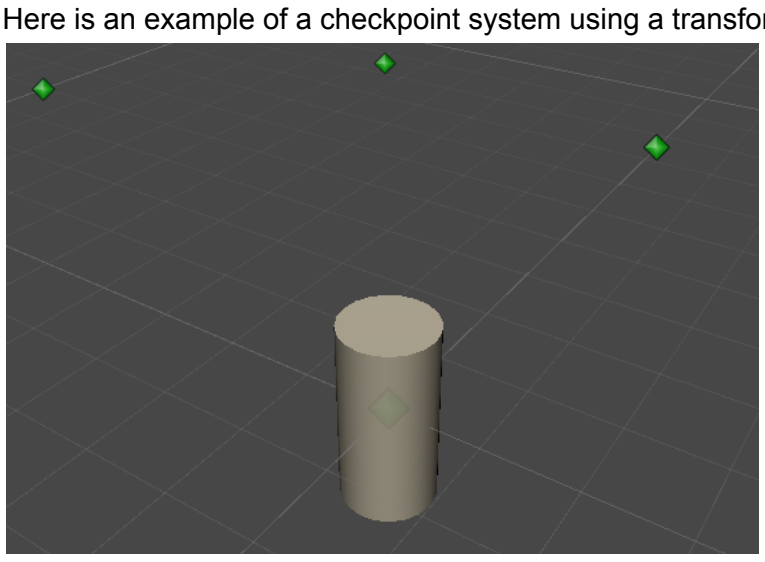
COMP2007 - Game Development

Week 6 - Code session

Checkpoints

A series of transform setup in a hierarchy can be used as an ordered list for many different applications
Here is an example of a checkpoint system using a transform hierarchy



The code example stores the checkpoint current index, then stores the Vector3 position in current point
We will use transforms GetChild method to get a child by index from the checkpoints transform hierarchy

```
// the checkpoint holder
// contains child transforms used for the checkpoints movement
public Transform checkPoints;

// current transform index for the checkpoints
private int currentPointIndex = 0;

// the current position from the checkpoints
private Vector3 currentPoint = Vector3.zero;
```

Start by getting the position of the first checkpoint and apply it to the transform position

```
// set the position at the first checkpoint
transform.position = checkPoints.GetChild(0).position;
```

This example uses MoveTowards to move towards the target over time
NOTE: MoveTowards uses a delta value to move over time instead of a time value

```
// we use the move towards method to move towards the next checkpoint
transform.position = Vector3.MoveTowards(transform.position, currentPoint, Time.deltaTime * speed);
```

Move to the next point if the position matches
NOTE: we are matching **float** values which can be very precise!
This may not work in all situations (like moving fast) and you might have to match within limits (between -0.1 and 0.1 etc)

```
// move to next points if we have arrived at the current position
if (transform.position == currentPoint)
{
    // next point will set the current point to move towards
    NextPoint();
}
```

Coroutines

A coroutine is a method that can pause running at any point and as many times as you wish

- Pre-set sequences of code with unique run times
- A for loop for set numbers of run times
- A while loop to keep running until a condition is met

Coroutine setup

A coroutine requires 3 parts:

1. StartCoroutine to call the method
    a. Coroutines can be stopped by using StopCoroutine or StopAllCoroutines
2. A method returning an IEnumerator
    a. IEnumerator refers to a collection - like an array but more general (hastables, dictionaries, tuples etc)
    b. IEnumerator is a structure to deal with these collections
    c. IEnumerator is part of the **System.Collections** library
3. The method contains a yield statement

Sequence example

In the Start method we use StartCoroutine with a call the our Coroutine method named “ChangeColourSequence”
NOTE: StartCoroutine requires an actual call to the method, with brackets included

```
// 1. change colour sequence
StartCoroutine(ChangeColourSequence());
```

The coroutine method is required to return an IEnumerator
NOTE: the method can have parameters too!

```
// 1. change colour sequence
// a sequence of coroutine "waits"
// can perform linear sequence of codde with custom wait times between each
IEnumerator ChangeColourSequence()
{
```

At least one yield statement is required in a coroutine method
Here we use yield to return a WaitForSeconds
This will pause the method for a set amount of time before running the code after the yield statement

```
// a coroutine "wait" - waits for 1 second
// note: we use a yield to allow the unity engine to "wait" (this is a sub-process on the computers taks manager)
yield return new WaitForSeconds(1);
```

For loop example

Using StartCoroutine to start the method “ChangeColourForLoop”

```
// 2. change colour for loop
StartCoroutine(ChangeColourForLoop());
```

Same setup for the coroutine method - return IEnumerator

```
IEnumerator ChangeColourForLoop()
{
```

We can use a normal for loop containing our yield statement
The Coroutine will actually pause the for loop mid flow
We can do specific numbers of timed activities using this setup

```
// we can use a yield statement within a for loop to perform the "wait" every loop iteration
for (int i = 0; i < 10; i++)
{
    // apply a random colour to the cubes material
    GetComponent<MeshRenderer>().material.color = GetRandomColour();

    // the yield statement here is exactly the same as the previous example, just within a for loop
    yield return new WaitForSeconds(1);
}
```

While loop example

Just like the for loop, we can use a condition based while loop to keep doing a timed activity until the condition is met

```
// 3. change colour while loop
StartCoroutine(ChangeColourWhileLoop());
```

The method as always has to return IEnumerator
The “isOn” boolean is set elsewhere in the example from a key press
The while loop will continue running the yield statements until the condition is met
We can keep doing an activity over any number of times using this method

```
// 3. change colour while loop
// run a coroutine until a condition is met within a while loop
IEnumerator ChangeColourWhileLoop()
{
    // we can use a yield statement within a while loop until the loop condition is met
    while(isOn == true)
    {
        // apply a random colour to the cubes material
        GetComponent<MeshRenderer>().material.color = GetRandomColour();

        // the yield statement here is exactly the same as the previous example
        yield return new WaitForSeconds(1);
    }
}
```

Raycasting & other casting methods

A raycast is an “invisible laser” that can detect any collider intersecting with it
Raycasting is part of the Unity Physics system
NOTE: unlike colliders interacting with each other, a raycast does not require a rigidbody to be present to detect a collider

Other types of “casting” will use a shape instead of a “laser”
A spherecast uses a sphere shape instead of a laser, this will detect any intersecting colliders within its sphere shape
Also box (or cube) casting and capsule casting shapes are available

Raycasting setup

Requires 3 parts:

- A Ray
  - A Vector3 for origin and direction
  - Or a Ray object
- A RaycastHit
  - Stores a collider that was hit with a raycast
  - Stores the exact point of impact in a vector3
  - The distance from the point that fired the raycast to the impact point
  - A “normal” of the direction of the impact point
- A Raycasting method
  - There are 3 types of method
  - Physics.Raycast
    - Will capture a single target
  - Physics.RaycastAll
    - Will capture an array of targets
    - Requires an array of RaycastHit objects
  - Physics.RaycastNonAlloc
    - Will capture up to a pre-set number of targets

Raycast examples setup

We create a Ray to use in all the examples
The Ray has a position (origin) and direction

```
// create the ray for use in all examples
// uses the transform position and forward direction
ray = new Ray
{
    // the origin is the starting point of the ray
    origin = transform.position,

    // the direction is the heading of the laser
    direction = transform.forward
};
```

Single raycast

We need a single RaycastHit field to store any collider we hit

```
// RaycastHit stores a hit gameobject from a raycast
// also stores the distance and the exact point of contact
private RaycastHit hit;
```

Physics.Raycast returns true if a target was hit - we can easily use this in an if statement

```
// single raycast
if (Physics.Raycast(ray, out hit))
{
```

Parameters

- ray
  - the Ray object we created earlier
- out hit
  - “out” is a **parameter modifier,** it passes a value (field or variable) by reference instead of copying it
  - In other words it “fills up” an existing object we created elsewhere
  - In this example we “fill up” the “hit” field (the RaycastHit field)

Multiple target Raycast

We can store any number of targets by using an array of RaycastHit objects
NOTE: will only work with a standard array, not a list
NOTE: this is considered a little resource intensive if you are doing a lot of raycasting, consider using allocated instead if possible

```
// an array of hits, used in the multiple examples
private RaycastHit[] hits;
```

Physics.RaycastAll returns an array of RaycastHits
We can assign our hits array to the return value

```
// multiple raycast
// returns an array of Raycast hit objects
hits = Physics.RaycastAll(ray);
```

The hits array can be any size, which can cause performance problems if you are doing a lot of raycasting (especially inside an update method)
Having an array of a specified size will help stabilise problems due to using a set amount of memory (or slots in the array)
The allocated raycast uses this strategy

Allocated Raycast

To use allocated raycasting, create an array of a set size defined in its constructor
Here we use an array of 2 hits, allocated raycasting will only provide 2 targets maximum

```
// an array of hits with a pre-set size
// used in the alloc examples
private RaycastHit[] allocatedHits = new RaycastHit[2];
```

Physics.RaycastNonAlloc returns the number of targets

```
int numberOfHits = Physics.RaycastNonAlloc(ray, allocatedHits, 7);
```

Parameters

- ray
  - the Ray object we created earlier
- allocatedHits
  - Our RaycastHits array of specific size
- 7
  - The distance the ray will travel

Check the number of hits, then loop through the allocated array
NOTE: you may need to check for null objects in the array in case not all the targets have been allocated!

```
if (numberOfHits > 0)
{
    for (int i = 0; i < numberOfHits; i++)
    {
        print(allocatedHits[i].transform);
    }
}
```

Spherecasting

The sphere casting examples follow the same method as raycasting, they require:

- A Ray
  - All types of casting use the Ray & RaycastHit objects
- A RaycastHit
- A “casting” method
  - We use Spherecast, Boxcast or Capsulecast instead of Raycast

Single Spherecast

A single Sperecast returns a bool, so we can use it in a conditional

```
if(Physics.SphereCast(ray, 0.5f, out hit))
```

Parameters

- ray
  - the Ray object we created earlier
- 0.5f
  - The radius of our sphere
- out hit
  - The RaycastHit object as a reference

Multiple Spherecast

As before, we supply an array of RaycastHits

```
// an array of hits, used in the multiple examples
private RaycastHit[] hits;
```

Physics.SphereCastAll will return an array of ALL the hits - the array size may vary considerably!

```
hits = Physics.SphereCastAll(ray, 0.5f, 100);
```

Parameters

- ray
  - the Ray object we created earlier
- 0.5f
  - The radius of our sphere
- 100
  - The distance our sphere will travel
  - The shape will intersect, adding more hits like a sphere collider moving through a scene

Allocated Spherecast

A more performance-friendly version of multiple Spherecasting
We use the allocated array - an array that has a set number of items in the constructor

```
// an array of hits with a pre-set size
// used in the alloc examples
private RaycastHit[] allocatedHits = new RaycastHit[2];
```

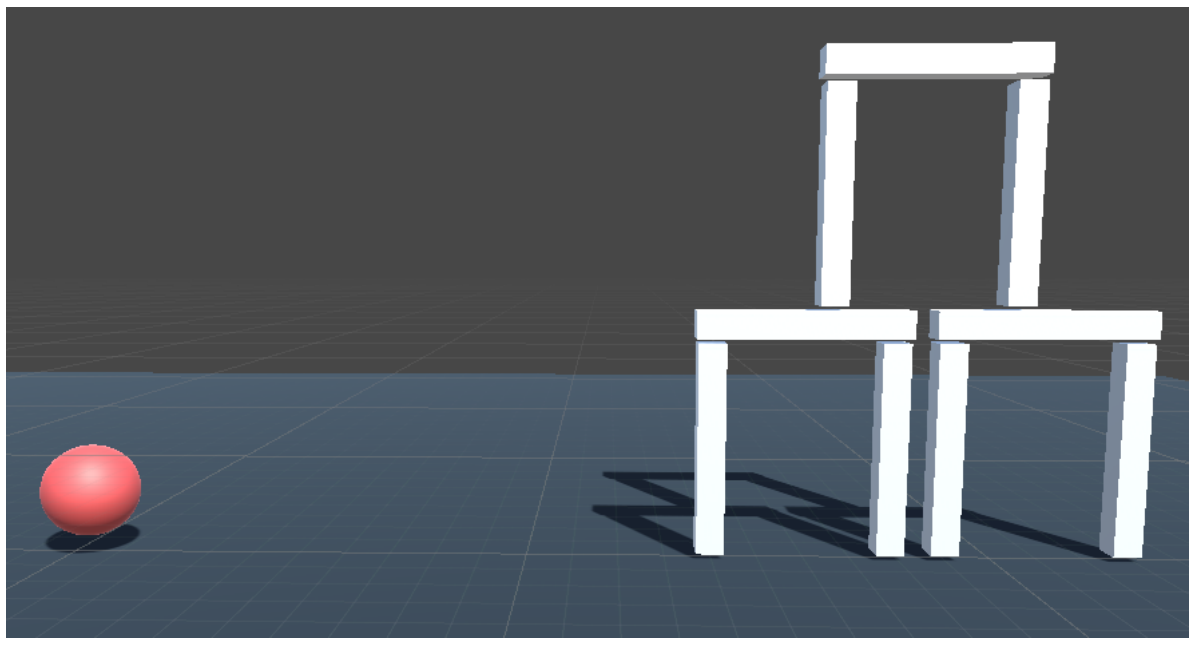
Physics.SphereCastNonAlloc returns a set number of hits

```
int numberOfHits = Physics.SphereCastNonAlloc(ray, 0.5f, allocatedHits, 100);
```

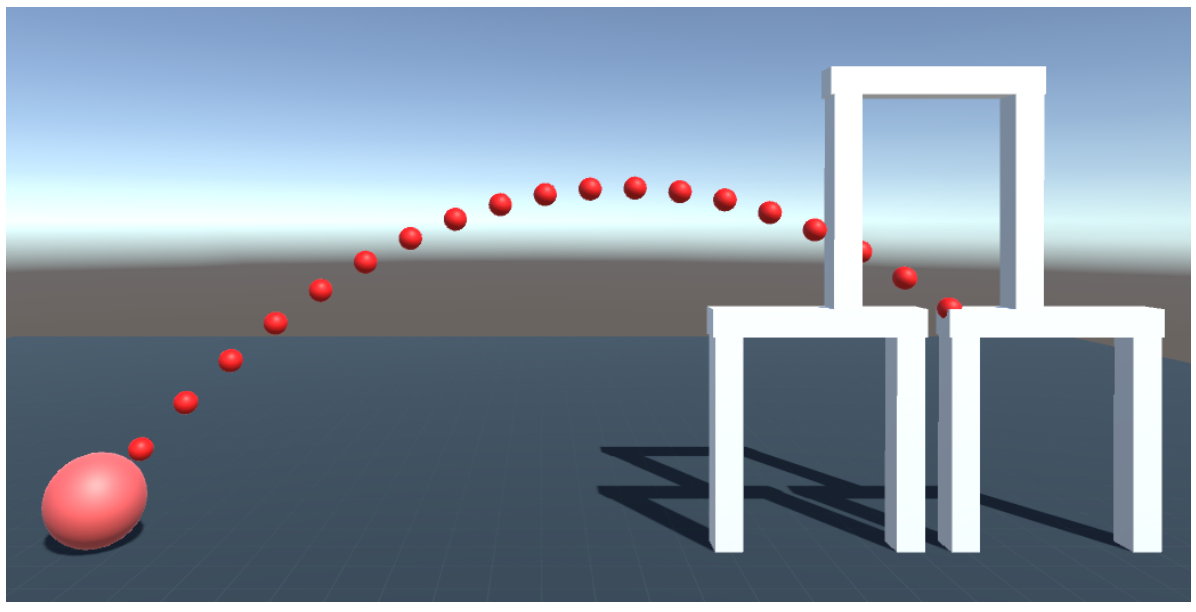
Parameters

- ray
  - the Ray object we created earlier
- 0.5f
  - The radius of our sphere
- allocatedHits
  - Our RaycastHits array of specific size
- 100
  - The distance our sphere will travel
  - The shape will intersect, adding more hits like a sphere collider moving through a scene

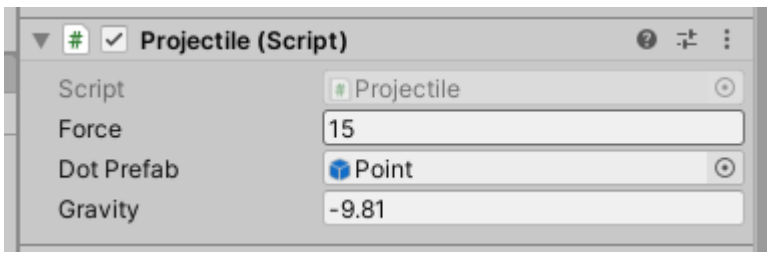
Angry birds 3D



The projectile component hurls the ball in a direction
The trajectory is predicted before releasing the projectile



The force is used to calculate the trajectory
The dot prefabs form the trajectory
The gravity is custom for this example, but you can add unity gravity using Physics.gravity



We get a direction from the screen mouse position and transform

```
// get the position of the transform on the screen
Vector3 screenPos = mainCam.WorldToScreenPoint(transform.position);

// reset the z axis so our trajectory is in 2D (X and Y axis)
screenPos.z = 0;

// calculate the facing direction from the mouse cursor to the transform
// normalise the direction so we can use it elsewhere
direction = (Input.mousePosition - screenPos).normalized;
```

We calculate the velocity by multiplying the direction by the force field

```
Vector2 velocity = new Vector2
{
    x = direction.x * force,
    y = direction.y * force
};
```

The spacing is the space in between each point
We calculate the X position from the transform and velocity and spacing

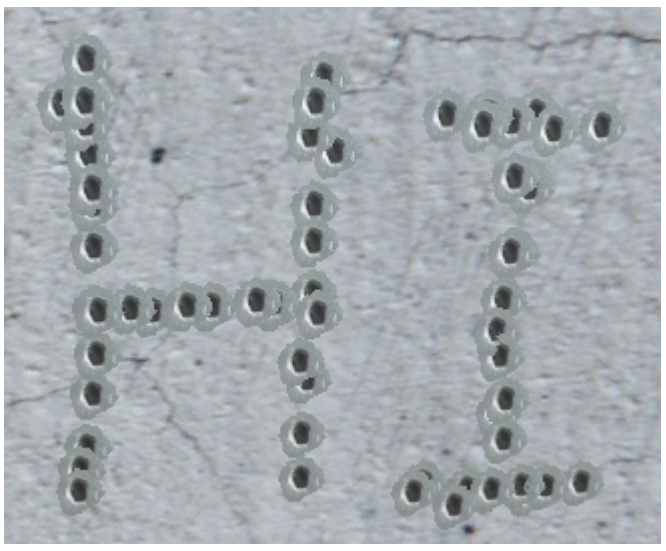
```
// calculate the spacing between the dots
float spacing = i * 0.1f;

// x position is the velocity * spacing, evenly spacing the dots out
float x = transform.position.x + (velocity.x * spacing);
```

The Y position is the transform, velocity and spacing
Gravity is calculated as gravity multiplied by spacing to the power of 2 and halved
The final Y position is the position minus the gravity

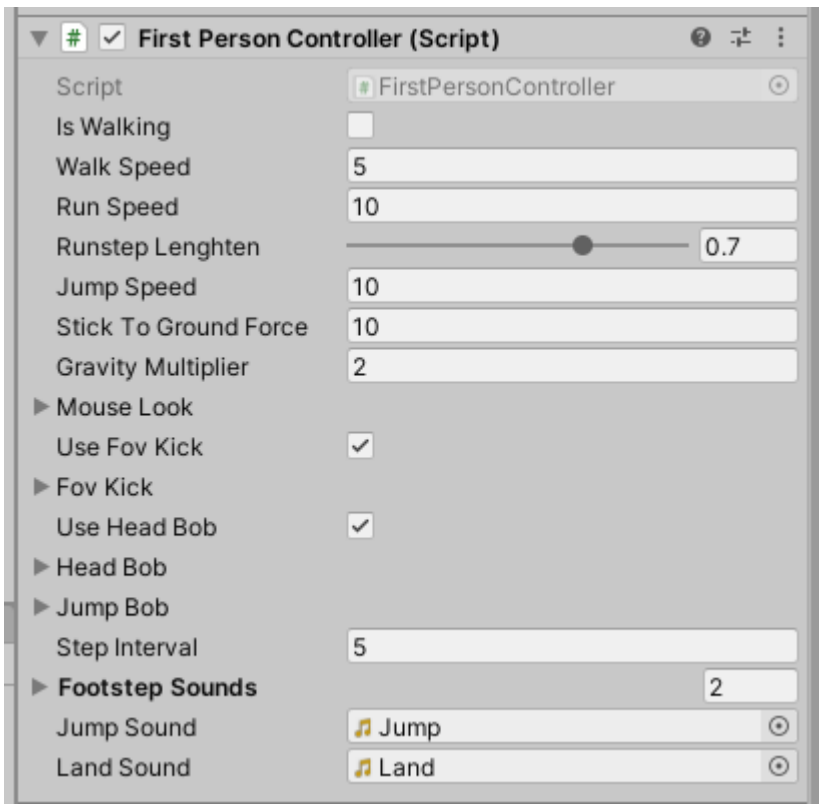
```
// calculate the y position using spacing
float yPosition = transform.position.y + (velocity.y * spacing);

// calculate the y gravity
float yGravity = ((-gravity * spacing) * spacing) / 2.0f;

// minus the y position from the provided gravity field
float y = yPosition - yGravity;
```

Bullet Holes

Using raycasting, we can place a mesh upon a surface, aligning it to the face exactly

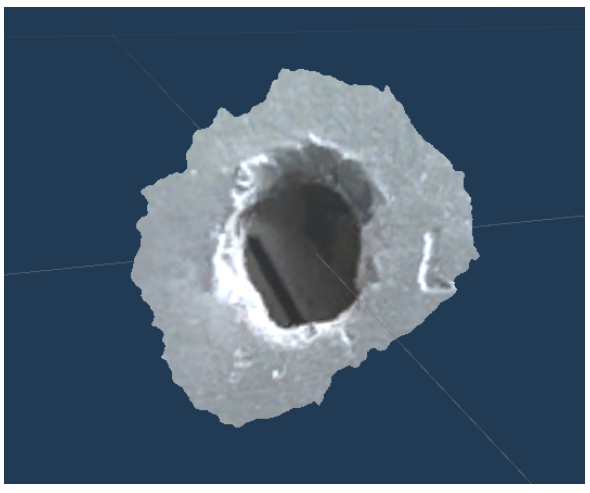


This example uses the FirstPersonController script supplied as part of the Standard Assets pack from the Unity Asset store

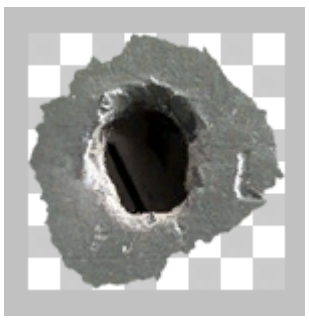


The example bullet holes are prefabs containing a flat mesh with a bullet hole texture
NOTE: the material uses the Cutout setting to remove the edges from the image

Bullet hole mesh



Bullet hole image



SniperGun script code

We create a Ray object using the centre of the main Camera.
ViewportPointToRay will take a Vector3 position from the viewport (Game view in the editor or in an executable), we are using the centre of the screen
ViewportPointToRay returns a ray that can be used in the scene

```
// ViewportPointToRay creates a ray using the camera position and forward direction
ray = Camera.main.ViewportPointToRay(new Vector3(0.5F, 0.5F, 0));
```

We perform a single raycast, this returns true if something was hit
The hit is stored in the “hit” field as a reference (using the out modifier)
raycastDistance is the maximum distance to “fire” the Ray from the origin point

```
// use the ray with the ray distance field in a raycast
// the first object hit is stored in the hit field
if (Physics.Raycast(ray, out hit, rayDistance))
```

We store the point of impact, this is the hit.point property of the RaycastHit object

```
// raycast hit has a point field for the impact point
Vector3 impactPoint = hit.point;
```

FromToRotation will convert a direction to a rotation along an axis
We will use this to rotate the bullet hole to face away from the surface we hit

```
Quaternion facingRotation = Quaternion.FromToRotation(Vector3.forward, -hit.normal);
```

We are using Vector3.forward as the axis and -hit.normal as the direction
Vector3.forward is the WORLD forward facing direction - transform.forward is the facing direction local to a particular transform
hit.normal is the direction facing AWAY from the surface - if we minus it (-hit.normal) the rotation will face the opposite way along the Vector3.forward axis

When the bullet hole is spawned, facingRotation will be a rotation facing the SAME WAY as the surface
NOTE: we are facing the same way as the SURFACE we hit, not the object itself!

```
// spawn a bullet hole at the exact point of collision
Instantiate(bulletHolePrefab, impactPoint, facingRotation);
```

Links

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

StartCoroutine
https://docs.unity3d.com/2020.2/Documentation/ScriptReference/MonoBehaviour.StartCoroutine.html

StopCoroutine
https://docs.unity3d.com/2020.2/Documentation/ScriptReference/MonoBehaviour.StopCoroutine.html

IEnumerator
https://docs.microsoft.com/en-us/dotnet/api/system.collections.ienumerator?view=net-5.0

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

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

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

Debug.DrawRay
https://docs.unity3d.com/2020.2/Documentation/ScriptReference/Debug.DrawRay.html

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

Camera.main
https://docs.unity3d.com/2020.2/Documentation/ScriptReference/Camera-main.html

Camera.WorldToScreenPoint
https://docs.unity3d.com/2020.2/Documentation/ScriptReference/Camera.WorldToScreenPoint.html

Camera.ViewportPointToRay
https://docs.unity3d.com/2020.2/Documentation/ScriptReference/Camera.ViewportPointToRay.html

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

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

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

Physics.Raycast
https://docs.unity3d.com/2020.2/Documentation/ScriptReference/Physics.Raycast.html

Physics.RaycastAll
https://docs.unity3d.com/2020.2/Documentation/ScriptReference/Physics.RaycastAll.html

Physics.RaycastNonAlloc
https://docs.unity3d.com/2020.2/Documentation/ScriptReference/Physics.RaycastNonAlloc.html

Physics.SphereCast
https://docs.unity3d.com/2020.2/Documentation/ScriptReference/Physics.SphereCast.html

Physics.SphereCastAll
https://docs.unity3d.com/2020.2/Documentation/ScriptReference/Physics.SphereCastAll.html

Physics.SphereCastNonAlloc
https://docs.unity3d.com/2020.2/Documentation/ScriptReference/Physics.SphereCastNonAlloc.html

GAME DEV. TECH.

UNIVERSITY OF PLYMOUTH