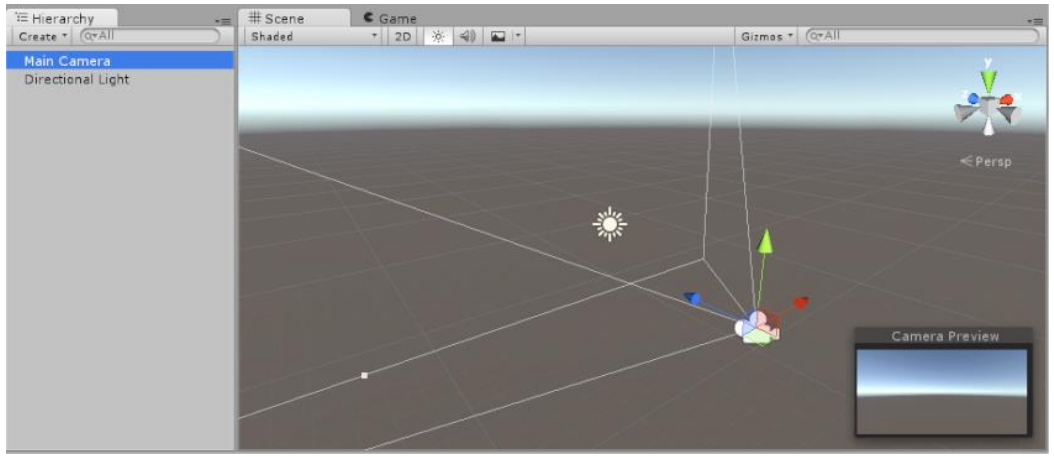
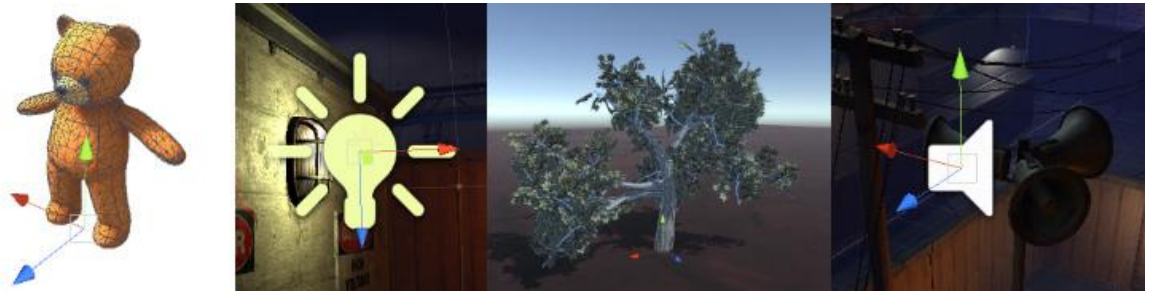
# Unity fundamental concepts

# Scenes




- Scenes contain the environments and menus of your game. Think of each unique **Scene** file as a unique level. In each **Scene**, you place your environments, obstacles, and decorations, essentially designing and building your game in pieces.
- When you create a new Unity project, your **scene view** displays a new Scene.
- The Scene is empty except for a **Camera** (called **Main Camera**) and a Light (called **Directional Light**).
- It is possible to have multiple Scenes open for editing at one time.
- You can load a scene at runtime using **SceneManager**.
- You can load a scene asynchronously in the background using **LoadSceneAsync** function

# Game Objects



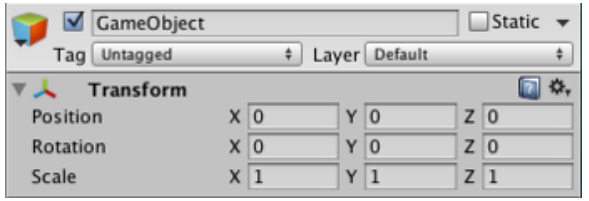
- **GameObjects** are the fundamental objects in Unity that represent characters, props and scenery. They do not accomplish much in themselves but they act as containers for **Components**, which implement the real functionality.
- A GameObject always has a Transform component attached (to represent position and orientation) and it is not possible to remove this. The other components that give the object its functionality can be added from the editor's **Component** menu or from a script. There are also many useful pre-constructed objects (primitive shapes, Cameras, etc).

# Scripting

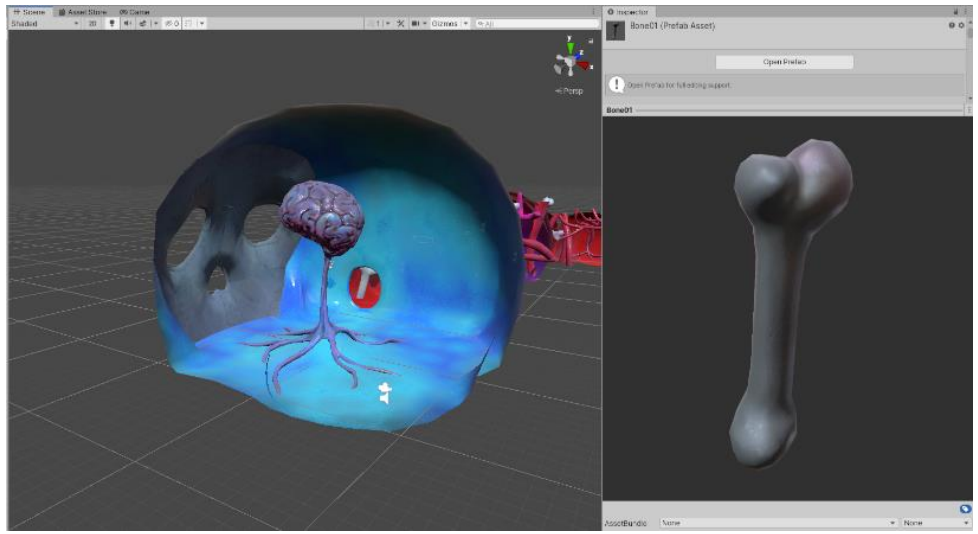
- Unity has its own built-in behavior class called MonoBehaviour.
- MonoBehaviour is the base class from which every Unity script derives.
- When you use C#, you must explicitly derive from MonoBehaviour.
- To write the code, Unity offers a Visual Studio 2017 community installation.
- Editing and saving scripts in Visual Studio (PC) or MonoDevelop (macOS) will immediately update the script in Unity.

# Components

- Game object can contain multiple components.
- The **Transform Component** is added by default for any game object.
- This component defines the **GameObject's** position, rotation, and scale.
- You can add Components to the selected GameObject through the Components menu.
- There are two main types of Properties: **Values** and **References** in the component.



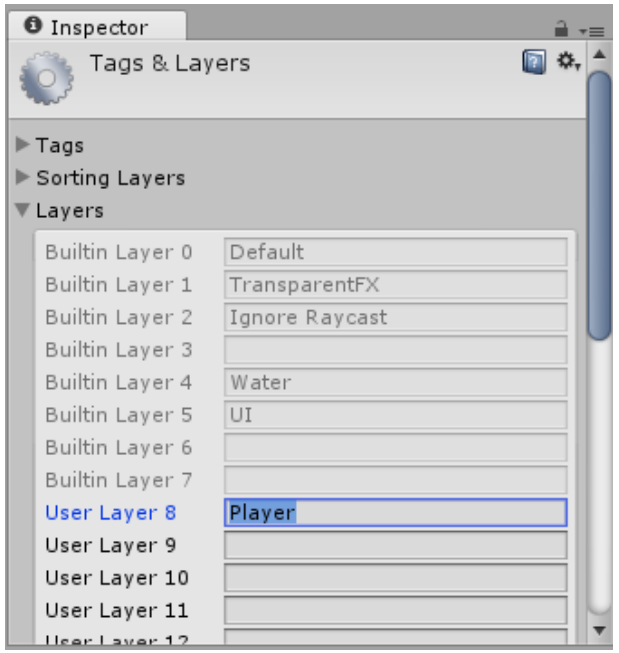
# Prefabs



- Unity's **Prefab** system allows you to create, configure, and store a GameObject complete with all its components, property values, and child **GameObjects** as a reusable Asset.
- Any edits that you make to a Prefab Asset are automatically reflected in the instances of that Prefab.
- You can nest Prefabs inside other Prefabs to create complex hierachies of objects that are easy to edit at multiple levels.
- You should also use Prefabs when you want to instantiate GameObjects at runtime that did not exist in your Scene at the start.

# Layers

- Layers primarily used to restrict operations such as raycasting or rendering.
- They are only applied to the relevant groups of game objects.
- The first eight **Builtin Layers** are defaults used by Unity, so you cannot edit them. However, you can customise **User Layers** from 8 to 31.
- Using layers you can cast rays and ignore colliders in specific layers.
- You can use Layers to ignore physics collision between certain objects.
- **Layers** are most commonly used by **Cameras** to render only a part of the scene, and by **Lights** to illuminate only parts of the scene.



# Tags

- A **Tag** is a reference word which you can assign to one or more **GameObjects.**
- For example, you might define “Player” Tags for player-controlled characters and an “Enemy” Tag for non-player-controlled characters.
- Tags help you identify GameObjects for scripting purposes.
- You can use the GameObject.FindWithTag() function to find a GameObject by setting it to look for any object that contains the Tag you want.

# Coroutines

- A coroutine is like a function that has the ability to pause execution and return control to Unity but then to continue where it left off on the following frame.
- It is essentially a function declared with a return type of IEnumerator and with the yield return statement included somewhere in the body.
- A coroutine is resumed on the frame after it yields but it is also possible to introduce a time delay using WaitForSeconds.
- This would greatly reduce the number of checks carried out without any noticeable effect on gameplay.

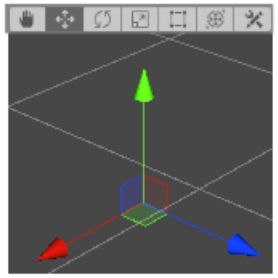
```
IEnumerator Fade()
{
    for (float ft = 1f; ft >= 0; ft -= 0.1f)
    {
        Color c = renderer.material.color;
        c.a = ft;
        renderer.material.color = c;
        yield return null;
    }
}
```

# Player Input Settings

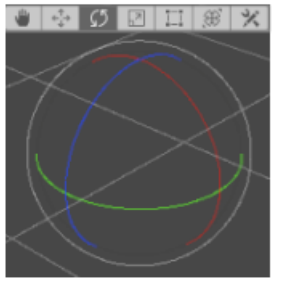
- Use the **Input** settings (top Menu: **Edit** > **Project Settings**, then select the **Input** category) to define the input axes and game actions for your Project.
- All the axes that you set up in the **Input** settings serve two purposes:
- 1- They allow you to reference your inputs by axis name in scripting.
- 2- They allow the players of your game to customize the controls to their liking.
- It is best to write your **scripts** making use of axes instead of individual buttons, as the player may want to customize the buttons for your game.

# Transforms

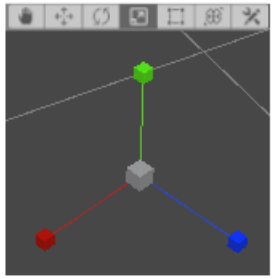
- The **Transform** is used to store a GameObject's position, rotation, scale and parenting state and is thus very important.
- Transforms are manipulated in 3D space in the X, Y, and Z axes or in 2D space in just X and Y.
- When a GameObject is a **Parent** of another GameObject, the **Child** GameObject will move, rotate, and scale exactly as its Parent does.
- Non-uniform scaling is when the **Scale** in a Transform has different values for x, y, and z.
- The physics engine assumes that one unit in world space corresponds to one metre. If an object is very large.



Translate (W)



Rotate (E)



Scale (R)