CT3536 (Unity3D)
Section 4

Physics in more detail

- Rigidbody
- Collider
- Collision
- ContactPoint
- PhysicMaterial
- Trigger
- (Later: raycasting)

Rigidbody

- https://docs.unity3d.com/Manual/RigidbodiesOverview.html
- https://docs.unity3d.com/ScriptReference/Rigidbody.html
- A Rigidbody is the main component that enables physical behaviour for a GameObject; it puts the GameObject under control of the physics engine
- With a Rigidbody component attached, the object will respond to gravity (unless its useGravity field is set to false)
- If one or more Collider components are also added, the GameObject is affected by incoming collisions (from other Colliders) according to shape, mass, momentum, linear and angular velocities, as well as Physics Materials which define bounciness and friction.
- Since a Rigidbody component takes over the movement of the GameObject it is attached to, you (normally) shouldn’t try to move it from a script by changing the Transform properties such as position and rotation. Instead, you should apply forces to push the GameObject and let the physics engine calculate the results.

Rigidbody

- When a Rigidbody is moving slower than a defined minimum linear or rotational speed, the physics engine assumes it has come to a halt. When this happens, the GameObject does not move again until it receives a collision or force, and so it is set to “**sleeping**” mode.
- This optimisation means that no processor time is spent updating the Rigidbody until the next time it is “awoken” (that is, set in motion again).
- Sleeping can also be useful as it removes small 'jitters' that may happen due to inaccuracies in the physics simulation
- In your scripts which apply physics forces to Rigidbodies, use the FixedUpdate( ) method rather than the Update( ) method, since FixedUpdate is synced with the physics simulation updates

Rigidbody: Properties (read/write)

- public float drag
- public float angularDrag

- public float mass

- public Vector3 velocity
- public Vector3 angularVelocity

- public Vector3 centerOfMass - (offset from Transform centre)

Rigidbody: Methods

- public void AddForce(Vector3 force) - force is a vector in world coords, applied through centre of mass of the Rigidbody
- public void AddRelativeForce(Vector3 force) - force is in local coords

- public void AddForceAtPosition(Vector3 force, Vector3 position)
  - position is also world coords, and allows the force to apply torque

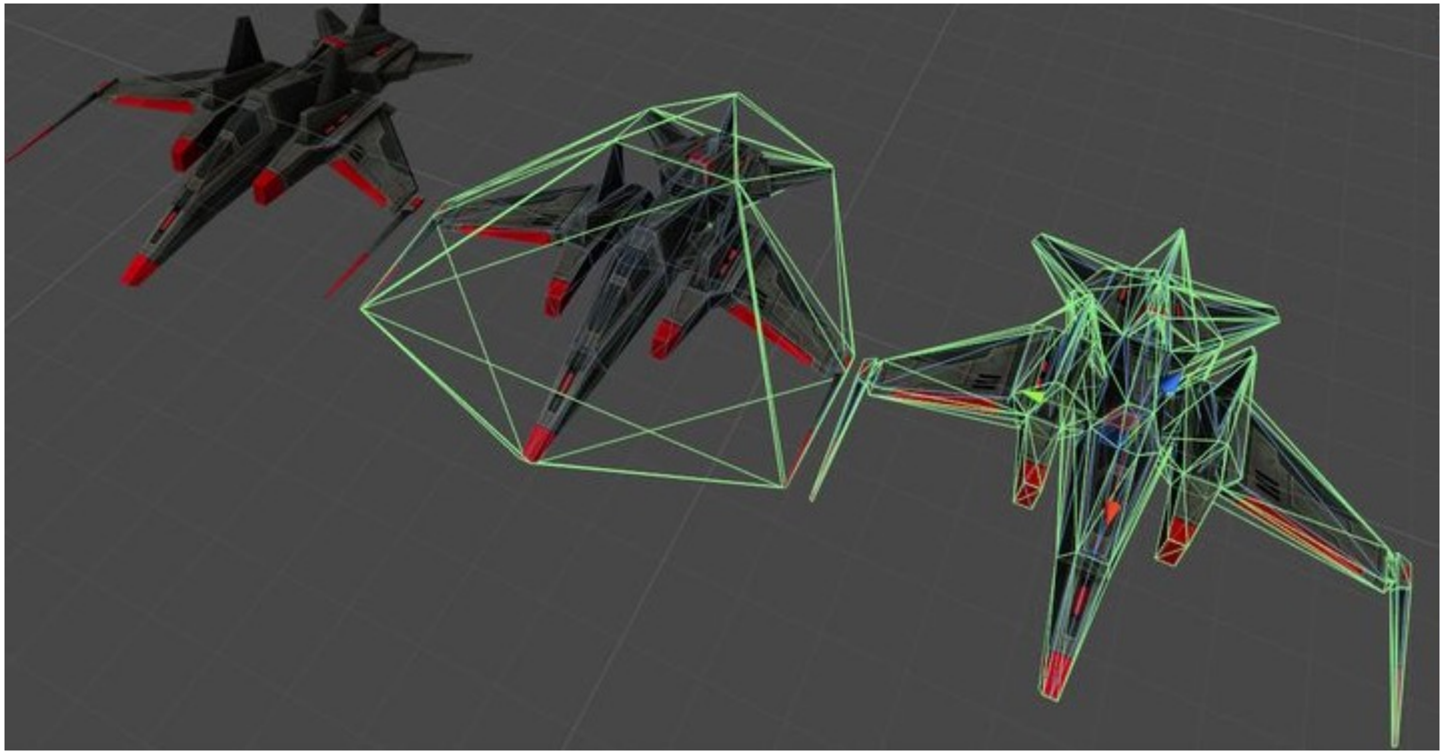
- public void AddTorque(Vector3 torque)
- public void AddRelativeTorque(Vector3 torque)

- public void MovePosition(Vector3 position) - (often) the correct way to move it, i.e. rather than resetting its Transform.position.. This will check for intervening collisions rather than “teleporting”
- public void MoveRotation(Quaternion rot)

Colliders

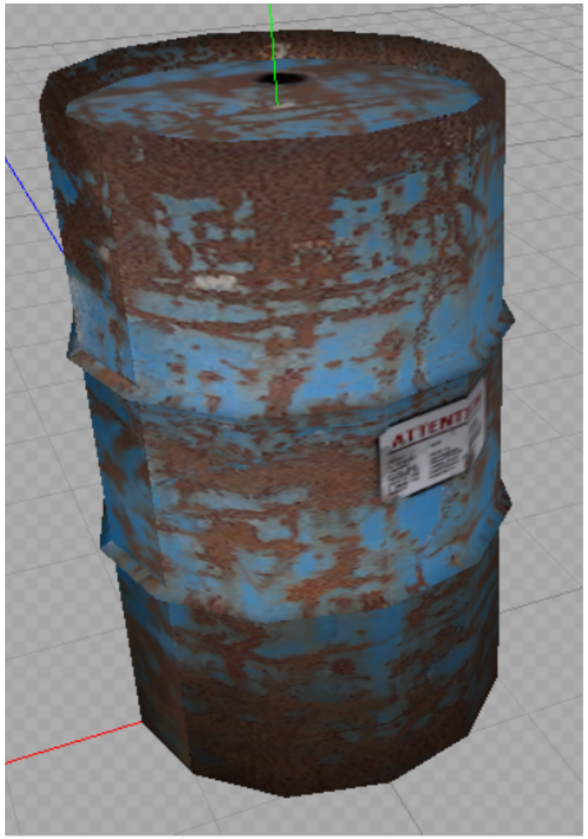
- https://docs.unity3d.com/Manual/CollidersOverview.html
- Collider components define the shape of an object for the purposes of physical collisions. A collider (invisible at runtime) need not be the exact same shape as the object's mesh and in fact, a rough approximation is often more efficient and indistinguishable in gameplay
- The simplest (and least processor-intensive) colliders are the so-called **primitive** collider types. In 3D, these are the BoxCollider, SphereCollider and CapsuleCollider. In 2D, you can use BoxCollider2D and CircleCollider2D.
- Any number of these can be added to a single object to create composite colliders to reasonably approximate a 3D model.
- If you need more accuracy (at increased processor cost), use MeshCollider which accurately matches the 3D graphical model (= polygonal mesh)
- A MeshCollider will be unable to collide with another MeshCollider unless you mark it as Convex in the inspector. This will generate the collider shape as a "convex hull" which is like the original mesh with concavities filled in.
- The general rule is to use mesh colliders for static scene geometry (walls, ground, etc.) and to approximate the shape of moving objects using composite primitive colliders

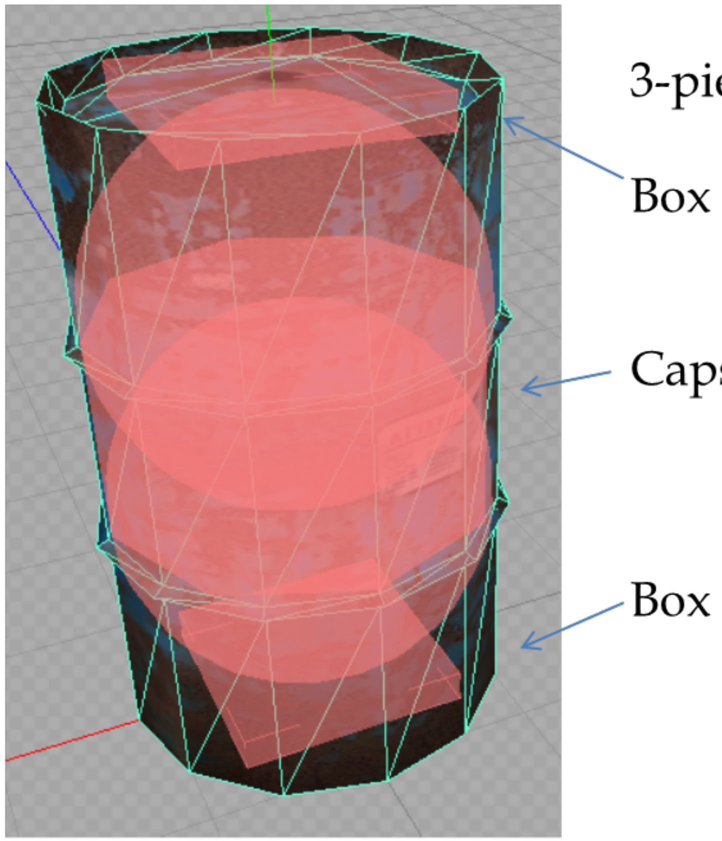
MeshCollider



- Convex Hull versus Non-Convex

Composite Colliders





Collision Messages

Any object with a Collider receives messages from the physics engine when it collides with other Colliders. Any script on the object may choose to respond, by implementing these methods:

```
void OnCollisionEnter(Collision collision) {
      // the Collision object contains information about contact points, impact
velocity etc. (see next slide)
}

void OnCollisionExit(Collision collision) {
}

void OnCollisionStay(Collision collision) {
}
```

NB if you get these method signatures wrong, your code will compile but the methods will not get called when you expect..

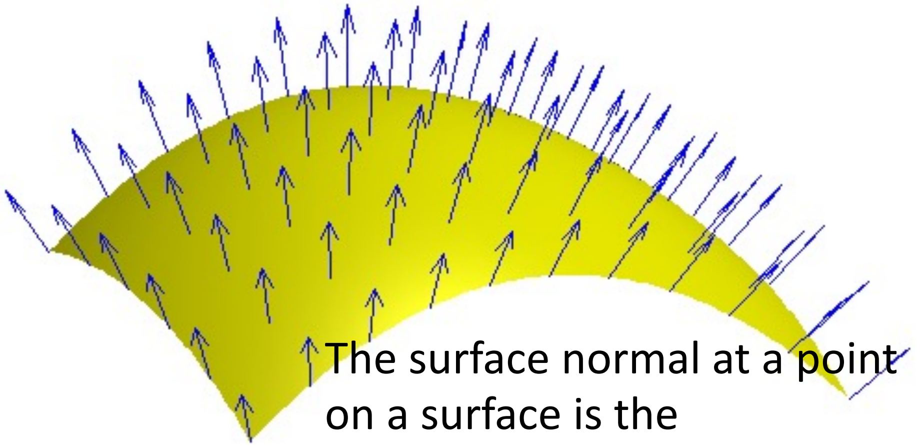
Collision class

- https://docs.unity3d.com/ScriptReference/Collision.html
- The OnCollision methods receive objects of this type, containing useful information
- Member data:
  - **collider** The Collider object that we hit (Collider is the base class of BoxCollider, SphereCollider, CapsuleCollider, MeshCollider)
  - **contacts** The contact point(s) generated by the physics engine, as an array of ContactPoint structs (see next slide)
  - **gameObject** The GameObject whose collider we are colliding with.
  - **relativeVelocity** The relative linear velocity of the two colliding objects (as a Vector3)
  - **rigidbody** The Rigidbody we hit. This is null if the object we hit has a collider but has no rigidbody.

ContactPoint struct

- https://docs.unity3d.com/ScriptReference/ContactPoint.html
- Collision objects contains arrays of ContactPoint structs, in their .contacts member
- Member data of ContactPoint:
  - **point** The point of contact (Vector3 world coords).
  - **normal** Surface Normal at the contact point (=Vector3 world coords)
  - **otherCollider** The other Collider in contact at the point.
  - **thisCollider** The first collider in contact at the point (useful if a GameObject has more than one collider and we need to know which one)

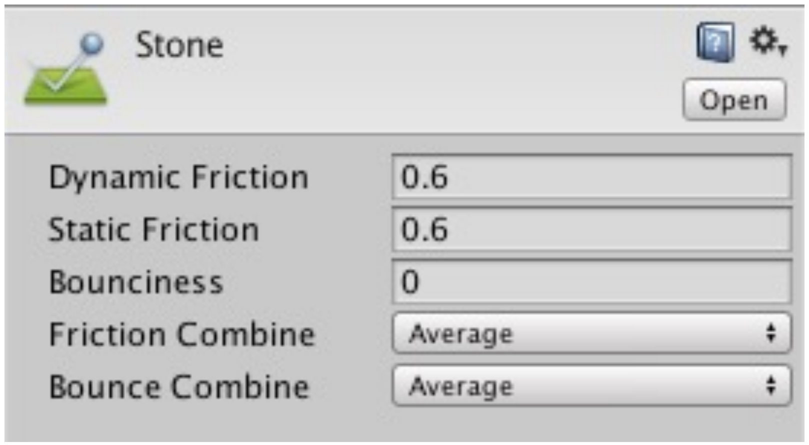
- These data allow us to operate on the object that hit, and also to create accurate special effects etc. at the point of contact, rotated according to the surface normal at that point



The surface normal at a point on a surface is the (normalised) vector perpendicular to the surface

PhysicMaterial (sic.)

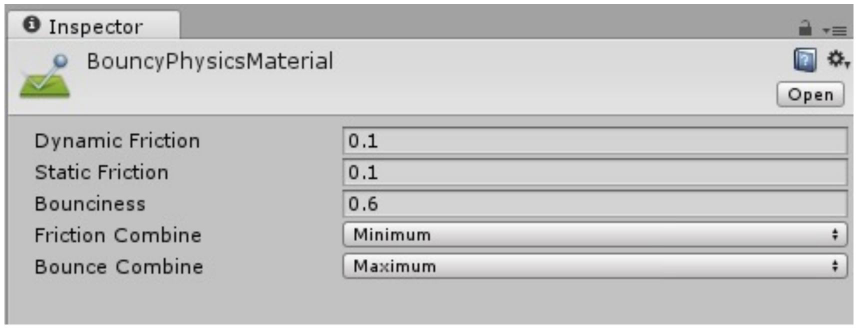
- When colliders interact, their surfaces need to simulate the properties of the material they are supposed to represent.
- For example, a sheet of ice will be slippery while a rubber ball will offer a lot of friction and be very bouncy.
- Although the shape of colliders is not deformed during collisions (hence the term "rigid body physics"), their friction and bounce can be configured using Physics Materials (the asset type to create is "PhysicMaterial")



Dynamic Friction	0.6
Static Friction	0.6
Bounciness	0
Friction Combine	Average
Bounce Combine	Average

Example

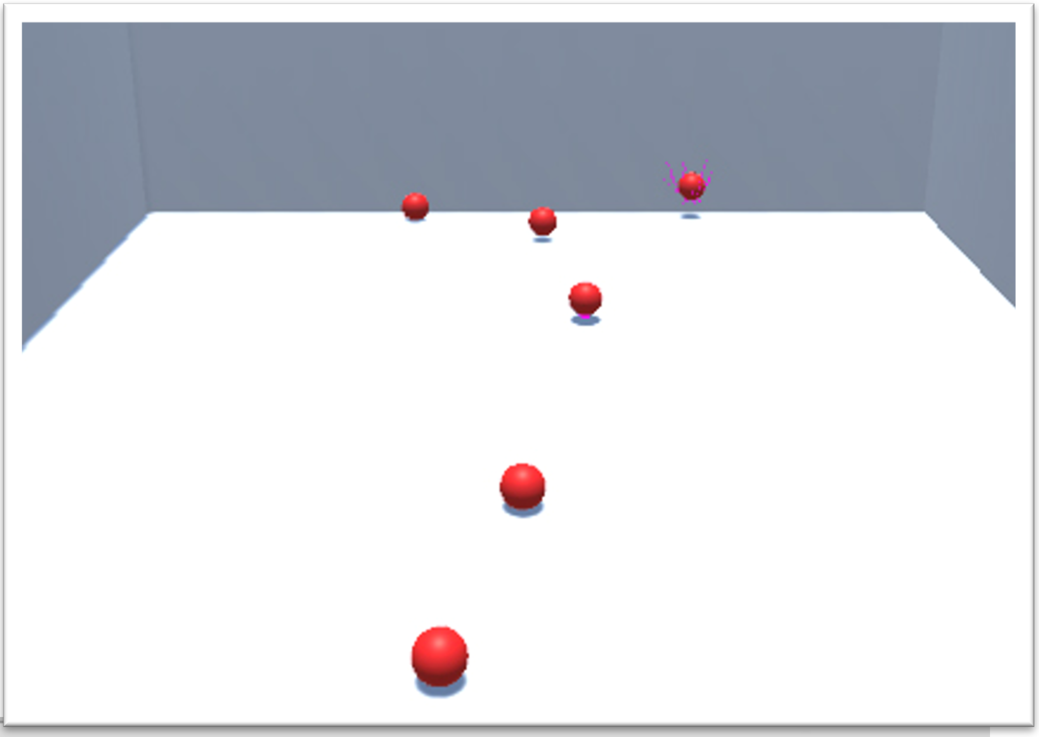
- Throwing balls with the mouse, and spawning a spark (cone-shaped) emitter at collision point when a wall/floor is hit, with the emitter oriented outwards using collision normal
- We'll see emitters in detail a bit later
- You can download the **Week4LectureExamples project** from Blackboard

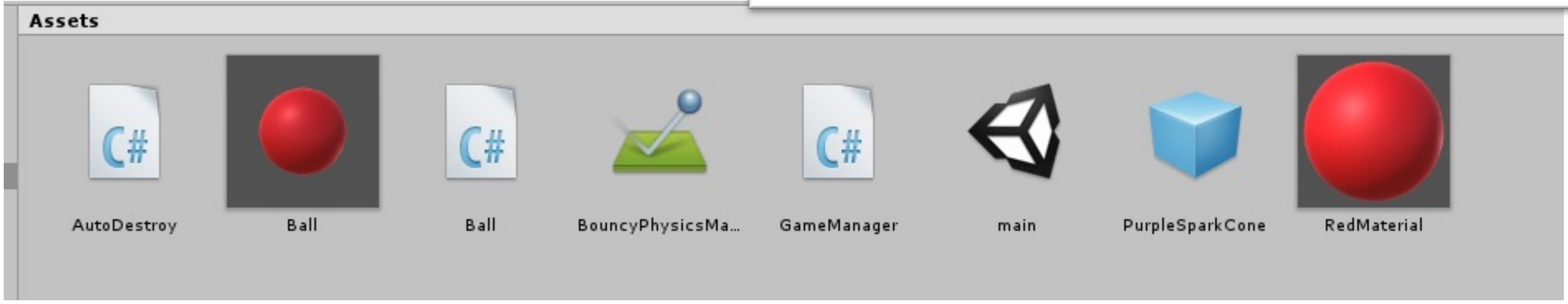


Inspector

BouncyPhysicsMaterial

Dynamic Friction	0.1
Static Friction	0.1
Bounciness	0.6
Friction Combine	Minimum
Bounce Combine	Maximum





Assets

```
public class GameManager : MonoBehaviour {

    // inspector settings
    public GameObject ballPrefab;
    public GameObject sparkEmitterPrefab;
    //

    public static GameManager instance;

    void Start () {
        instance = this;
        Camera.main.transform.position = new Vector3(-30f,15f,0f);
        Camera.main.transform.LookAt(new Vector3(0f,5f,0f));
    }

    void Update() {
        // mouse button has just been pressed
        if (Input.GetMouseButtonDown(0)) {
            Vector3 mousePosOnScreen = Input.mousePosition; // 2d position on screen (pixels)
            // 15 units in front of camera
            mousePosOnScreen.z = 15f;
            Vector3 mousePosInWorld = Camera.main.ScreenToWorldPoint(mousePosOnScreen);

            // spawn a new ball and give it some velocity
            GameObject go = Instantiate(ballPrefab);
            go.transform.position = mousePosInWorld;
            Vector3 dir = (mousePosInWorld-Camera.main.transform.position).normalized;
            go.GetComponent<Rigidbody>().velocity = dir * 40f;

        }
    }

}
```

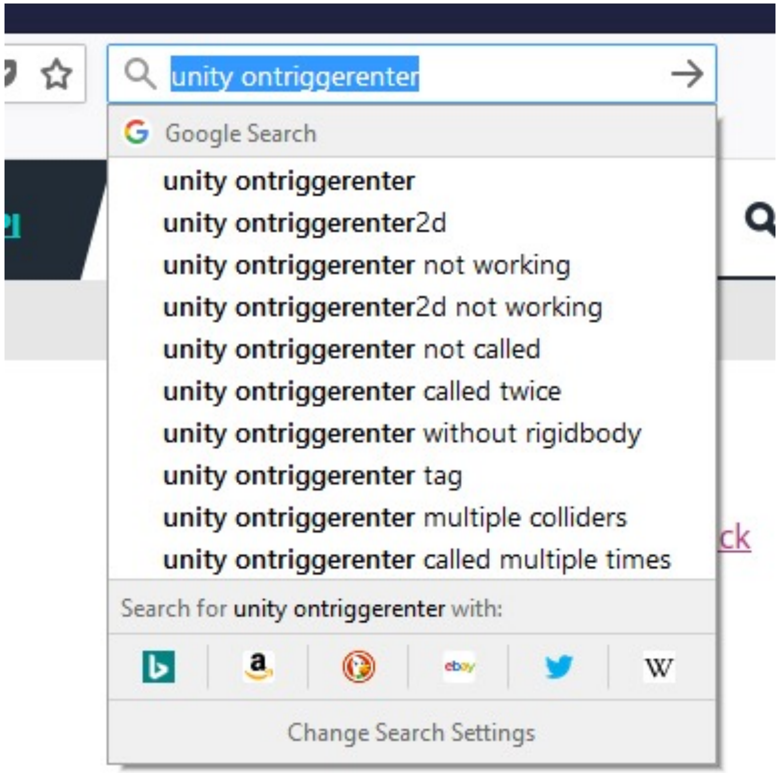
```
public class Ball : MonoBehaviour {

    // inspector settings
    public Rigidbody rigid;
    //

    void Update() {
        // remove ball if it has stopped moving or fallen off the platform
        if (rigid.velocity.magnitude<0.01f || transform.position.y<-10f)
            Destroy(this.gameObject);
    }

    void OnCollisionEnter(Collision collision) {
        // spawn a spark emitter at each point of contact,
        // and orient it so that the sparks emit outwards from the surface hit
        for (int i=0; i<collision.contacts.Length; i++) {
            ContactPoint cp = collision.contacts[i];
            GameObject go = Instantiate(GameManager.instance.sparkEmitterPrefab);
            go.transform.position = cp.point;
            go.transform.LookAt(cp.point+cp.normal);
        }
    }

}
```

This component is attached to the PurpleSparkCone prefab, so that instances look after their own destruction after their lifetime has elapsed

```
public class AutoDestroy : MonoBehaviour {

    // inspector settings
    public float lifetime = 2f;
    //

    void Awake() {
        StartCoroutine( ProcessLifetime() );
    }

    private IEnumerator ProcessLifetime() {
        yield return new WaitForSeconds(lifetime);
        Destroy(this.gameObject);
    }

}
```

Triggers

- As discussed above, the scripting system can detect when collisions occur and initiate actions using the OnCollisionEnter/Exit/Stay methods.
- However, you can also use the physics engine simply to detect when one collider enters the space of another without creating a physical collision response.
- A collider configured as a Trigger (using the isTrigger property) does not behave as a solid object and will simply allow other colliders to pass through.
- When a collider enters its space, a trigger will call the OnTriggerEnter/Exit/Stay methods on the trigger object's scripts.


unity ontriggerenter

Google Search

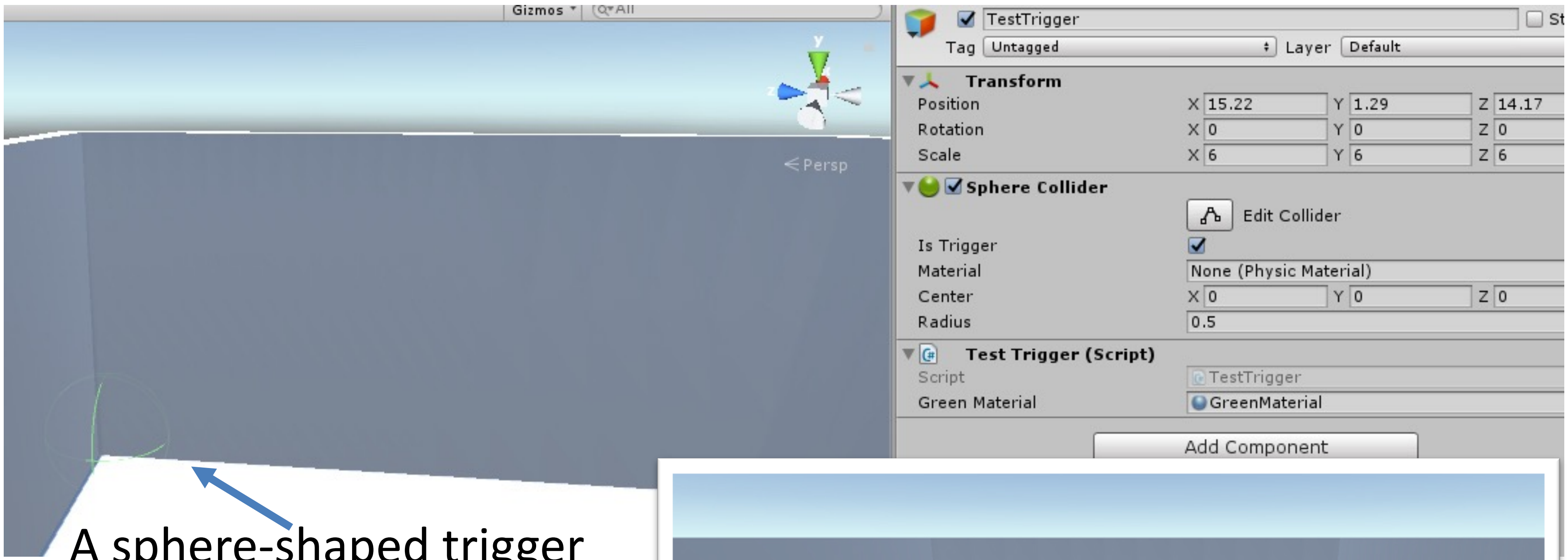
unity ontriggerenter
unity ontriggerenter2d
unity ontriggerenter not working
unity ontriggerenter2d not working
unity ontriggerenter not called
unity ontriggerenter called twice
unity ontriggerenter without rigidbody
unity ontriggerenter tag
unity ontriggerenter multiple colliders
unity ontriggerenter called multiple times

Search for unity ontriggerenter with:

Change Search Settings

Among the favourite Google search terms for any Unity topic seems to be "X not working" ☺

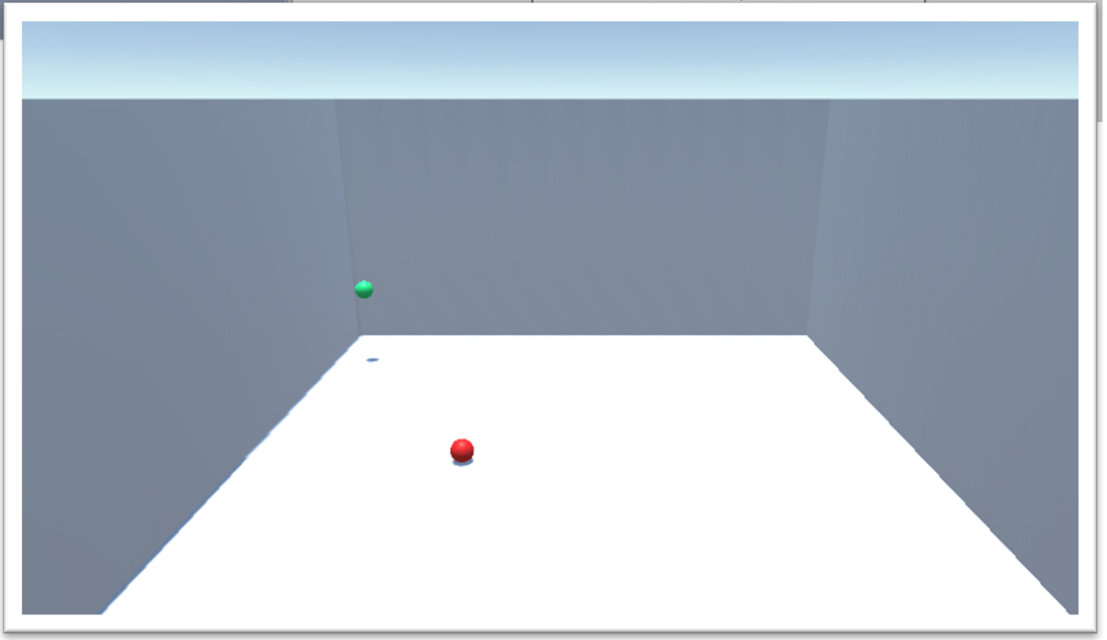
Example



```
TestTrigger                                   St
Tag  Untagged              Layer  Default

▼ Transform
Position      X 15.22      Y 1.29      Z 14.17
Rotation      X 0          Y 0         Z 0
Scale         X 6          Y 6         Z 6

▼ ✓ Sphere Collider
                    Edit Collider
Is Trigger     ✓
Material       None (Physic Material)
Center        X 0          Y 0         Z 0
Radius         0.5

▼ Test Trigger (Script)
Script         TestTrigger
Green Material GreenMaterial

              Add Component
```

A sphere-shaped trigger which turns balls green! (expanded on previous example)

The trigger's game object itself is invisible (no renderer)



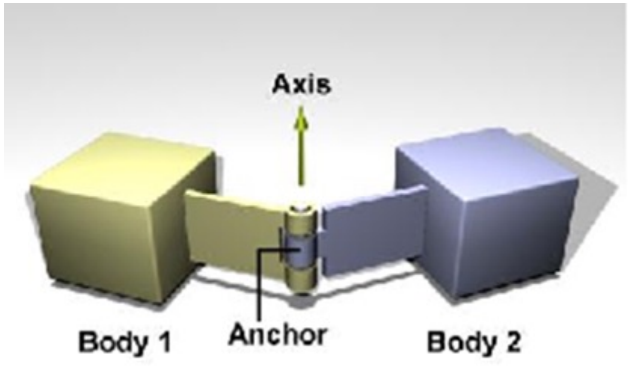
```
public class TestTrigger : MonoBehaviour {

  // inspector settings
  public Material greenMaterial;
  //

  void OnTriggerEnter(Collider other) {
    // only continue if what hit us was a Ball
    if (other.gameObject.GetComponent<Ball>()!=null) {
      other.gameObject.GetComponent<Renderer>().material = greenMaterial;
    }
  }
}
```

Renderer is the base class of all types of renderer (MeshRenderer, BillboardRenderer, LineRenderer, SpriteRenderer, etc.)

Physics Joints

- Unity provides a number of types of joints to constrain the motion of rigid bodies relative to each other, e.g. to simulate chains, ropes, swings, car+trailer, etc.
- E.g.:
  - The Hinge Joint groups together two Rigidbodies, constraining them to move like they are connected by a hinge. It is perfect for doors, but can also be used to model chains, pendulums, etc.
  - The Spring Joint joins two Rigidbodies together but allows the distance between them to change as though they were connected by a spring.

https://docs.unity3d.com/Manual/Joints.html



Hinge Joint

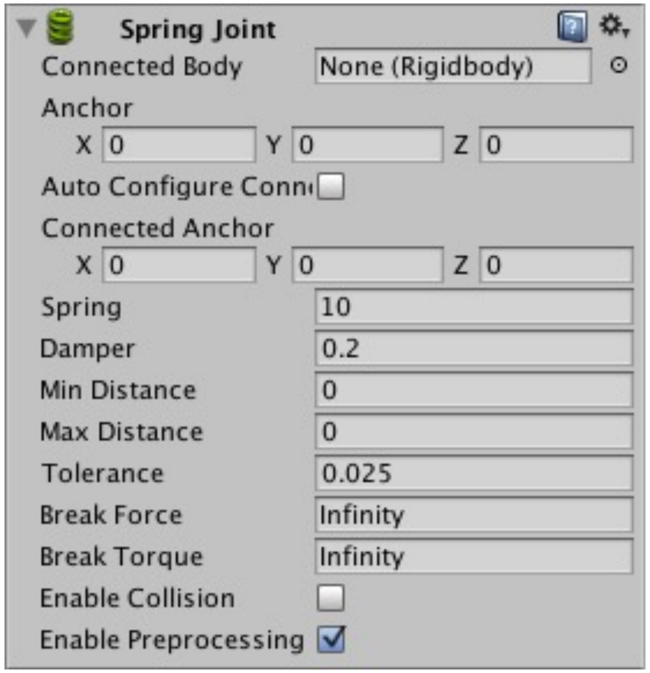
SpringJoint

The spring acts like a piece of elastic that tries to pull the two anchor points together to the exact same position.

The strength of the pull is proportional to the current distance between the points with the force per unit of distance set by the *Spring* property.

To prevent the spring from oscillating endlessly you can set a *Damper* value that reduces the spring force in proportion to the relative speed between the two objects. The higher the value, the more quickly the oscillation will die down.

You can set the anchor points manually but if you enable *Auto Configure Connected Anchor*, Unity will set the connected anchor so as to maintain the initial distance between them (i.e., the distance you set in the scene view while positioning the objects).

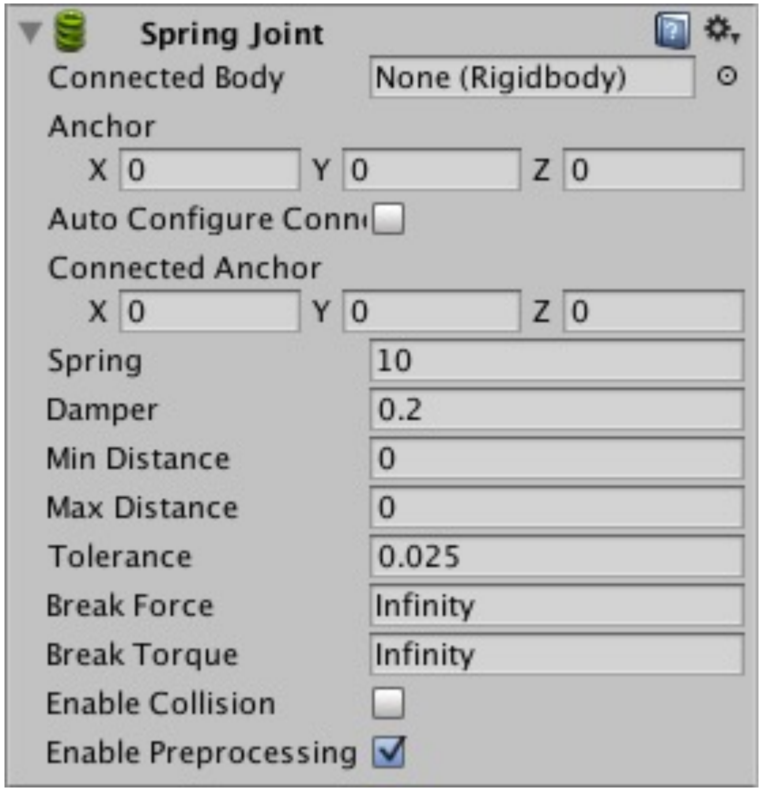


Spring Joint

Connected Body	None (Rigidbody)
Anchor	X 0 Y 0 Z 0
Auto Configure Conn	☐
Connected Anchor	X 0 Y 0 Z 0
Spring	10
Damper	0.2
Min Distance	0
Max Distance	0
Tolerance	0.025
Break Force	Infinity
Break Torque	Infinity
Enable Collision	☐
Enable Preprocessing	✓

SpringJoint

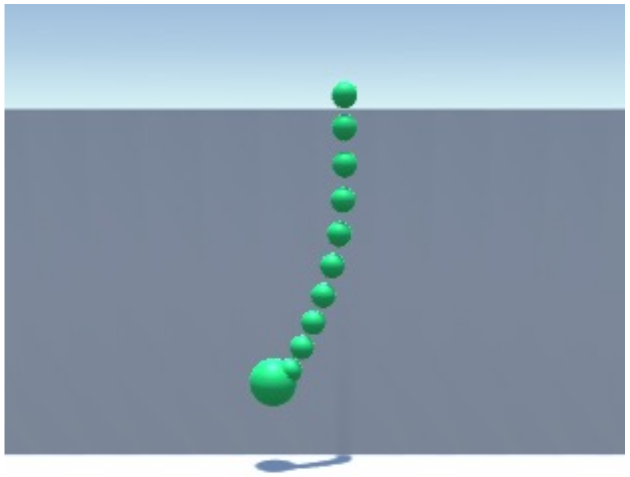
- **Connected Body**: The Rigidbody object that the object with the spring **joint** is connected to. If no object is assigned then the spring will be connected to a fixed point in space.
- **Anchor**: The point in the object's local space at which the joint is attached.
- **Connected Anchor:** The point in the connected object's local space at which the joint is attached.
- **Spring**: Strength of the spring.
- **Min/Max Distance:** Lower/upper limit of the distance range over which the spring will not apply any force.
- **Enable Collision:** Should the two connected objects register collisions with each other?


Spring Joint

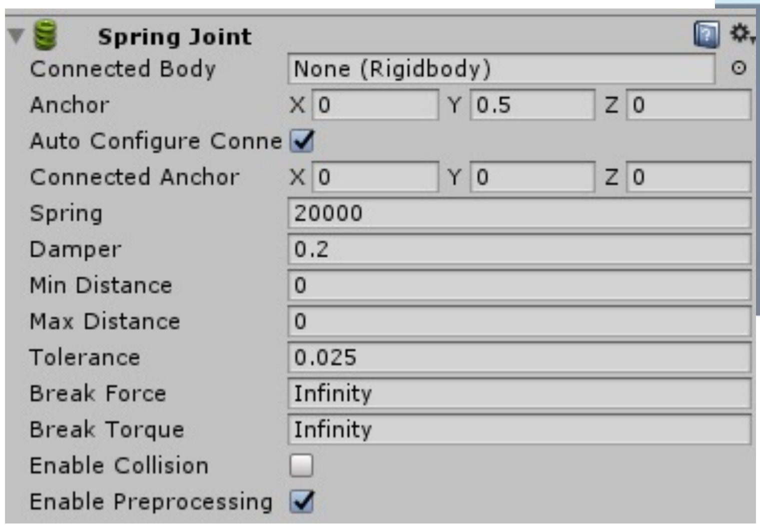
Connected Body	None (Rigidbody)				
Anchor					
X	0	Y	0	Z	0
Auto Configure Conn	☐				
Connected Anchor					
X	0	Y	0	Z	0
Spring	10				
Damper	0.2				
Min Distance	0				
Max Distance	0				
Tolerance	0.025				
Break Force	Infinity				
Break Torque	Infinity				
Enable Collision	☐				
Enable Preprocessing	✓				

Example

- A chain with a heavy object at the end
- The Chain Root object is in the scene at design time, and its ChainRoot script's Start( ) method instantiates the link objects and connects them
- Expanded on the previous example

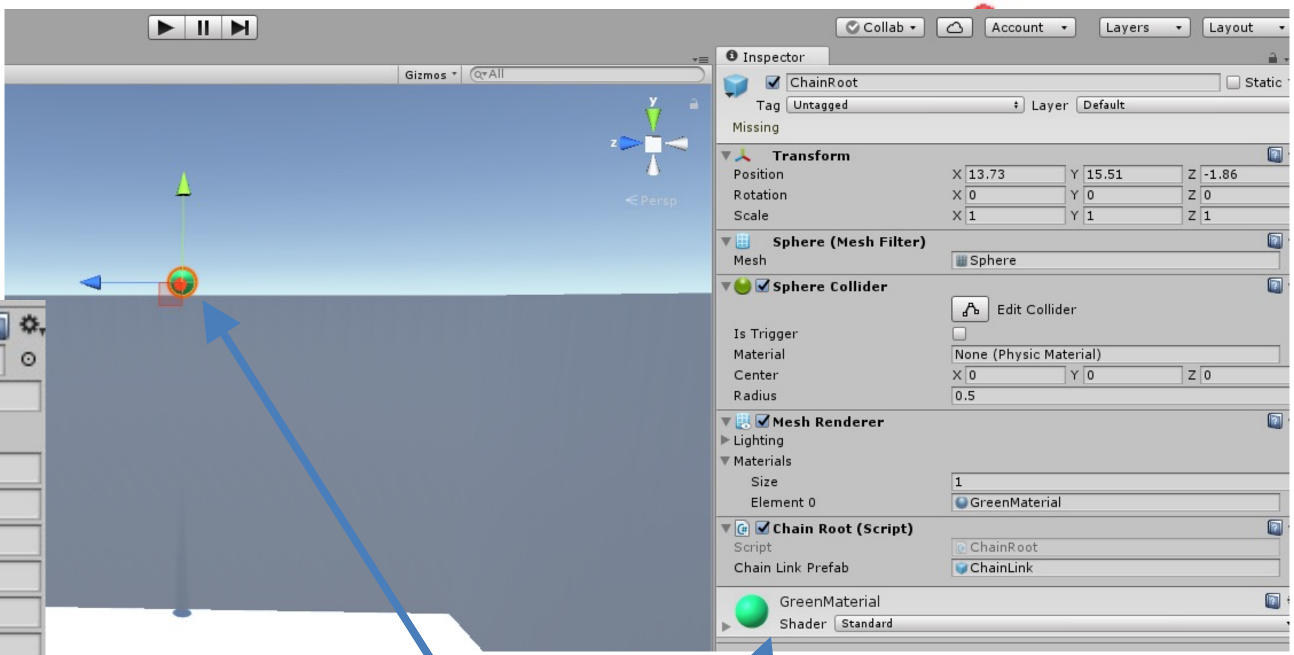


SpringJoint component on the ChainLink object



Spring Joint

Connected Body	None (Rigidbody)
Anchor	X 0 Y 0.5 Z 0
Auto Configure Conne	✓
Connected Anchor	X 0 Y 0 Z 0
Spring	20000
Damper	0.2
Min Distance	0
Max Distance	0
Tolerance	0.025
Break Force	Infinity
Break Torque	Infinity
Enable Collision	☐
Enable Preprocessing	✓



Inspector

ChainRoot ✓ — Static ☐
Tag Untagged — Layer Default
Missing

Transform

	X	Y	Z
Position	13.73	15.51	-1.86
Rotation	0	0	0
Scale	1	1	1

Sphere (Mesh Filter)
Mesh: Sphere

Sphere Collider ✓
Edit Collider

Is Trigger	☐
Material	None (Physic Material)
Center	X 0 Y 0 Z 0
Radius	0.5

Mesh Renderer ✓
Lighting
Materials

Size	1
Element 0	GreenMaterial

Chain Root (Script) ✓

Script	ChainRoot
Chain Link Prefab	ChainLink

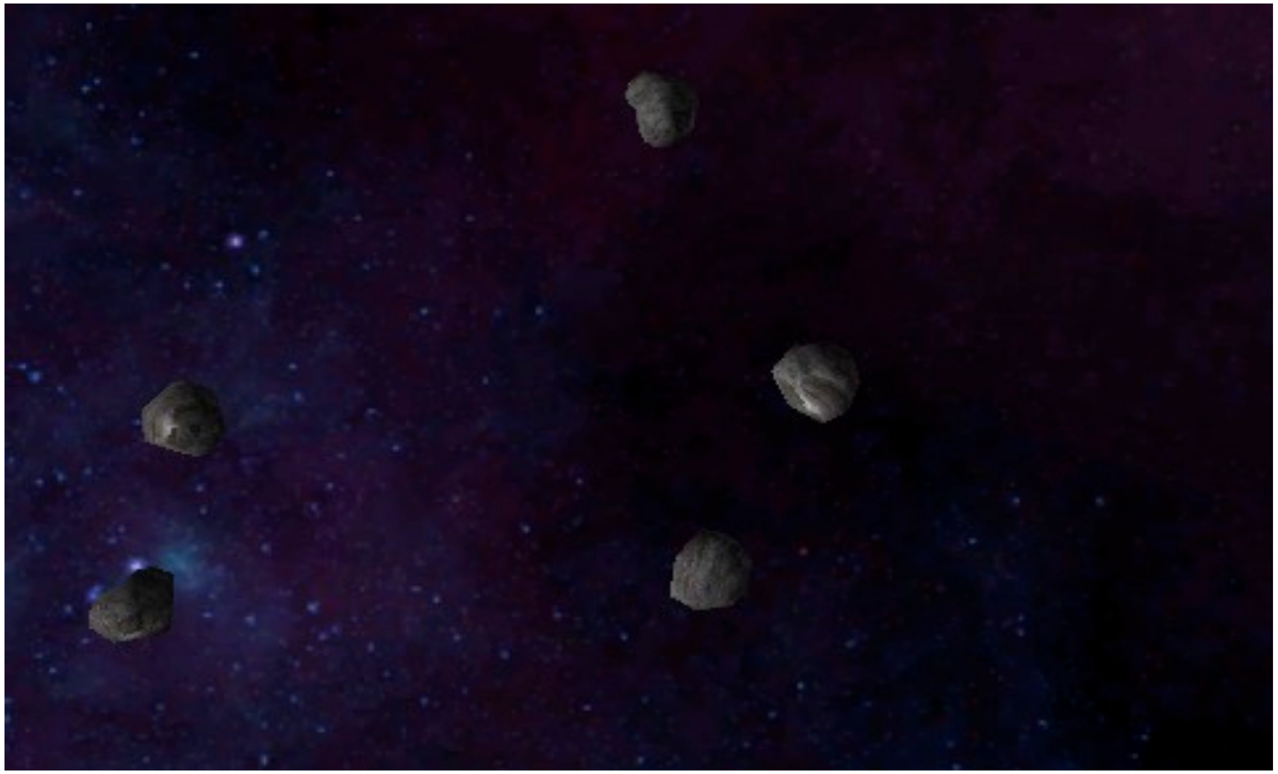
GreenMaterial
Shader: Standard

ChainRoot object

```
public class ChainRoot : MonoBehaviour {

    // inspector settings
    public GameObject chainLinkPrefab;
    //

    void Start () {
        // create a bunch of connected chain links
        Vector3 pos = transform.position;
        Rigidbody previous = this.GetComponent<Rigidbody>();
        Vector3 anchorOffset = new Vector3(0f, 1.5f, 0f);

        for (int i=0; i<10; i++) {
            GameObject go = Instantiate(chainLinkPrefab);
            pos.y -= 1f; // each link is 1m lower than the previous
            go.transform.position = pos;
            SpringJoint sj = go.GetComponent<SpringJoint>();
            sj.connectedBody = previous;
            sj.connectedAnchor = anchorOffset;

            if (i==9) {
                // make the last link bigger and heavier
                go.GetComponent<Rigidbody>().mass *= 5f;
                go.transform.localScale = new Vector3(2f,2f,2f);
            }

            previous = go.GetComponent<Rigidbody>();
        }

    }

}
```

Lab 4

- This week, we're starting a simple Asteroids game
- The game will be progressed over the next 4 weeks
- You may choose to finish off this game as your project; or, you may choose to make a different game. (A higher standard is expected if you complete the Asteroids game).



Lab 4.. How to calculate the screen edges, in world coordinates?

- Assuming the camera is set up as I suggested (i.e. positioned at 0,30,0 and looking at 0,0,0 with its 'up' axis set to 0,0,1):
  - then the bottom left corner of the screen can be obtained using Camera.ViewportToWorldPoint with the viewport point (i.e. screen space) 0,0,30.
- The key thing here is that the z component of this is the distance from the camera that you want to find the world x,y,z of, for viewport point 0,0... and that value needs to be 30 since we know the game objects will be at that distance from the camera; they're all going to have their y positions clamped to zero
- Use the same approach for the top-right corner of the viewport, i.e. using Camera.ViewportToWorldPoint with 1,1,30.
- The reason the z component of Camera.ViewportToWorldPoint is needed is that we're using a perspective camera, so its viewing volume is a cone rather than a box.