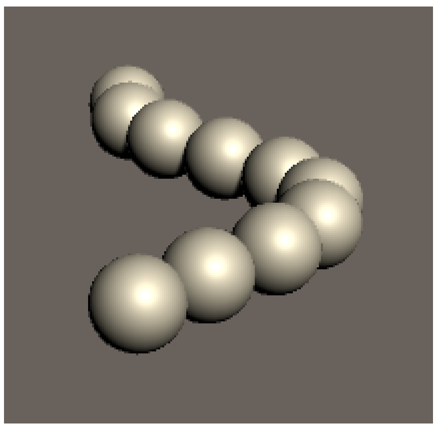
STEP 5: apply the rotation
The end rotation can be applied directly to the rotation on the transform
NOTE: transform.rotation is in radians, rotations have to be provided as a Quaternion



```
transform.rotation = radiansRotation;
```

Sine Waves

Create a line of gameobjects and oscillate them using a sine wave

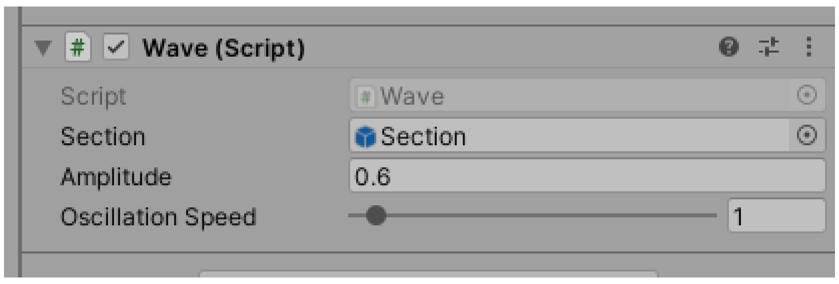


The Wave component

Section: the prefab you wish to spawn for each section

Amplitude: the distance a section travels left to right

Oscillation speed: the movement speed of each section from left to right



Wave (Script)

Script: Wave

Section: Section

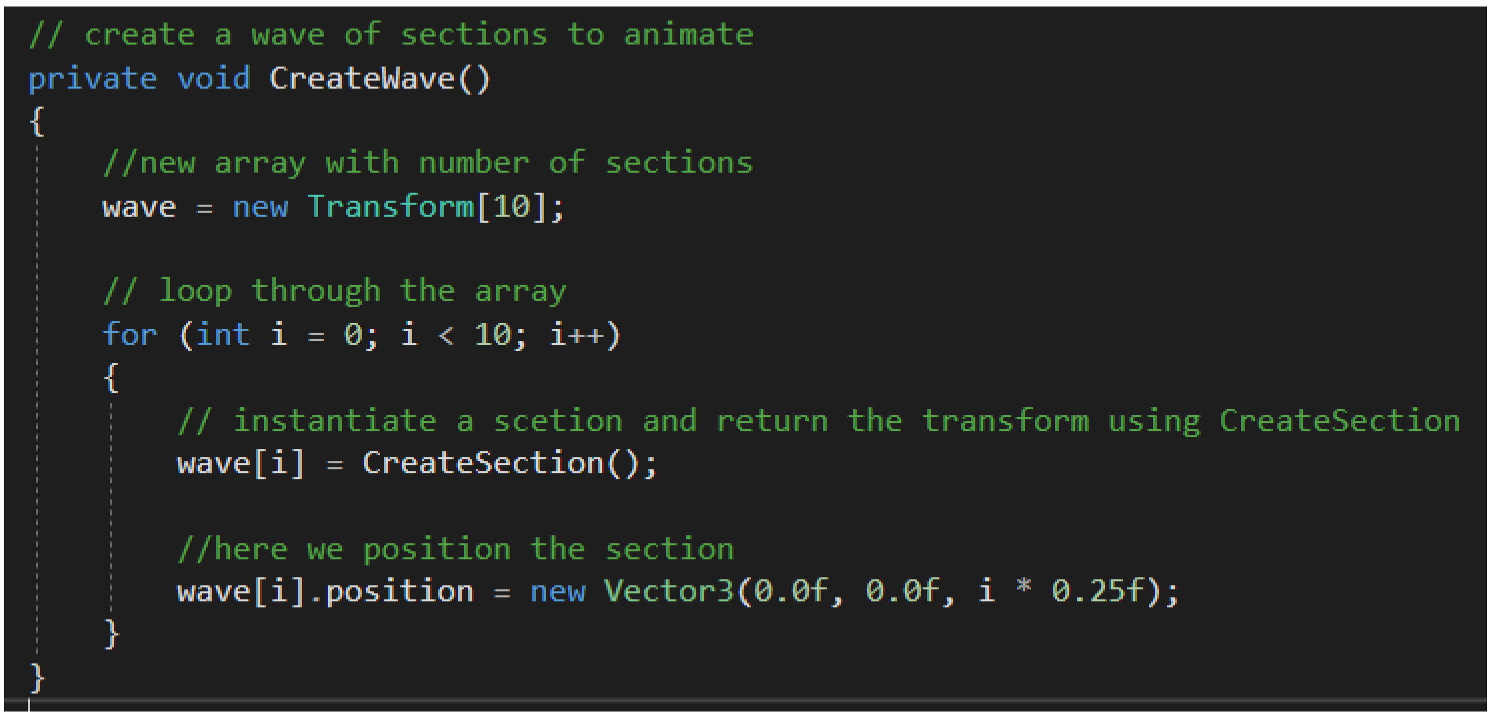
Amplitude: 0.6

Oscillation Speed: 1

The CreateWave method sets up our sections into an array we can use in the update method to animate them

Note we setup the positions of each transform

There is a utility method called CreateSection that simple instantiates a section into the scene and returns it transform for our wave array



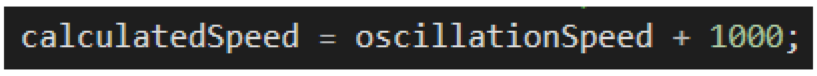
```
// create a wave of sections to animate
private void CreateWave()
{
    //new array with number of sections
    wave = new Transform[10];

    // loop through the array
    for (int i = 0; i < 10; i++)
    {
        // instantiate a scetion and return the transform using CreateSection
        wave[i] = CreateSection();

        //here we position the section
        wave[i].position = new Vector3(0.0f, 0.0f, i * 0.25f);
    }
}
```

Within the update method, we calculate our speed by adding 1000 to it

NOTE: This actually stops the oscillation from moving too fast, it can crash the game if set too low!



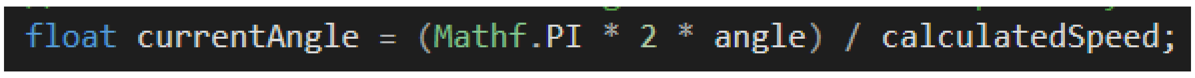
```
calculatedSpeed = oscillationSpeed + 1000;
```

Within a loop for each section we calculate the angle (or x position)

The angle variable used in the code is added to at the end of the loop to adjust for the loop index and shift all sections along a little bit

To convert the angle to degrees, we use Mathf.PI * 2, a standard trigonometry formula

We divide the angle by the calculated speed to set our movement speed along the X axis for the angle

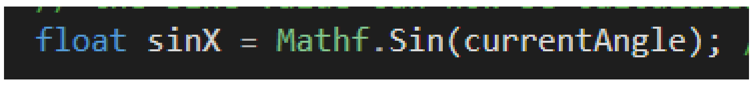


```
float currentAngle = (Mathf.PI * 2 * angle) / calculatedSpeed;
```

Here we use Mathf.Sin - a Sine calculation graph a value from our current angle

Consider Mathf.Sin like an animation curve, we can "graph" or "plot" a position along a sine wave

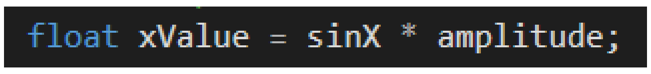
The value returned from Mathf.Sin is our graphed or plotted value from the input, currentAngle



```
float sinX = Mathf.Sin(currentAngle); /
```

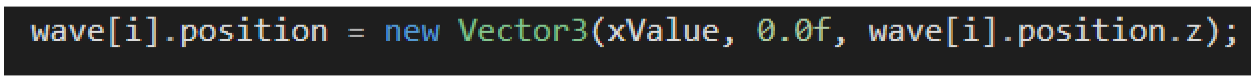
We calculate our final x value by multiplying the sine x by our amplitude

The amplitude is the distance from left to right we want each section to move



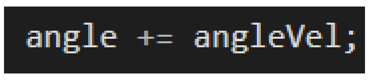
```
float xValue = sinX * amplitude;
```

Here the position is set (in the loop) for each section
The X axis of the position is our calculated xValue, zero for Y and no change to the Z axis



```
wave[i].position = new Vector3(xValue, 0.0f, wave[i].position.z);
```

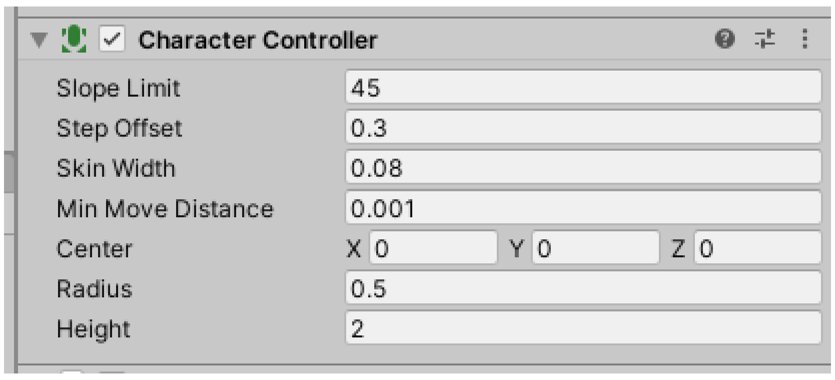
Lastly we nudge the angle by our angle velocity so each section moves a little each time we go through the sections loop



```
angle += angleVel;
```

Third person camera

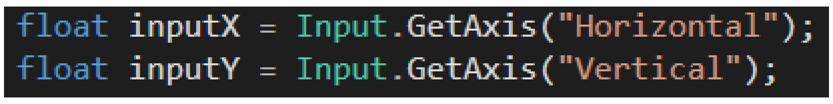
This example uses a CharacterController component for the movement part with default settings
A CharacterController is often used instead of a Collider as it provides extra functionality for movement as well as collision



☑ Character Controller

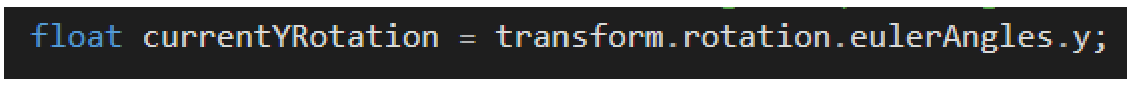
Slope Limit	45
Step Offset	0.3
Skin Width	0.08
Min Move Distance	0.001
Center	X 0 Y 0 Z 0
Radius	0.5
Height	2

We get our X and Y input the standard way for both keyboard and joystick



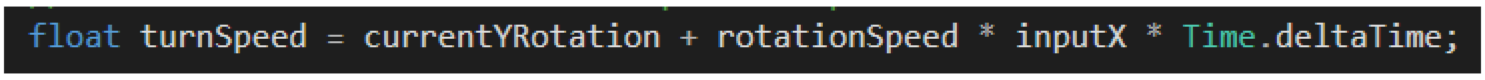
```
float inputX = Input.GetAxis("Horizontal");
float inputY = Input.GetAxis("Vertical");
```

ROTATION
First we store the current Y angle in degrees using eulerAngles.y



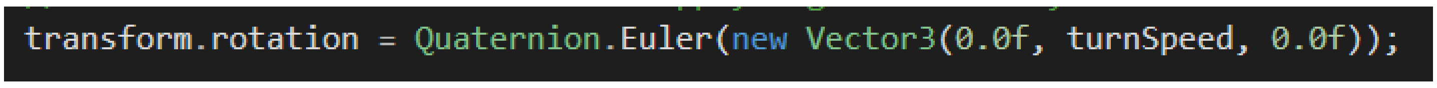
```
float currentYRotation = transform.rotation.eulerAngles.y;
```

Now we calculate the turn speed or degrees per second our character rotates
The y rotation + rotation speed * input X
We multiply by Time.deltaTime to sync with the update



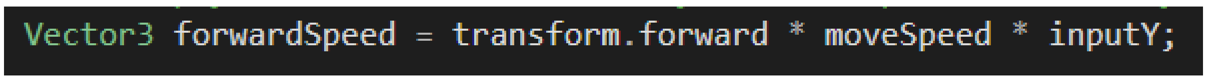
```
float turnSpeed = currentYRotation + rotationSpeed * inputX * Time.deltaTime;
```

We set the final rotation of the transform
Quaternion.Euler will convert a set of degrees (in a Vector3) to a Quaternion (rotation in radians)
Very useful for converting sets of rotations (X,Y,Z) in one go!



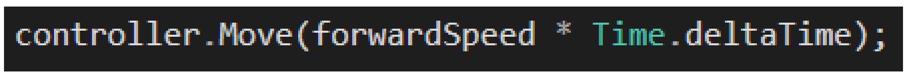
```
transform.rotation = Quaternion.Euler(new Vector3(0.0f, turnSpeed, 0.0f));
```

MOVEMENT
To get a forward facing direction, we can use Transform.forward
To get a move speed, multiply the forward by our move speed field
Multiply by the input Y axis to set the move speed



```
Vector3 forwardSpeed = transform.forward * moveSpeed * inputY;
```

The CharacterController component has a Move method to move a transform over time
We multiply our forward movement by time.deltaTime to sync with the update



```
controller.Move(forwardSpeed * Time.deltaTime);
```

Links

Transform.Rotate
https://docs.unity3d.com/2020.2/Documentation/ScriptReference/Transform.Rotate.html

Transform.RotateAround
https://docs.unity3d.com/2020.2/Documentation/ScriptReference/Transform.RotateAround.html

Camera.WorldToScreenPoint
https://docs.unity3d.com/ScriptReference/Camera.WorldToScreenPoint.html

Mathf.Atan2
https://docs.unity3d.com/2020.2/Documentation/ScriptReference/Mathf.Atan2.html

Mathf.Rad2Deg
https://docs.unity3d.com/2020.2/Documentation/ScriptReference/Mathf.Rad2Deg.html

Mathf.PI
https://docs.unity3d.com/2020.2/Documentation/ScriptReference/Mathf.PI.html

Mathf.Sin
https://docs.unity3d.com/2020.2/Documentation/ScriptReference/Mathf.Sin.html

Input.mousePosition
https://docs.unity3d.com/2020.2/Documentation/ScriptReference/Input-mousePosition.html

Vector3.Normalize()
https://docs.unity3d.com/2020.2/Documentation/ScriptReference/Vector3.Normalize.html

Space.World
https://docs.unity3d.com/2020.2/Documentation/ScriptReference/Space.World.html

Quaternion
https://docs.unity3d.com/2020.2/Documentation/ScriptReference/Quaternion.html

Quaternion.AngleAxis
https://docs.unity3d.com/2020.2/Documentation/ScriptReference/Quaternion.AngleAxis.html

CharacterController
https://docs.unity3d.com/2020.2/Documentation/ScriptReference/CharacterController.html

More info about Quaternions
http://www.euclideanspace.com/maths/algebra/realNormedAlgebra/quaternions/
https://mathworld.wolfram.com/Quaternion.html

GAME DEV. TECH.

UNIVERSITY OF PLYMOUTH