Unity Game Engine

Unity

- A cross-platform game engine initially released by Unity Technologies in 2005
- Both 2D and 3D games/interactive contents can be developed
- Supports over 20 platforms
  - Most popular are PC, Android and iOS systems

Installation

- **The Hub** - primary way to install
  - Unity Editor
  - Create projects
  - Manage your unity experience
  - Main advantage – multiple versions of Unity can be managed at one place!

- You can also install Unity Editor only - **U**sing command line
  - Install Unity offline without using the Hub
  - Only one version

Installation

- Current Unity version 2021.2.9f1

- We will discuss version 2020.3.26f1 (LTS)
  - Long term support
  - Basic concept for all the version is same
  - We will use different versions of unity while building applications later

Installation

- Unity Hub
  - A standalone application that streamlines the way you find, download, and manage your Unity Projects and installations
  - Can manually add versions of Unity that are already installed (if any) to hub
  - Easy to install unity using unity hub

Installation

- System requirements (for **development**)
  - **OS**
    - Windows 7 SP1+ (64-bit versions only)
    - macOS 10.13+
    - Ubuntu 16.04, Ubuntu 18.04 and CentOS 7
  - **CPU**
    - Any modern Intel and AMD CPU (supporting SSE2 instruction set)
  - **GPU**
    - Graphics card with DirectX10 + (for windows)
    - Metal capable intel and AMD GPU (for macOS)
    - OpenGL 3.2 + or Vulkan capable, Nvidia and AMD GPUs
  - **RAM**
    - 8-16 GB (more is better)

Installation

- Once unity hub is installed we can
  - Install unity editor
  - Activate license

Installation

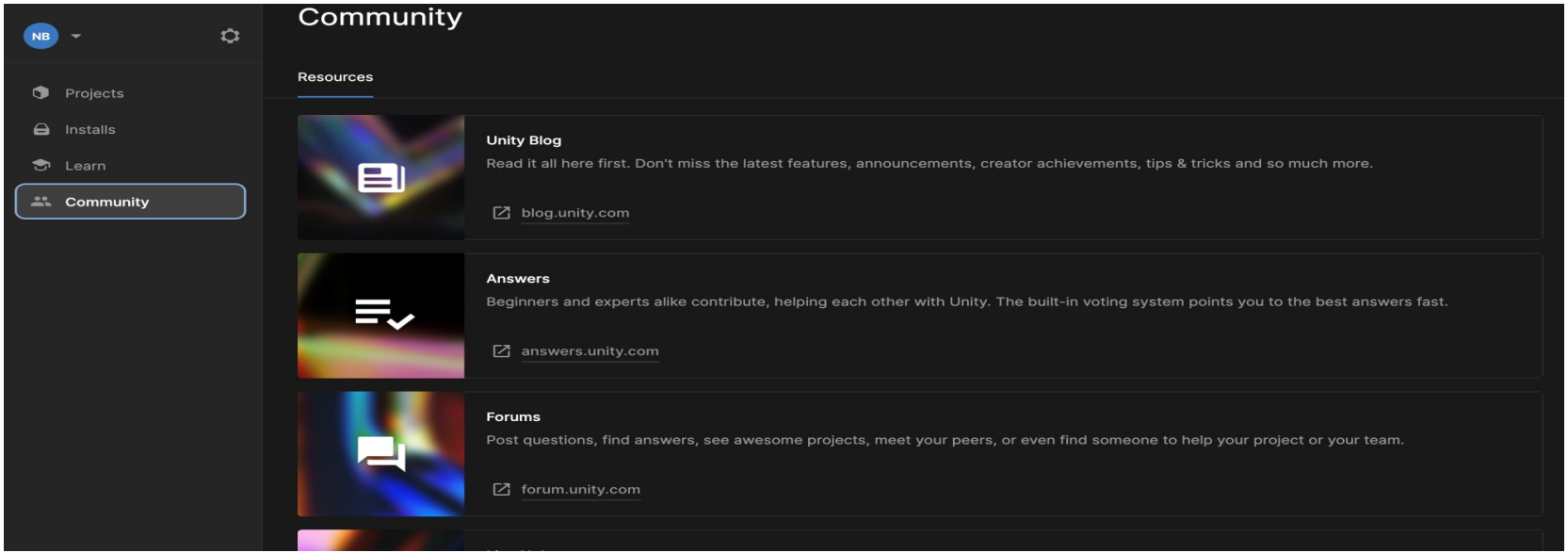
- **Unity player** system requirements (Desktop)
  - **OS**
    - Windows 7 SP1+
    - macOS 10.13+
    - Ubuntu 16.04 and Ubuntu 18.04
  - **CPU**
    - Any modern Intel and AMD CPU (supporting SSE2 instruction set)
  - **GPU**
    - Graphics card with DirectX10 + (for windows)
    - Metal capable intel and AMD GPU (for macOS)
    - OpenGL 3.2 + or Vulkan capable, Nvidia and AMD GPUs
  - **RAM**
    - 8 GB (16 GB preferable)

Installation

- **Unity player** system requirements (Mobile)
  - **OS**
    - Android 4.4+ (API 19)
    - iOS 11+
  - **CPU**
    - Any modern android smartphone CPU (ARMv7 with Neon support (32 bit) or ARM64 for android)
    - Any iPhone 5s+ model CPU (A7 SoC+ for iOS)
  - **RAM**
    - 8 GB (16 GB preferable)

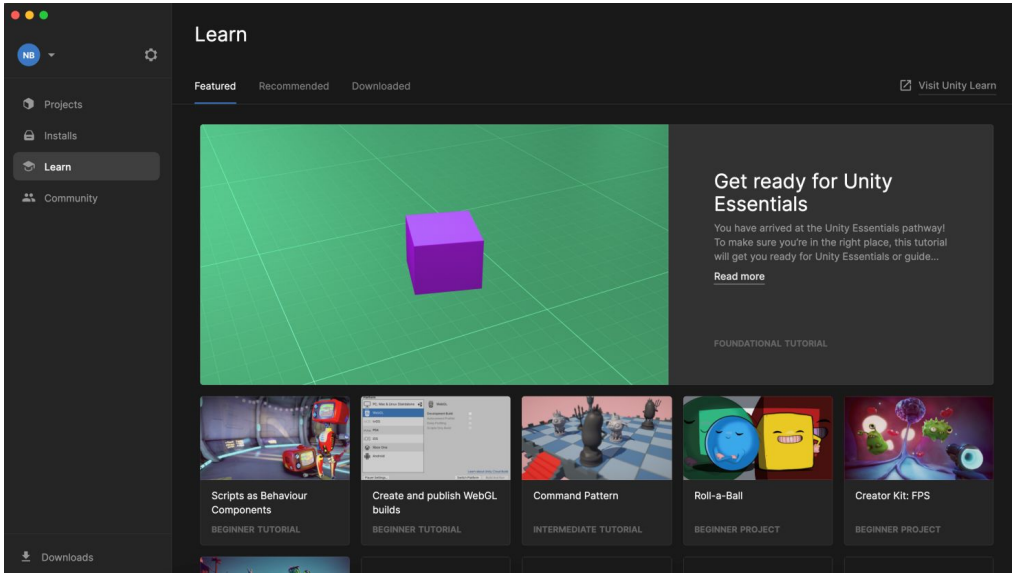
Unity Hub Interfaces

- The community screen
  - Access resources like blogs, answers, forums, live help etc



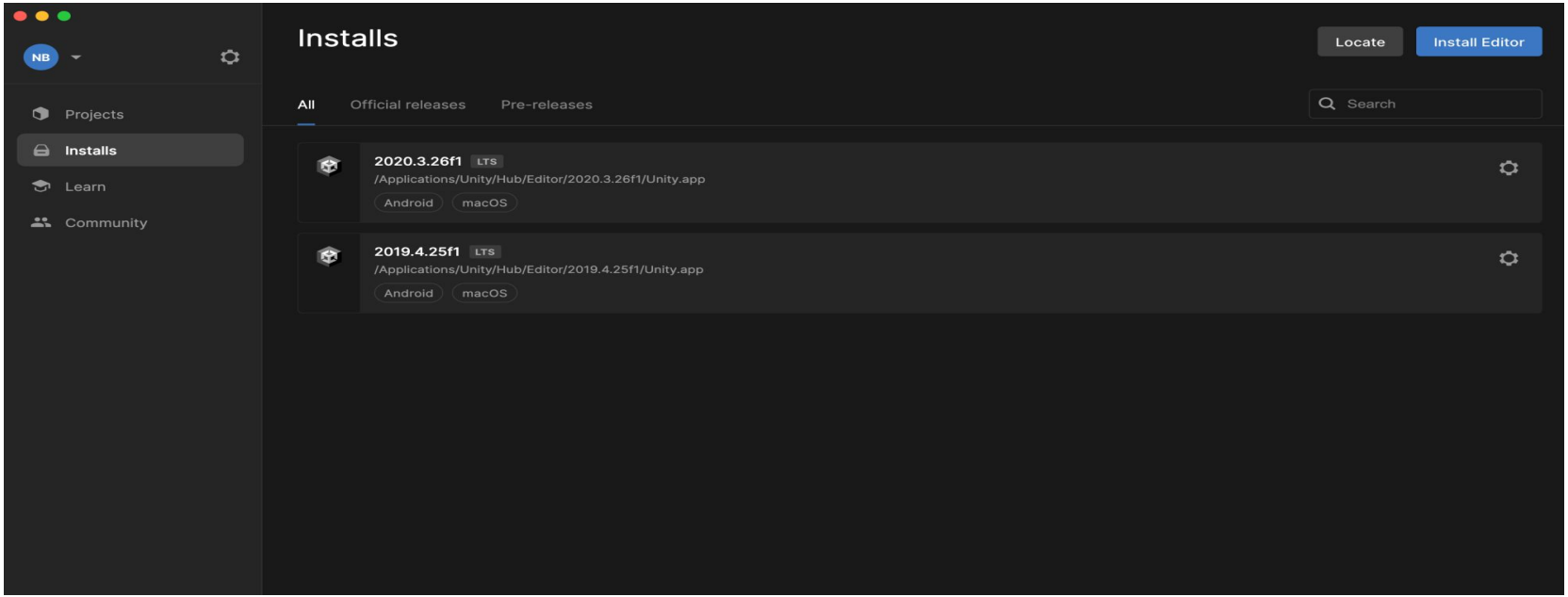
Unity Hub Interfaces

- The learn screen
  - Gives you access to a variety of tutorials and learning resources
  - Includes some example projects that can be imported directly to unity



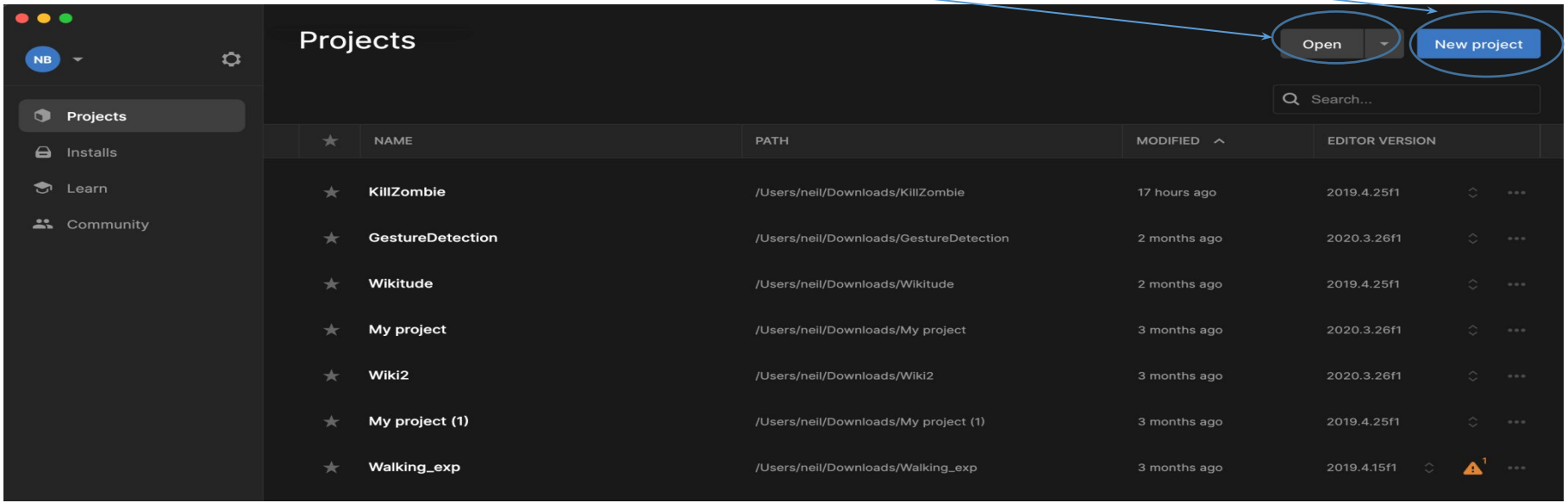
Unity Hub Interfaces

- The installs screen
  - Shows all installed versions of unity here
  - You can also install/locate other versions of unity



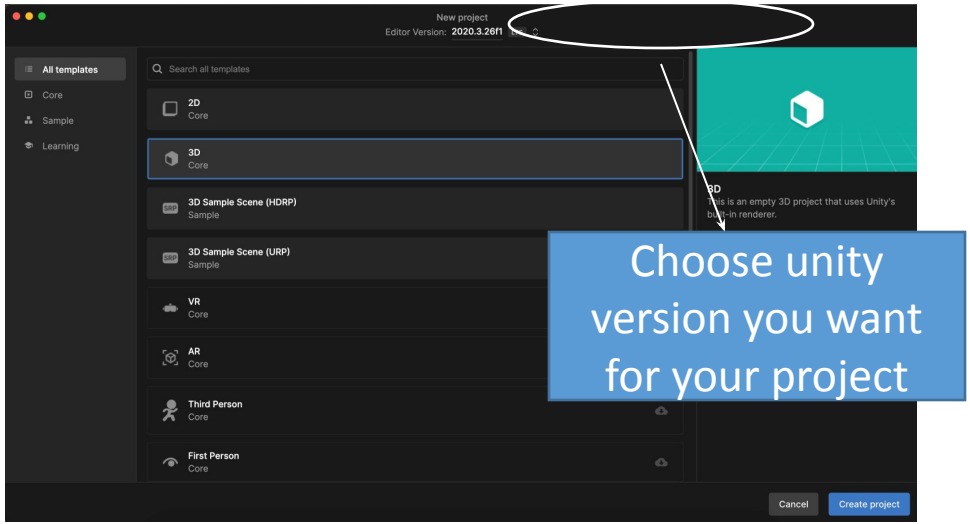
Unity Hub Interfaces

- The Projects screen
  - Projects tab contains the projects you have created in unity
  - Using new project you can create a new project
  - To open an existing project you can click the open button and select the project from your device

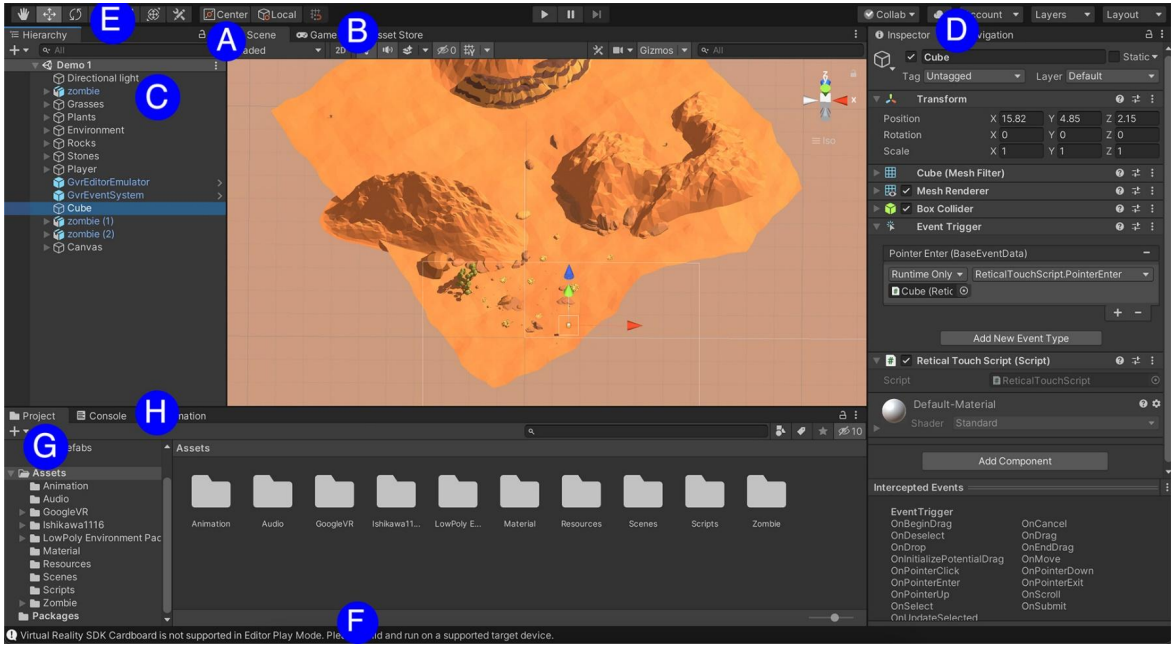


Unity Hub Interfaces

- Creating a project
  - Create a project and select a suitable template for your project (eg. Select 3D for 3D applications)
  - Choose the unity version, name your project and create the project



Unity Editor Interfaces



A. The scene view
B. The game view
C. The hierarchy window
D. The inspector window
E. The toolbar
F. The status bar
G. The project window
H. The console window

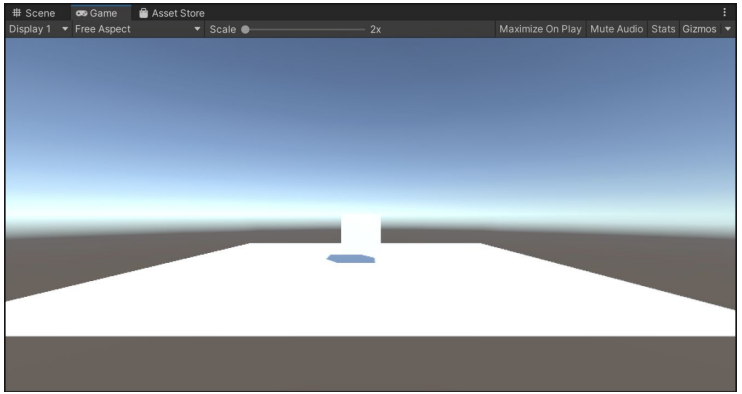
The Scene view

- Interactive view of the world you are creating

- Selecting, manipulating, and modifying GameObjects (most fundamental objects, more about it later) in the scene can be done

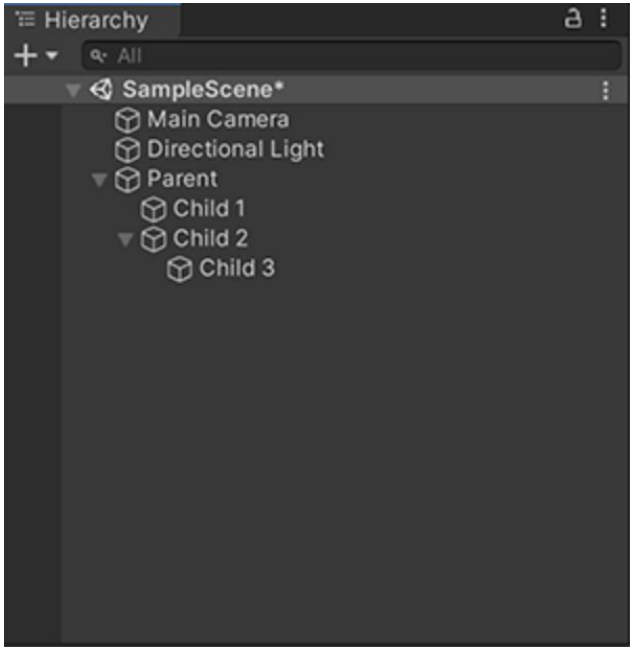
The Game View

- Rendered from the camera(s) in your application
- Represents final published application
- Consists of buttons
  - Display
  - Aspect ratio
  - Scale slider
  - Maximize on play
  - Mute audio
  - Stats (rendering statistics)
  - Gizmos+



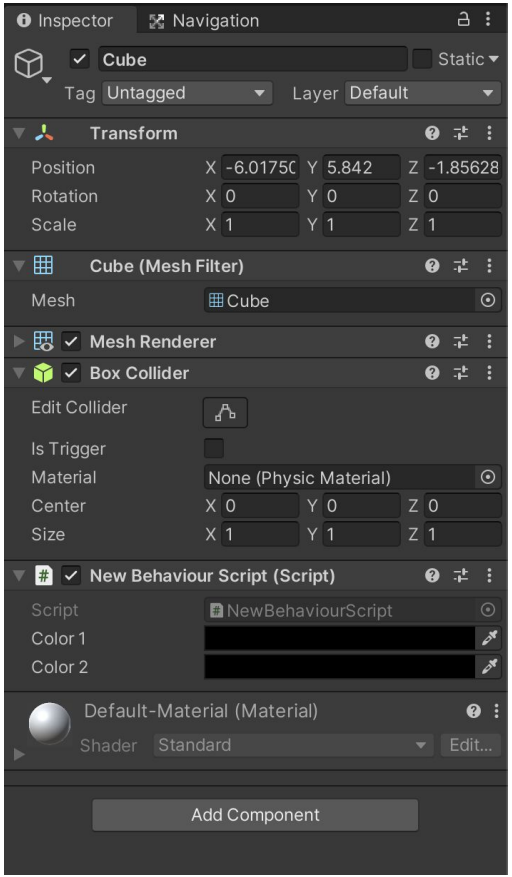
The Hierarchy window

- Displays every GameObject in a scene
- Parenting
  - Unity uses concept of parent-child hierarchies to group GameObjects
  - You can link GameObjects together to help move/scale/transform a collection of GameObjects
  - You can also create nested parent-child GameObject
- Organizing GameObjects
  - Create new child/parent/sibling GameObjects by dragging



The Inspector window

- View and edit properties and settings for almost everything in the unity editor
  - Game objects, Unity Components, Assets, appearance of your scene and in-Editor settings and preference
- Displays properties for the current selection (GameObjects, assets etc.) by default
- When GameObjects have custom script components attached, it displays the scripts' public variable



The Toolbar

Control:	Description:
	Use the Transform tools in the Scene view: - The first tool in the toolbar, the **Hand Tool**, allows you to pan around the Scene. - The **Move**, **Rotate**, **Scale**, **Rect Transform** and **Transform** tools allow you to edit individual GameObjects. Selected GameObjects also display a Gizmo in the Scene view if you have one of the four Transform tools selected.
	Toggling the Transform Gizmo affects the the Scene view.
	Use the Play, Pause, and Step buttons in the Game view.
	Launch Unity Collaborate from the **Collab** drop-down menu.
	Click the Cloud button to open the Unity Services window.
	Access your Unity Account from the **Account** drop-down menu.
	You can control which objects appear in **Scene** view from the Layers drop-down menu.
	You can change the arrangement of your views and then save the new layout or load an existing from the Layout drop-down menu.

The Status Bar

- A global progress bar for various asynchronous tasks
  - Clicking on it opens background tasks window

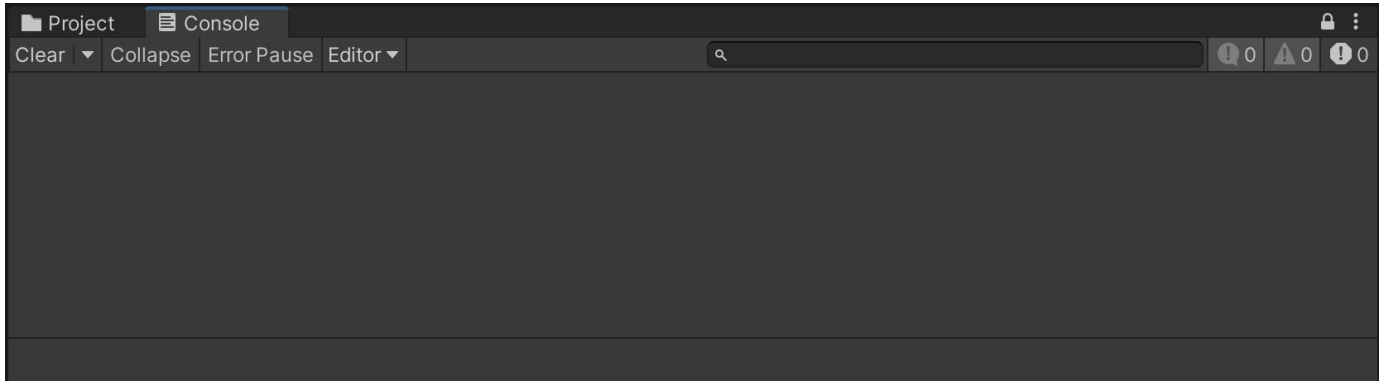
- Shows - Current code optimization mode
  - Debug mode (enables C# debugging)
  - Release mode (disables C# debugging)

- Shows - Cache server status
  - Cache server: a standalone app that you can run on your local computer that stores the imported asset data to reduce the time it takes to import assets

- Shows - Automatic lighting generation status for global illumination

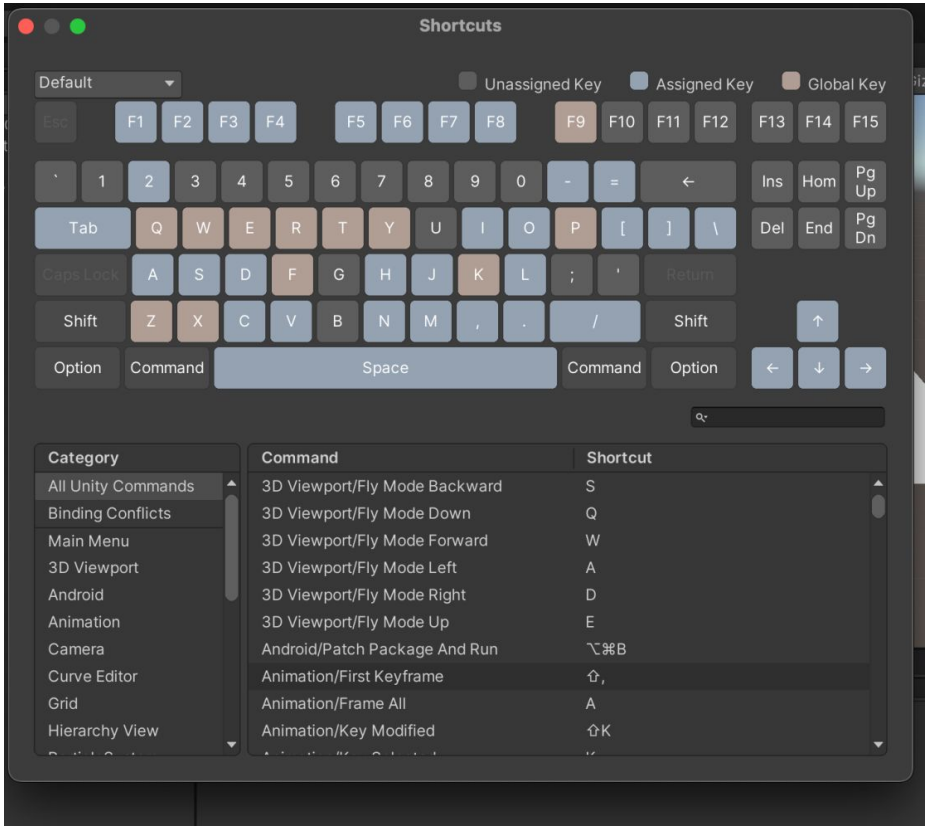
Console Window

- Shows errors, warnings and other messages generated by Unity
- Can also show your own messages using Debug class
- Everything that is displayed here is also written to Log file
- Open it using Window> General > Console



The Shortcut Manager

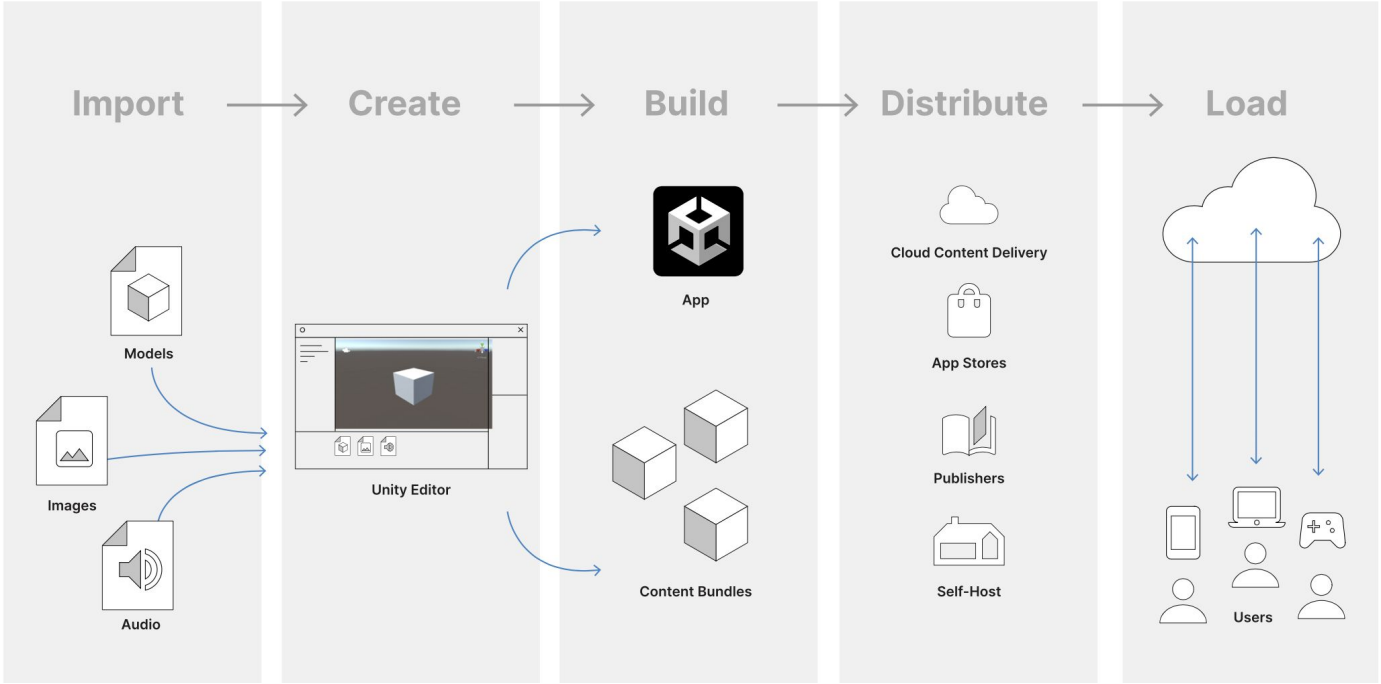
- Lets you view and manage keyboard shortcuts in Unity
- Edit > Shortcuts (Windows)
- Unity > Shortcuts (Mac)



Unity Assets

- Any item that you use in your Unity project to create your game or app
- Assets can represent visual or audio elements in your project, such as 3D models, textures
- An asset might come from a file created outside of Unity, such as a 3D Model, an audio file, or an image
- You can save or copy files that you want to use in your project into the Assets folder (can be viewed in the project window)
- Alongside external assets unity also offers creation of internal assets

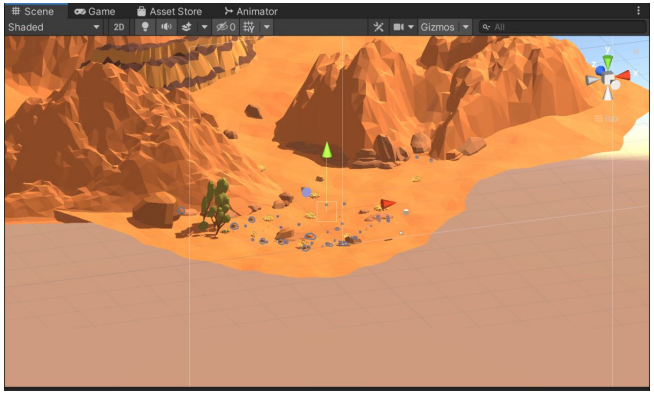
Assets



Asset Workflow

Unity Scenes

- Scenes are where you work with content in Unity
- They are assets that contain all or part of a game or application
- You can create any number of scenes in a project
- Unity has Scene Interface to edit a scene
- When you create a new project and open it for the first time, Unity opens a sample scene that contains only a Camera and a Light



Game Objects

- Fundamental objects in Unity that represents characters, props and scenery
  - They act as container for components, which implement the functionality
  - To give a game object the properties, you need to add components to it
  - Unity has lots of built-in component types
    - You can make your own using the ***Unity Scripting API***
    - For instance, Light object can be built by adding Light component to a game object

Game Objects

- A game object always has a Transform component attached to it (cannot be removed)
- Some primitive object types can be created directly within unity
  - Cube, Sphere, Capsule, Cylinder, Plane, Quard
- You can mark a game object as *inactive* to temporarily remove it from scene
- A tag can be assigned to one or more game objects to refer it in scripts

Prefabs

- Prefab system allows you to create, configure and store a game object
  - Prefabs act as a template from which you can create new prefab in the scene
  - In object oriented viewpoint, prefab can be considered as an object the user can make instances of (e.g., a certain type of bullet used multiple times around a level of a game)
  - It can be created by *dragging* a game object from the Hierarchy window into the Project window
  - You can *double click* the prefab to edit it in the prefab editing mode
    - Once one prefab is edited, the change will be applied to all game objects (instances) that belongs to the prefab

Graphics

- Unity’s graphics features let you control appearance of your application and are highly-customizable

- Important graphics support offered by unity
  - Render pipelines
  - Cameras and view transform
  - Lighting
  - Meshes, Materials, Textures, and Shaders
  - Creating environments
  - Sky
  - Visual Effects

Render Pipelines

- Unity provides three prebuilt render pipelines
  - **Built-in Render Pipeline**: a general-purpose render pipeline that has limited options for customization
  - **Universal Render Pipeline (URP)** : a Scriptable Render Pipeline that is quick and easy to customize, and lets you create optimized graphics across a wide range of platforms
  - **High Definition Render Pipeline (HDRP)** : a Scriptable Render Pipeline that lets you create cutting-edge, high-fidelity graphics on high-end platforms
- You can create your own custom render pipeline using Unity's Scriptable Render Pipeline API
- Our focus - Built-in Render Pipeline

Camera & View Transform

- Unity supports “taking snapshot” concept we learned in computer graphics (view transform)
  - It does this using *camera* concept

- In Unity, you create a camera by adding a *Camera* component to a game object

Lighting Effect

- With Unity, you can achieve realistic lighting for game objects
- Unity allow us to specify intensity, direction, and color of the light that falls on game objects
- We can even do more!

Shadows

- Shadows add a degree of depth and realism to a scene
- Unity uses a technique called *shadow mapping* to render real-time shadows
- You can configure shadow settings for each light component using **the Inspector interface**
- Each **Mesh Renderer** in the scene also has a *Cast Shadows* and a *Receive Shadows* property, which must be enabled as required
- Use the *Shadow Distance* property to determine distance from camera up to which Unity renders real-time shadows

Models

- Files containing data -- about shape and appearance of 3D objects
- Model files can contain a variety of data, including meshes, materials, and textures
- They can also contain animation data, for animated characters
- Unity supports a number of standard and proprietary model file formats
  - .fbx, .dae, .dxf, .obj

Mesh

- 3D Meshes are main graphics primitive of Unity
- Unity supports triangulated or quadrangulated polygon meshes
- The *Mesh Renderer* □ takes geometry from Mesh Filter □ renders it at the position defined by the game object's Transform component

Materials

- Meshes describe shapes; materials describe *appearance of surfaces*
- Materials and shaders are closely linked - we always use materials with shaders
- Materials are definitions of how a surface should be rendered, including references to textures used, tiling information, color tints and more
- Available options for a material depend on which shader the material is using

Shaders

- Small scripts that contain algorithms for calculating color of each pixel rendered, based on lighting input and the material configuration

Texture

- A bitmap image applied over the mesh surface
- A material may contain references to textures, so that material's shader can use the textures while calculating the surface color
- In addition to basic color of an obejct's surface, textures can represent many other aspects of a material's surface such as its *reflectivity* or *roughness*
- Texture dimensions represented as powers of 2
  - e.g. 32x32, 64x64, 128x128, 256x256, etc.
- Simply placing them in your project's Assets folder is sufficient, and they will appear in the Project View

Creating Environments

- Unity provides a selection of tools that let you create environmental features such as landforms and vegetation
  - Terrain Features in the Unity Editor - consist of basic tools that let you create and modify landscapes in the Unity
  - Terrain Tools preview package - provides additional functionality on top of the built-in Terrain features
  - Tree Editor - lets you design Trees directly in the Editor

Sky

- A type of background that a camera draws before it renders a frame

- You can use skybox to render a sky
  - Skybox - a cube with different texture on each face
  - Unity renders skybox first, so the sky always renders at the back

- Unity provides multiple Skybox Shaders
  - Each Shader uses a different set of properties and generation techniques
  - Each Shader falls into one of the **two** categories: Textured (6 sided, cubemap and panaromic) and Procedural

Visual Effects Components

- Visual effects can be applied to cameras, GameObjects, light sources, and other elements of your game

- To add visual effects, in the Inspector window go to Add Component > Effects

- Some visual effects are:
  - **Halo**: light areas around light sources, used to give the impression of small dust particles in the air
  - **Lens Flares**: simulate effect of lights refracting inside a camera lens
  - **Line Renderer**: takes an array of two or more points in 3D space, and draws a straight line between - can use to draw anything from a simple straight line to a complex spiral

Graphics API Support

- Unity supports DirectX, Metal, OpenGL, and Vulkan graphics APIs
  - Depending on availability of the API on a particular platform

- Unity uses a built-in set of graphics APIs, or the graphics APIs that you select in the Editor

Graphics API Support

- To use Unity's default graphics APIs:
  - Open the Player settings (menu: Edit > Project Settings, then select the Player category)
  - Navigate to Other Settings and make sure Auto Graphics API is checked
  - When Auto Graphics API for a platform is checked, the Player build includes a set of built-in graphics APIs and uses the appropriate one at run time to produce a best case scenario
  - When Auto Graphics API for a platform is not checked, the Editor uses the first API in the list
  - If the default API isn't supported by the specific platform, Unity uses the next API in the Auto Graphics API list

Physics

- Unity helps simulate physics in your Project to ensure that the objects correctly accelerate and respond to collisions
  - With help of *physics engine*
  - Includes concepts of Rigid bodies, Colliders, Joints, Physics articulations and Character Controllers

Rigid Bodies

- With a “Rigidbody” property attached, GameObject *respond* to gravity

- Need not use transform component
  - Should apply forces to push GameObject and let the *physics engine* calculate the results

- **Is Kinematic** property
  - Removes object from physics engine control and allow it to be moved kinematically from a script

Colliders

- Define shape of GameObject for the purpose of *physical collisions*
  - Can be primitive collider such as Box Collider, Sphere Collider and Capsule Collider
  - Or mesh collider - more processor intensive, but accurate

- You can add static colliders to a GameObject without a Rigidbody component to create floors, walls and other motionless elements of a Scene

- Friction and bounce of surface can be configured using *Physics Materials*

- A collider configured as a Trigger (using **Is Trigger** property) does not behave as a solid object and will simply allow other colliders to pass through

Joints

- Connects a Rigidbody to another Rigidbody, or to a fixed point in space
- Joints can apply forces that move rigid bodies, and joint *limits* can restrict that movement
- Can be useful to emulate
  - A ball and socket joint, like a hip or shoulder
  - Any skeletal joint
  - Doors and finger joints
  - A piece of elastic

Articulation

- Idea of physics articulations - to provide a realistic physics behavior applications that involve joints
- You can configure articulations via the ArticulationBody class, or the corresponding Articulation Body component
- A set of Articulation Bodies organized in a logical tree
- Each parent-child relationship reflects mutually constrained relative motion

Audio

- Unity has dedicated components for audio perception and playback

Audio Source

- Primary component *attached* to a game object to make it play a sound
- Plays back an AudioClip when triggered through mixer/code or by default
- AudioClip can be any standard audio file such as mp3, wav etc

AudioListener

- *Listens* to all audio playing in the scene and transfers to the computer speaker
- Acts like ears in the game
- All audio you hear is w.r.t positioning of this listener
- Only one AudioListener should be there in the scene - by default attached to Camera

Audio Filters

- Output of the AudioSource or intake of an AudioListener can be modified with audio filters – for various sound effects

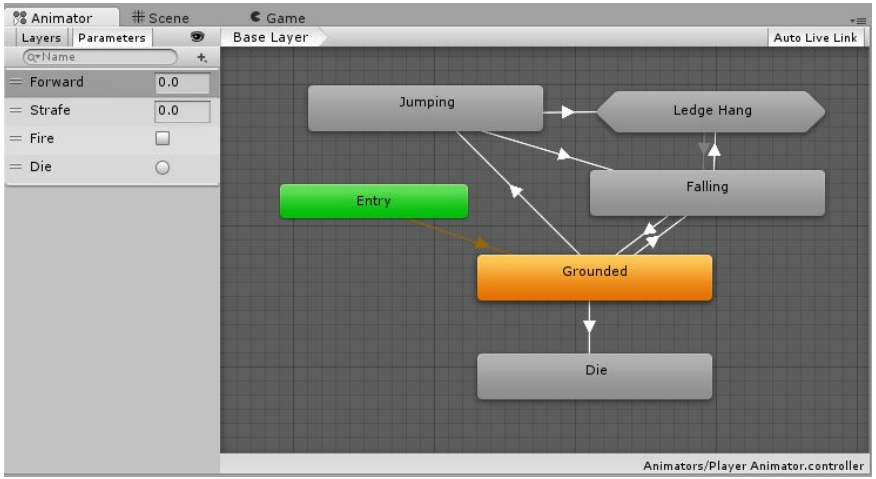
- Each filter comes as own component

Animation

- Unity has a rich and sophisticated animation system
  - Support for importing of animation clips and/or animation created within Unity
  - Humanoid animation retargeting
    - Ability to apply animations from one character model onto another
  - Simplified workflow for aligning animation clips
  - Convenient preview of animation clips, transitions and interactions between them
  - Management of complex interactions between animations with a visual programming tool
  - Animating different body parts with different logic

Animator Controllers

- Allows to arrange and maintain a set of animations for a character or other animated Game Object
- Animator Controller assets created from Assets menu, or from Create menu in Project window
- Animator controller applied to object by attaching an Animator component



User interfaces

- Unity provides three UI systems that you can use to create user interfaces (UI) for the Unity Editor
  - UI Toolkit
  - The Unity UI package (uGUI)
  - IMGUI

User interfaces

- UI Toolkit
  - Designed to optimize performance across platforms
  - Can be used to create extensions for Unity Editor, and to create runtime UI for games and applications

- Unity UI (uGUI) package
  - An older, GameObject-based UI system that you can use to develop runtime UI for games and applications
  - Can use components and Game view to arrange, position, and style the user interface
  - Supports advanced rendering and text features

- IMGUI
  - A code-driven UI Toolkit that uses OnGUI function and scripts to draw and manage user interfaces
  - Not recommended for building runtime UI

Setting Up Scripting Environment

- Integrated development environment (IDE) support
  - Unity supports: Visual Studio, Visual Studio Code, JetBrains Rider
- When you install Unity on Windows and macOS, by default Unity also installs Visual Studio
  - We will use visual studio in this lecture

Scripting Language – C#

- Unity supports C# (pronounced as C-Sharp) for scripting
  - An object-oriented programming language created by Microsoft
  - You can attach a C# script (as a component) to a game object
  - Allows you to trigger game events, modify component properties over time and respond to user input in any way you like
  - When creating a script, you are essentially creating your own new type of component that can be attached to Game Objects just like any other component

Creating Scripts

- Unlike most other assets, scripts are usually created within Unity directly
- Goto Assets > Create > C# Script from the main menu
- New script will be created in whichever folder you have selected in the Project panel
- Once attached, the script will start working when you press Play and run the game

Anatomy of a Script File

- When you double-click a script Asset in Unity, you can see the script

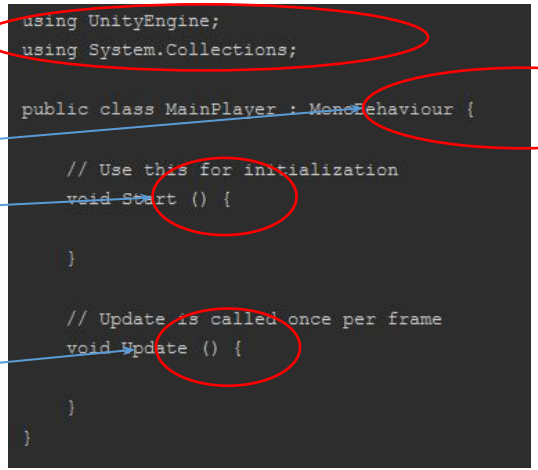
- Class name is same as file name

Name space

Inheriting from MonoBehaviour class

This function is first called whenever the script is called

This function is called in each frame



Variables and the Inspector

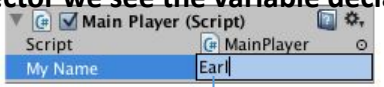
- Like any other programming language we can use different variables to store any value
- Just like other Components often have properties that are editable in the inspector, you can allow values in your script to be edited from the Inspector too
- We use Vector3 data type to represent the position of a 3D point

Script attached to a GameObject called Main Player

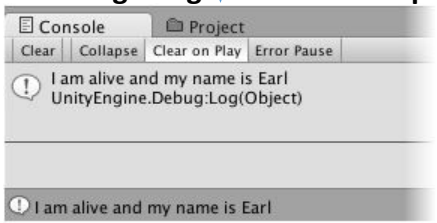
```
using UnityEngine;
using System.Collections;

public class MainPlayer : MonoBehaviour
{
    public string myName;

    // Use this for initialization
    void Start ()
    {
        Debug.Log("I am alive and my name is " + myName);
    }
}
```

Debug.Log is a simple command that just prints a message to Unity's console output

In inspector we see the variable declared in the script



After running the game we see output in console



Destroying an Object

- Destroy function comes under MonoBehaviour class

- You can attach GameObject and time (after how many second it should be destroyed) to it

Destroy(gameObject, 5f); Once it is called the gameObject is destroyed from the scene after 5 second

Functions

- Once we define a function we can use whenever it is needed

- The function will probably require some inputs or parameters, which are given to the function each time it is called

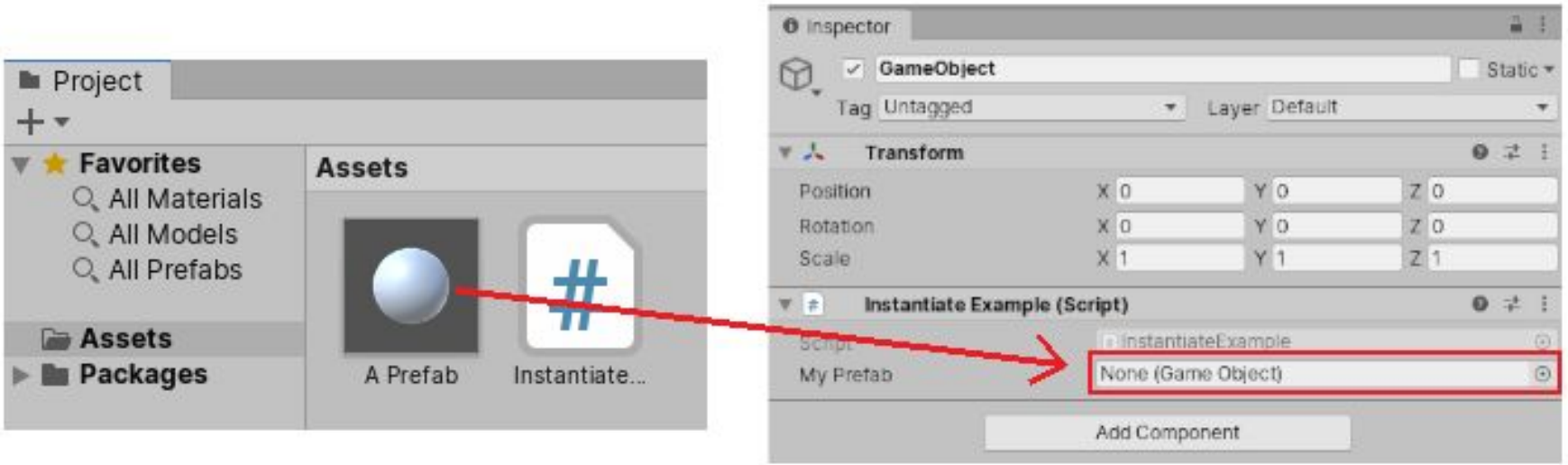
- Start() is a example of a function
  - However, we can define our own function

Event Functions

- Activated by Unity in response to events that occur during gameplay
- Some common and important events
  - **Regular Update Events**: e.g: Update(), FixedUpdate(), LateUpdate()
  - **Initialization Events:** e.g. Start(), Awake()
  - **GUI events:** OnGUI(), OnMouseOver(), OnMouseDown()
  - **Physics events:** OnCollisionEnter(), OnCollisionStay() and OnCollisionExit() (When contact is made/held or broken); OnTriggerEnter, OnTriggerStay and OnTriggerExit (Called when collider is configured as Trigger)
- Unity passes control to a event function when “events” happen
- Once an event function has finished executing, control is passed back to Unity

Instantiating Prefabs at Run Time

- To instantiate a Prefab at run time, your code needs a reference to that Prefab



Instantiating Prefabs at Run Time

```
using UnityEngine;
public class InstantiationExample : MonoBehaviour
{
    // Reference to the Prefab. Drag a Prefab into this field in the Inspector.
    public GameObject myPrefab;

    // This script will simply instantiate the Prefab when the game starts.
    void Start()
    {
        // Instantiate at position (0, 0, 0) and zero rotation.
        Instantiate(myPrefab, new Vector3(0, 0, 0), Quaternion.identity);
    }
}
```

- To use this example:
  - Create a new C# script in your Project, and name it “InstantiationExample”
  - Copy and paste above into your new script, and save it
  - Create an empty GameObject using the menu GameObject > Create Empty
  - Add the script to the new GameObject as a component by dragging it onto the empty GameObject
  - Create any Prefab, and drag it from the Project window
  - into the My Prefab field in the script component

If Else Statements

```
public class NewBehaviourScript : MonoBehaviour
{
    int count;
    // Start is called before the first frame update
    void Start()
    {
        count = 1;
    }

    // Update is called once per frame
    void Update()
    {
        if (count > 100)
        {
            Debug.Log("Frame is updated more than 100 times ");
        }
        else
        {
            Debug.Log("Not yet 100");
            count++;
        }
    }
}
```

Loops

- Loops can be entry controlled or exit controlled
  - Entry controlled: for, while
  - Exit controlled: do-while (loop body will be evaluated for at-least one time as the testing condition is present at the end of loop body)
- For arrays (collection of homogeneous data type) foreach loop can be used

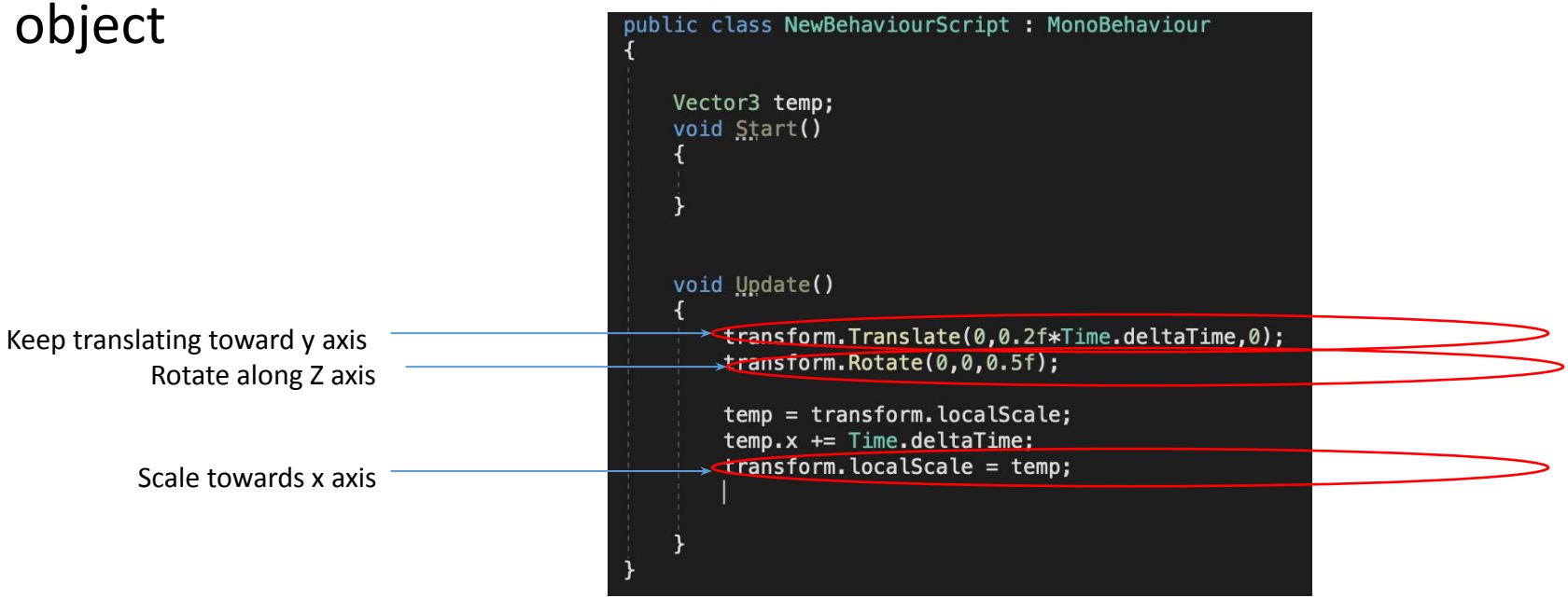
```
void Start()
{
    for (int i = 0; i < 5; i++)
    {
        Debug.Log(i);
    }

}
```

```
int i = 0;
void Start()
{
    do
    {
        Debug.Log(i);
    } while (i < 5);

}
```

```
public int[] array; //this will be visible in the inspector
void Start()
{
    foreach(int i in array)
    {
        Debug.Log(i);
    }

}
```

Transform

- Every object in a scene has a Transform property attached
- It's used to store and manipulate position, rotation and scale of object



Co-routines

- A co-routine allows you to spread tasks across several frames
  - A method that can pause execution and return control to Unity but then continue where it left off on the following frame
- Important - co-routines aren’t threads
  - Synchronous operations that run within a co-routine still execute on the main thread

Co-routines

As an example, consider the task of gradually reducing an object's alpha (opacity) value until it becomes invisible

```
void Fade()
{
    Color c = renderer.material.color;
    for (float alpha = 1f; alpha >= 0; alpha -= 0.1f)
    {
        c.a = alpha;
        renderer.material.color = c;
    }
}
```

The Fade method doesn't have the effect you might expect. To make the fading visible, you must reduce the alpha of the fade over a sequence of frames to display the intermediate values that Unity renders

```
IEnumerator Fade()
{
    Color c = renderer.material.color;
    for (float alpha = 1f; alpha >= 0; alpha -= 0.1f)
    {
        c.a = alpha;
        renderer.material.color = c;
        yield return null;
    }
}
void Update()
{
    if (Input.GetKeyDown("f"))
    {
        StartCoroutine(Fade());
    }
}
```

You could add code to the Update function that executes the fade on a frame-by-frame basis

InvokeRepeating

- Invokes method methodName in time seconds, then repeatedly every repeatRate seconds
- Example - LaunchProjectile function starts in 2 seconds of calling and a projectile will be launched every 0.3 seconds

```
public Rigidbody projectile;

void Start()
{
    InvokeRepeating("LaunchProjectile", 2.0f, 0.3f);
}

void LaunchProjectile()
{
    Rigidbody instance = Instantiate(projectile);

    instance.velocity = Random.insideUnitSphere * 5;
}
```

Mouse/Keyboard Input

- Input.GetKey(KeyCode), Get.KeyUp(KeyCode), Input.GetKeyDown(KeyCode) is used
  - If you want to detect key “a” from the keyboard use KeyCode.a
- GetMouseButtonDown
  - GetMouseButtonDown(code) is used to detect the mouse click
    - Code 0,1 and 2 means left, right and middle click of the mouse

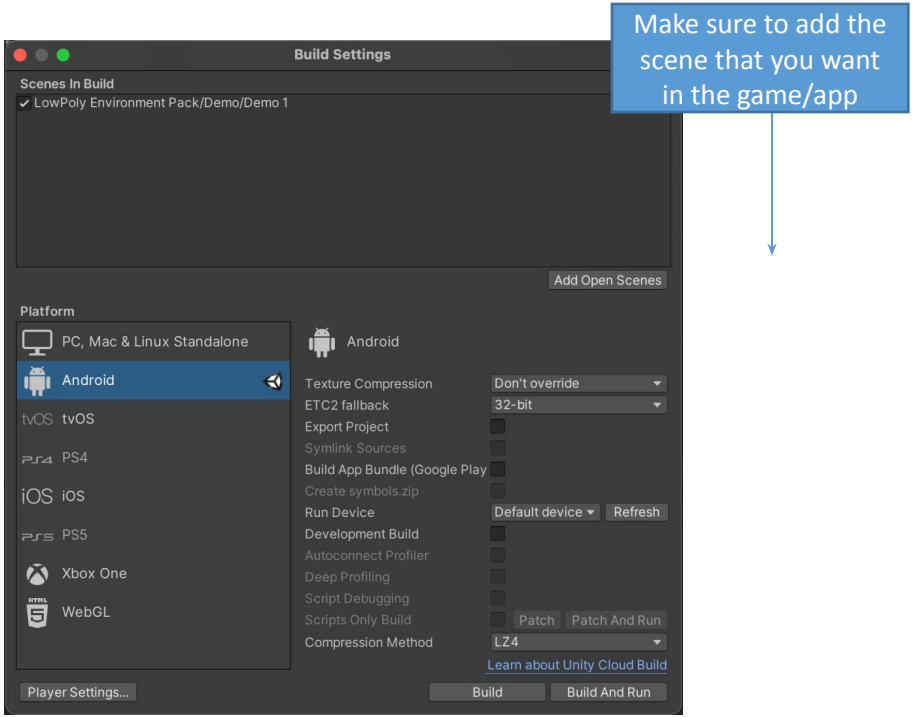
GetButton

- There are some keys/button mapped with some name
  - You can find it from edit> Project Settings> Input
- Input.GetButton can be used to detect that button name

```
if (Input.GetButton("Fire1") && myTime > nextFire)
{
    nextFire = myTime + fireDelta;
    newProjectile = Instantiate(projectile, transform.position, transform.rotation) as GameObject;

    // create code here that animates the newProjectile

    nextFire = nextFire - myTime;
    myTime = 0.0F;
}
```

Publishing Builds

- Build settings window contains all settings and options to publish your build
  - Go to File> Build Settings
  - *Scenes in build panel* is used to manage which scenes unity includes in the build
  - *Platform* section of the window to select which platform you want to build to
  - Select the *Build* or *Build And Run* button to begin the build process



Thank You