CT3536 (Games Programming)
Section 2
Key Concepts and Classes

- The Game Loop and how Unity's MonoBehaviour class fits into it
  - Critical to understanding overall flow of control in the game
- GameObjects
- Cameras
- Transforms
- Statics
- Lab 2: Moving GameObjects programmatically

Lab #1 Code

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

public class GameManagerScript : MonoBehaviour {

    public GameObject marsObject;

    // Use this for initialization
    void Start () {
        Camera.main.transform.position = new Vector3 (0f, 0f, -100f);
        Camera.main.transform.LookAt (marsObject.transform);

        // use the physics engine to rotate Mars
        marsObject.GetComponent<Rigidbody>().AddTorque (new Vector3(0f,20f,0f));
    }

}
```

Lab #1 Code (alternative)

```
public class GameManager : MonoBehaviour
{
  // inspector settings
  public GameObject marsObject;
  //

  void Start() {
    marsObject.transform.position = new Vector3(0,0,0);
    Camera.main.transform.position = new Vector3(0,0,100);
    Camera.main.transform.LookAt(marsObject.transform);
  }

  void Update() {
    // programmatically rotate Mars each frame:
    marsObject.transform.Rotate(new Vector3(0,10*Time.deltaTime,0));
  }

}
```

The Game Loop

- At their core, games operate a *game loop* (although game engines somewhat hide this from you, it's still useful to know):
  - Operating at 60 fps (or more?)
  - Deal with inputs
    - Received asynchronously via events (e.g. keyboard, or network data), or polled for right now
  - Process game objects
    - User-controlled objects
    - Enemy AI
    - Other scheduled behaviours (e.g. sprite frame updates)
  - Move game objects
    - Move the physics simulation (if any) forwards
    - Move objects by physics simulation or direct control
  - Redraw
  - Wait (maybe) or loop as fast as possible (maybe)
    - but don't block the main thread

MonoBehaviour

- https://docs.unity3d.com/ScriptReference/MonoBehaviour.html
- MonoBehaviour is the base class from which Unity C# scripts normally derive
- This hooks the class into the Game Loop (so it automatically receives calls to specific methods at specific times)
- It also provides various other useful Unity **methods** that are called by the game engine in specific situations:

  - Start() OnDestroy()
  - Awake()
  - Update() FixedUpdate() LateUpdate()
  - OnDisable() OnEnabled()
  - OnBecameInvisible() OnBecameVisible()
  - OnCollisionEnter() OnCollisionExit() OnCollisionStay()
  - OnTriggerEnter() OnTriggerExit() OnTriggerStay()
  - OnMouseEnter() OnMouseExit() OnMouseDown() etc.
  - … and others!

  - NB do not confuse Awake() and OnEnabled()

MonoBehaviour

- In games, it is often useful to be able to execute code at programmer-controlled intervals (perhaps much less often than every frame), or at some specified time in the future
- In Unity MonoBehaviour, these **methods** relate to this:
  - Invoke()
  - StartCoroutine()
  - .. more on these later…

- There are also several important public **data members**:
  - enabled (Boolean)
  - gameObject (GameObject)
  - transform (Transform)
  - name (String) (name of gameObject)

MonoBehaviour

- Finally, there are some more **methods** provided which are useful for manipulating GameObjects and their components:
  - SendMessage() BroadcastMessage() SendMessageUpwards()
  - GetComponent() GetComponentInChildren() GetComponentInParent()
  - GetComponents() GetComponentsInChildren() GetComponentsInParent()
  - GetInstanceID()

- Recall, GameObjects can contain many independent scripts (components), each inheriting from MonoBehaviour
- Each script/component is an instance from a class
- A GameObject is therefore composed of multiple software objects
- This is called a 'component based system'.. and is often seen as a superior approach than OO class hierarchies – better for code reuse and isolation of functionality (see last week for discussion)

- Also see:
  - https://docs.unity3d.com/Manual/ExecutionOrder.html

GameObjects in Unity

- https://docs.unity3d.com/ScriptReference/GameObject.html
- GameObject is the base class for all entities that exist in a Unity scene
- (As discussed above) each GameObject has a collection of components attached, and each component is an independent class object inheriting from MonoBehaviour
- The GameObject class is a special class though, that *all* entities in the game are derived from and which you don't have to attach as a component
- Some useful **data members** of GameObject:
  - activeInHierarchy (Boolean)
  - transform (Transform)
  - tag (Tag type as defined in the editor)
- Some useful **methods**:
  - AddComponent()
  - SendMessage() etc.. (same as method in MonoBehaviour classes)
  - GetComponent() etc.. (same as method in MonoBehaviour classes)
  - SetActive()
- Some useful **Static methods** of the GameObject class:
  - Find()
  - Destroy()
  - Instantiate()

Ways of Getting References to GameObjects at runtime

- For a specific GameObject:
  - Have it referenced as a Public member of the script that needs it, and associated at design-time in the inspector (this is what we did in the first lab for the Mars object, a reference to which was needed by the GameManager class)
  - Use GameObject.Find("name") to find it by name (somewhat inefficient so don't do this every frame)
  - Use GameObject.FindGameObjectsWithTag() to find all game objects with a specified tag (returns an array of GameObjects)

Runtime instantiation + Destruction of GameObjects (approach 1)

- So far, we have created all the GameObjects we need in the hierarchy at design-time (i.e. before starting the game)
- Often, we need to instantiate+destroy some (or all) of our game objects at run-time.. e.g. bullets, enemies, explosions, etc.
- Assuming we have a **prefab** in our Assets, and that it is located in a folder called "Resources" (this is a requirement)
- Here, prefabName is the name of our asset prefab (as a string)

```
GameObject go = Instantiate(Resources.Load(prefabName));
```

- And later on in the game.. (assuming we still have a reference to the object.. we should have stored it somewhere!)

```
GameObject.Destroy(go);
```

or

```
GameObject.DestroyImmediate(go);
```

Runtime instantiation + Destruction of GameObjects (approaches 2 & 3)

- Alternatively, you can instantiate an object by supplying an existing instantiated object as a template,
- E.g. in lab #1 our GameManagerScript class has a GameObject member called marsObject, which is instantiated in the inspector (at design time), so you could use this to make another Mars:

```
GameObject otherMarsObject = Instantiate(marsObject);
```

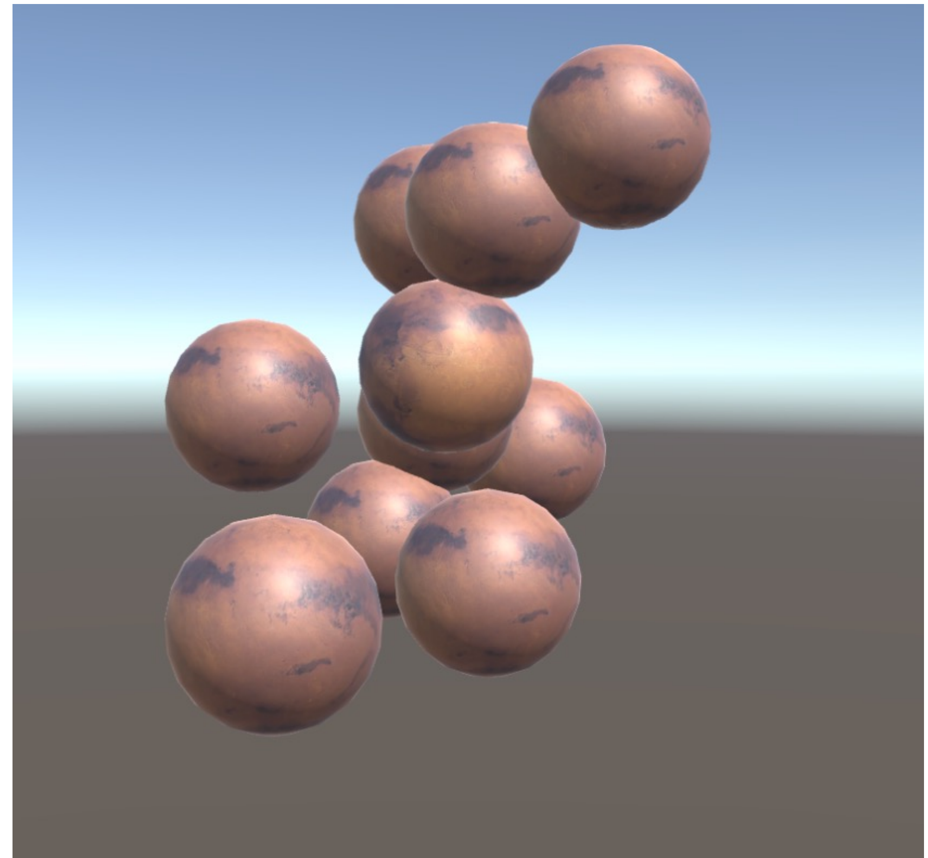
- Another alternative (perhaps the easiest of all?) is to have a public reference to a GameObject and associate this *with a **prefab** in the assets* by dragging in the inspector.
- Then you can use Instantiate() on that to instantiate a copy of the prefab into the scene at runtime.

Prefabs

- When you have created a GameObject in the hierarchy (at design time), added components to it, and set their various public values, you can drag the GameObject into the assets window to make a **prefab** - i.e. a template of that exact object with its settings
- This approach means that you can make Monobehaviour scripts for monsters, guns, etc., and then make a separate prefab with different settings for each actual type of monster or gun you need in your game
- Prefabs are useful therefore for creating and editing your game's data
- E.g. in DemonPit we use prefabs for things such as MonsterWaves, Guns, Projectiles (missiles), and Monsters -- see following slides

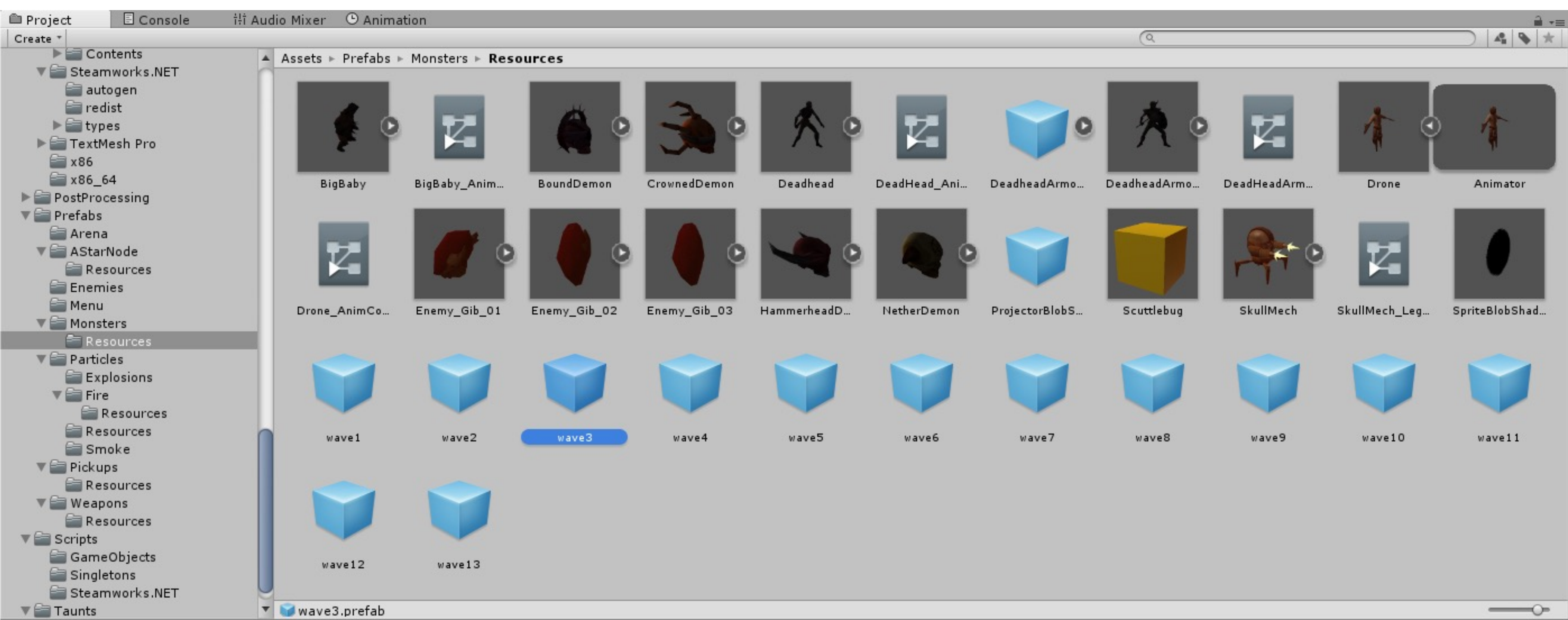
Example.. Many Marses

- Give GameManager a public reference to a GameObject and drag a prefab (*from Assets*) into it.
- Instantiate a bunch of copies of this at random positions (see code on next slide).

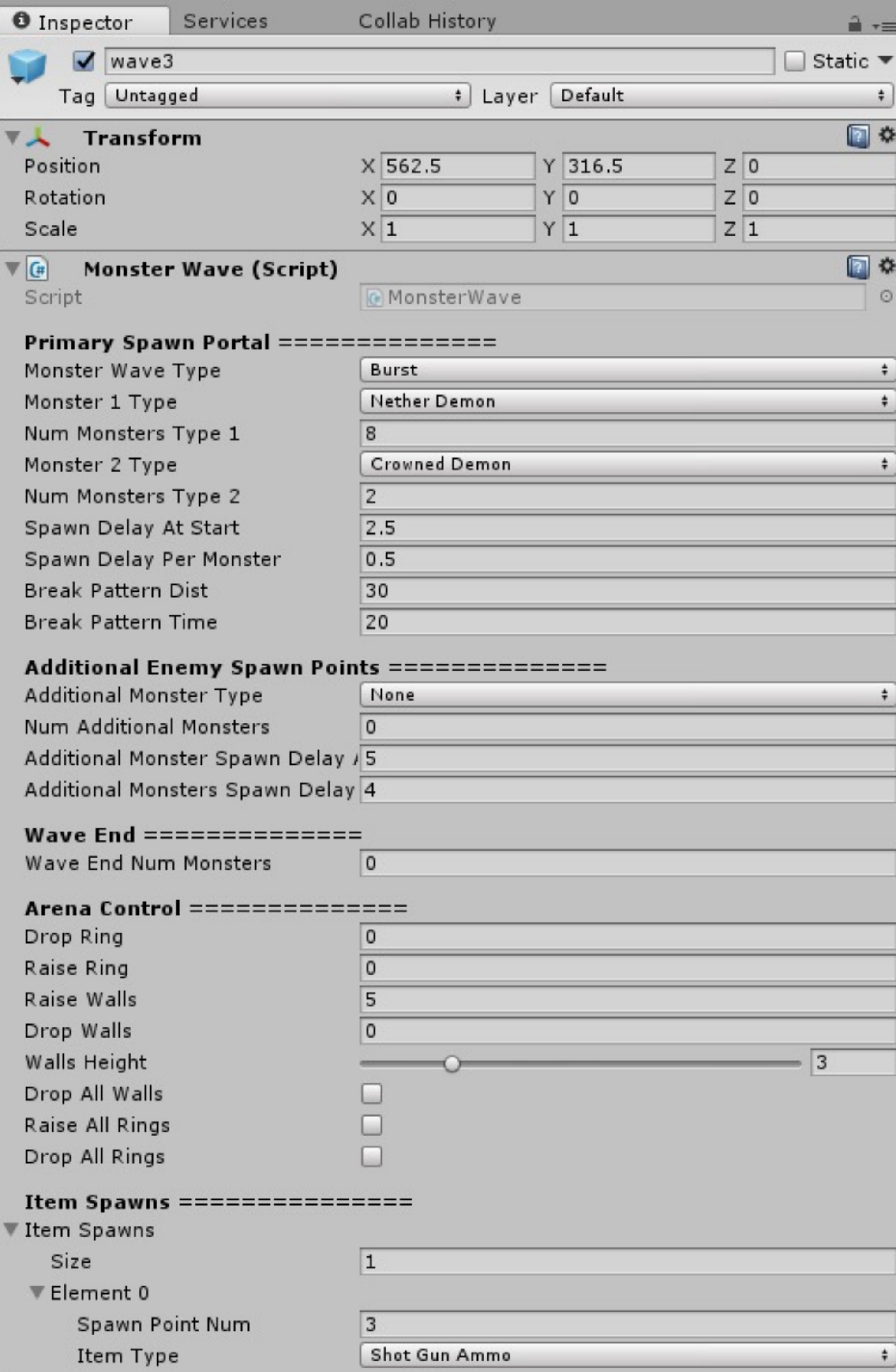


```
public class GameManagerScript : MonoBehaviour
{
    public GameObject goMars;
    public GameObject prefabMars;

    // Start is called before the first frame update
    void Start()
    {
        Camera.main.transform.position = new Vector3(0, 0, -100);
        Camera.main.transform.LookAt(new Vector3(0,0,0));

        Rigidbody rb = goMars.GetComponent<Rigidbody>();
        rb.AddTorque(new Vector3(0,30,0));

        MakeMoreMarsObjectsAtRuntime();
    }

    private void MakeMoreMarsObjectsAtRuntime() {
        // runtime-instantiate a bunch of Mars game objects, cloned from the
        // prefab associated with 'prefabMars'
        // why? why not.
        for (int i=0; i<10; i++) {
            // when you Instantiate, the new GameObject is created in the hierarchy
            GameObject newMars = Instantiate(prefabMars);
            newMars.transform.position = new Vector3(
                    Random.Range(-100f,100f),
                    Random.Range(-100f,100f),
                    Random.Range(-100f,100f)
                );
        }

        // move the camera back a bit more to see all the Marses
        Camera.main.transform.position = new Vector3(0, 0, -500);
    }
}
```

DemonPit - Wave Prefabs



- The Wave prefabs in DemonPit are GameObjects with no visual presence in the game (i.e. no Renderer component), and no presence in the Physics world either (i.e. no RigidBody or Collider components)
- They're essentially just data containers
  - (You can also use ScriptableObjects for this)
- I have exposed Public data members so that my artist collaborator can edit the waves without needing to see any C# code



DemonPit - wave Prefab and MonsterWave class inspector settings

```
public enum MonsterWaveType {
    Fountain = 1,
    Vortex = 2,
    Burst = 3,
    Emit = 4
}

[System.Serializable]  // Serializable so we can fill these
in inspector
public class ItemSpawn {
    public int spawnPointNum = 1;
    public ItemSpawnType itemType =
                     ItemSpawnType.MachineGunAmmo;
}

public class MonsterWave : MonoBehaviour {

    // inspector settings
    [Header("Primary Spawn Portal ==============")]
    public MonsterWaveType monsterWaveType;
    public MonsterTypes monster1Type =
MonsterTypes.NetherDemon;
    public int numMonstersType1;
    public MonsterTypes monster2Type = MonsterTypes.None;
    public int numMonstersType2;
    public float spawnDelayAtStart = 2.5f; // time before any
spawn, while walls are changing
    public float spawnDelayPerMonster = 0.5f; // time between
each monster
    public float breakPatternDist = 30f; // below this
distance, chases player; above it, does idle pattern if
applicable
    public float breakPatternTime = 20f; // this is the max
time that monsters stay in idle pattern, regardless of
distance to player
```

Continued on next slide..

MonsterWave class inspector settings, continued..

```
[Header("Additional Enemy Spawn Points ==============")]
// additional enemy spawn point is picked based on position of player:
// nearest point that's at least 3 metres away
public MonsterTypes additionalMonsterType = MonsterTypes.None;
public int numAdditionalMonsters;
public float additionalMonsterSpawnDelayAtStart = 5f;
public float additionalMonstersSpawnDelayPerMonster = 4f;

[Header("Wave End ==============")]
public int waveEndNumMonsters = 2;

[Header("Arena Control ==============")]
public int dropRing = 0;
public int raiseRing = 0;
public int raiseWalls = 0;
public int dropWalls = 0;
[Range(2f, 7f)] public float wallsHeight = 2f;
public bool dropAllWalls, raiseAllRings, dropAllRings;

[Header("Item Spawns ===============")]
public ItemSpawn[] itemSpawns;
```

DemonPit - Gun Prefabs

Gun (Script)

Script	Gun
Common Settings ==========================	
Gun Name	Machinegun
Gun Type	Machine Gun
Gun Enum Type	Machine Gun
Muzzle Flash	MuzzleFlash
Shoot Trajectory	ShootTrajectory
Limited Ammo	☑
Rotate Muzzle Flash	☑
Muzzle Flash Display Time	0.05
Clip Size	45
▼ Shoot Sounds	
Size	4
Element 0	DemonPit Machine Gun Sounds Firing_1
Element 1	DemonPit Machine Gun Sounds Firing_2
Element 2	DemonPit Machine Gun Sounds Firing_3
Element 3	DemonPit Machine Gun Sounds Firing_4
Dry Fire Sound	HANDLING_B_FS92_9mm_Dry_Fire_From_Mai
Reload Sound	RELOAD_Pump_stereo
Activate Sound	HANDLING_B_FS92_9mm_Cocking_Motion_Ch
Shoot Volume	1
Cooldown Secs	0.07
Recoil Settings ==========================	
Screen Shake	5
Camera Drift	0.5
Recoil Dist	X 0.05 Y 0.1 Z -0.2
Recoil Rotation	-2.5
Recoil Time Kickback	0.01
Kickback Curve	
Recoil Time Recovery	0.06
Recovery Curve	
Recoil Fov Change	0

Raycast-Type Settings =====================	
Is Raycast Gun	☑
▶ Ricochet Sounds	
▶ Monster Hit Sounds	
Max Range	500
Spark Prefab	BulletSpark
Hit Smoke Prefab	Smoke_small
Bullethole Decal Prefab	BulletholeDecal
Projectile-Type Settings ================	
Is Projectile Gun	☐
Muzzle Point	None (Transform)
Projectile Prefab	
Inherit Player Velocity	☑
Particle-Type Settings =================	
Is Particle Gun	☐
Damage Per Hit	0.25
▼ Shoot Particle Systems	
Size	0

Add Component

Cameras

- In computer graphics, cameras are software objects that define a viewpoint (in terms of position, orientation, field-of-view/zoom, 3D-v-Orthographic etc.) within the ‘virtual world'
- Mathematical projections are used to calculate how the 3D world should be displayed on the 2D camera surface (more on this next year! – CT404)
- In Unity, Cameras are GameObjects which have a Camera component. Since they're GameObjects, you can add other components, manipulate their Transform, etc., just like any other GameObject
- The Transform component provides methods that are very useful to cameras (as well as perhaps other non-camera objects), e.g. LookAt
- Camera.main is a static reference to the 'main' camera whose view is being displayed to the screen. It should have the tag "MainCamera"
- Other cameras can be used at the same time, e.g. to render to texture (e.g. mirrors, top-down minimaps), or to render to parts of the screen (e.g. split-screen multiplayer)
- You might also have multiple cameras in a scene and switch the active one during the game

Cameras

- https://docs.unity3d.com/ScriptReference/Camera.html
- https://docs.unity3d.com/Manual/class-Camera.html
- The camera class provides methods for converting coordinates between Screen-space, Viewport-space, and World-space (see later)

CalculateFrustumCorners: Given viewport coordinates, calculates the view space vectors pointing to the four frustum corners at the specified camera depth.

ScreenPointToRay: Returns a ray going from camera through a screen point.

ScreenToViewportPoint: Transforms position from screen space into viewport space.

ScreenToWorldPoint: Transforms position from screen space into world space.

ViewportPointToRay Returns a ray going from camera through a viewport point.

ViewportToScreenPointTransforms position from viewport space into screen space.

ViewportToWorldPointTransforms position from viewport space into world space.

WorldToScreenPointTransforms position from world space into screen space.

WorldToViewportPointTransforms position from world space into viewport space.

Transforms

- https://docs.unity3d.com/ScriptReference/Transform.html
- A Transform defines the position, orientation, and scale of an object
- Child objects have their own Transform which is interpreted as a nested co-ordinate system, i.e. a modification of the parent's
- This is a very powerful concept and is fundamental to the concept of a SceneGraph/hierarchy
- In Unity, the Transform class has these key members:
  - position (Vector3)
  - localPosition (Vector3)
  - rotation (Quaternion)
  - localRotation (Quaternion)
  - lossyScale (Vector3)
  - localScale (Vector3)
  - parent (Transform)
  - right, up, forward (each are Vector3, normalized)
  - gameObject (GameObject)
- "right" – positive direction on the local x axis
- "up" – positive direction on the local y axis
- "forward" – positive direction on the local z axis

Transforms

- **Methods** of the transform class include:
  - Rotate()
  - Translate()
  - TransformPoint()   InverseTransformPoint()
  - LookAt()
  - RotateAround()
  - SetParent()

TransformPoint Transforms position from local space to world space.

InverseTransformPoint Transforms position from world space to local space.

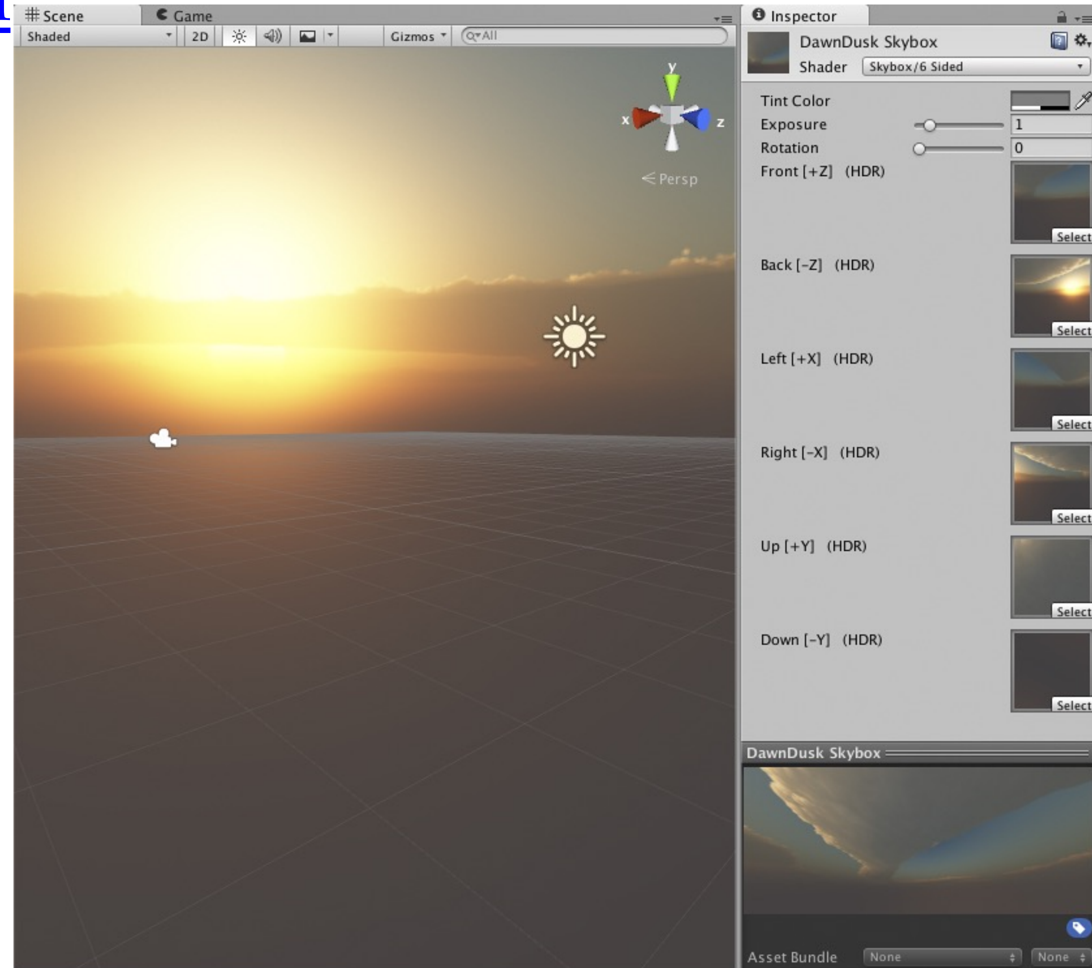
Rotate uses "Euler angles" (x,y,z) – all in degrees

LookAt(Vector3 point) turns so that the forward direction (i.e. positive z axis) faces the specified position

RotateAround(Vector3 point, Vector3 axis, float degrees)

Skyboxes

- https://docs.unity3d.com/Manual/class-Skybox.html
- Skyboxes are rendered around the whole scene (in the background) in order to give the impression of complex scenery at the horizon.
- To implement a Skybox, create a skybox Material in your Assets. Then add it to the scene by using the **Window -> Rendering -> Lighting** menu item and specifying your skybox material as the Skybox on the Scene tab.



Keyboard Input

- https://docs.unity3d.com/ScriptReference/Input.html
- We will revisit more details about Keyboard/Mouse/ Joystick/ Touchscreen input later, but for now:

- **Input.GetKey()** returns true as long as a key is held down
- https://docs.unity3d.com/ScriptReference/Input.GetKey.html
- E.g. this could be written in an Update() method:

```
if (Input.GetKey(KeyCode.LeftArrow))
    RotateCameraAroundMarsSomehow();
```

Lab #2
Moving GameObjects programmatically

- In our first lab, we attached a Rigidbody to the Mars object, applied angular velocity to it (using AddTorque), and let the game engine control the movement of it
- The other way of moving game objects is to directly manipulate their Transform using code (this was our approach in 2nd year Java)
- In lab #2 we'll be making Mars' two moons (Phobos and Deimos) orbit around it through direct manipulations of their Transforms
- Your code for doing this would typically be written in the Update() method of some script (- it could either be a script attached to the GameObject itself, or perhaps a script attached to a singleton 'manager' object)
- It is important to consider the fact that Update() is probably not happening at fixed time intervals
  - You can multiply all movement code by **Time.deltaTime**

Static Member Variables
(A quick refresher on the use of **static** in C#)

- By declaring a member variable static, you're creating a single copy of the variable which is owned by the class (rather than any specific instanced object). Each instance does not have its own unique copy. E.g. in DemonPit the Gun class has these statics:

```
private static float SHOTGUN_SPREAD_ANGLE = 6f;
private static int SHOTGUN_NUM_BULLETS = 12;
private static Vector3 screenShakePosInfluence = new Vector3(0.15f, 0.15f, 0.15f);
private static Vector3 screenShakeRotInfluence = new Vector3(1, 0, 0);
private static Vector3 xAxis = new Vector3(1f,0f,0f);
private static Vector3 camOffsetFromPlayerCentre = new Vector3(0f, 0.4f, 0f);
```

- A common use-case for static member variables is if you want to retain a collection of all instances of the class, for quick recall, e.g. in "Echoes of the Past" the FarmCrop class has this:

```
public static LinkedList<FarmCrop> instantiatedFarmCrops = new
LinkedList<FarmCrop>();
```

Static Member Functions
(A quick refresher on the use of **static** in C#)

- By declaring a method static, you're creating a method that is called on the class itself, not on an instance of it
- This is very useful since you do not need to have reference to an instantiated object from the class in order to call it
- For obvious reasons, a static method only has access to static member variables of the class, while a non-static method (which is called on an actual instantiated object) has, in addition, access to its own copies of non-static member variables
- Here's a static method from the GameManager class in DemonPit:

```
public static void GameOver() {
   Player.DeactivatePlayer();
   GUIManager.ShowFinalScore();
   LeaderboardManager.PostLeaderboardScore();
   GUIManager.ShowTaunt();
}
```

The Singleton Pattern
(A quick refresher on the use of **static** in C#)

- A common design approach in Unity is to create single-instance (singleton) classes for various 'management' roles
- E.g. GameManager, GUIManager, AudioManager, SaveGameManager, etc.
- The typical approach is to create a GameObject in the hierarchy which holds the one-and-only instance of these components that will exist at run-time (my own habit is to attach them to the main camera)
- Using statics can make the methods of these management singletons very easy and clean to access, e.g. consider the 'instance' static member variable used here; it gives direct access to data attached to the instantiated singleton object, from anywhere in the game, via **GameManager.instance**

```
public class GameManager : MonoBehaviour {
	public static GameManager instance;

	void Start () {
		instance = this;
	}
}
```

DemonPit Manager Singletons

- This picture shows all of the components which I have attached the to Main Camera in DemonPit
- The use of the 'instance' public static member makes it convenient to access these values from anywhere in the game
- This approach also makes it very easy for other (non-programmer) team members to edit things

- (The final component, "Post Processing Behaviour", controls camera-specific post-processing settings)

Some very small components

- In some cases, a simple behaviour that is useful to multiple different types of game object can be written as a very small script.
- This is the very essence of Entity Component Systems!
- E.g. LookAtPlayer will keep the object facing the player. Useful e.g. for 2D sprites in a 3D game:
- Here ‘thePlayer’ is a public static member of my Player class
  - There will only ever be one instantiated Player object in Demon Pit, and it’s useful to be able to access it from anywhere in the game

```
public class LookAtPlayer : MonoBehaviour {
  void Update () {
      transform.LookAt(Player.thePlayer.transform.position);
  }
```