# 1 Scripting Overview

The whole note is devoted to introduce some special techniques towards engine.

## 1.1 Creating and Using Script

The GameObject is controlled by the *Component*, and script is used to modify and customize.

### 1.1.1 How to create script

### 1.1.2 Anatomy of Script File

Start(): It's called by Unity before gameplay begins.
Update(): Handling the frame update for the GameObject
The constructor is not needed as it's handled by the editor and may not take place at the start of gameplay.

### 1.1.3 Controlling a GameObject

Script instance must be attached to a GameObject to control.
Debug.Log("message") can output to the console in the engine.

## 1.2 Variables and the Inspector

Variables are declared public to be access in the Inspector. You can also make changes to the variables during the gameplay temporarily.

## 1.3 Controlling GameObject Using Component

### 1.3.1 Accessing Components

*GetComponent()* function can be used to get reference to the component properties.

```
RigidBody rb = GetComponent<RigidBody>();
```

you can use it to access the parameter and manipulate the instances.

### 1.3.2 Accessing other object

You can either link objects with other variables with public gameObject, which is similar to the public variables. You can also declare a component and drag a other gameObject on it to attach component of other gameObject. This operation must be done in the editor rather than runtime.

**Finding Child Objects** and **Finding Objects by Name or Tag** can be used to locate gameObject in the runtime.

Finding Child Objects: when you want to keep track same kind of objects such as location of waypoints, you may make the child of a logical parent and can access them using foreach.

```
public gameObject wayPoints;
...
void Update() {
    int num = wayPoints.childCount;
    ...
    foreach(Transform t in wayPoints){
        wayPointArray[i ++] = t;
    }
    ...
    particularChild = wayPoints.Find("start");
}
```

Finding Objects by Name or Tag:

```
\\GameObject[] gameObject_3;
gameObject_1 = GameObject.Find("Car");
gameObject_2 = GameObject.FindWithTag("Player");
gameObject_3 = GameObject.FindGameObjectsWithTag("Enemy");
```

## 1.4 Event Functions

**Event Functions** are activated by Unity in response to events that occur during game play, such as Update() and Start().

### 1.4.1 Regular Update Events

**Update()** is the main function to make changes, taking place before each frames is rendered and each animations is calculated.

**FixedUpdate()** is called before each physics update. This function is dedicated to the change of physics.

**LateUpdate()** can be able to make additional changes after Update() and FixedUpdate(). It can be used to adjust camera and character orientation.

### 1.4.2 GUI events

Unity has a system for rendering GUI controls over the main action in the scene and responding to clicks on these control. These code should be included in *OnGUI()* function.

```
void OnGUI() {
    GUI.Label(labelRect, "Game Over"); //making a text or texture on screen
    /*
    Prototype: GUI.Label(Rect Position, string text);
               GUI.Label(Rect Position, Texture image);
    Rect: A 2D rectangle defined by left upper X and Y position, wight, height.
          It can also be constructed by xMin, xMax, yMin, yMax.
    */
}
```

You can also detect *mouse events* that occur over a GameObject as it appears in the scene, such as targeting weapons. *OnMouseDown()* is called when user has pressed the mouse button while over the GUIElement or Collider. A call to *OnMouseOver()* occurs on the first frame the mouse is over the object until the mouse moves away, at which point *OnMouseExit()* is called. It needs the collider to be marked as trigger. More specific mouse and keyboard operations can go to look up for Monobehaviour.

### 1.4.3 Physics events

A series of function will be called when collision is detected, such as *OnCollisionEnter()*. When collider is marked as trigger, it turns into *OnTriggerEnter()*. Object may contact with multiple objects, which requires for a parameter to identify.

## 1.5 Time and Framerate Management

The Update() allows monitoring inputs and other events from the script. A important thing is that neither framerate or length of time between Update() is called is constant. *Time.deltaTime* is the time in seconds it took to complete the last frame. It remains constant when called from FixedUpdate(). In order to subtract or add for a constant per frame, multiply it by Time.deltaTime can guarantee a constant offset per frame. For example:

```
void Update() {
    Transform.position = (0.0f, 0.0f, distancePerSeconds * Time.deltaTime);
}
```

### 1.5.1 Fixed Timestep

There is a fixed timestep between FixedUpdate() for physics engine. You can change it in *Time Manager* and read from *Time.fixedDeltaTime* in order to get better physical performance.

### 1.5.2 Maximum Allowed Timestep

If fixed timestep is too small that gameplay framerate is relatively low, many physical update will in one framerate update, causing even lower fresh rate and unsynchronized movement. With the set of *maximum allowed timestep* in *time manager*, when the rendering time of one frame exceeds it, the physical engine will stop calculating and staying at that conditions waiting for the update of frame. It's a trade off that causes the movement slower than the real world while having better framerate.

### 1.5.3 Time Scale

**timeScale** can be used to slow down or freeze the speed of game for bullet-time or pause. The value of it means the rate of speed. It will affect all the time and delta time measuring variables of the Time class except for *realTimeSinceStartUp*.

```
void Pause() {
    Time.timeScale = 0;
}
void Resume() {
    Time.timeScale = 1;
}
```

### 1.5.4 Capture Framerate

**Capture Framerate** property is devoted to record the image or video of your gameplay duration. When captureFramerate is other than 0, the frame update rate will be set to have an interval of $\frac{1}{CFR}$ to have time to save images. It can be set to 25fps for video.

```
string folder = "ScreenshotFolder";
public int frameRate;
void Start() {
    Time.captureFramerate = 1;
    System.IO.Directory.CreateDirectory(folder);
}

void Update() {
    string name = string.Format("{0}/{1:D04} shot.png", folder, Time.frameCount);
    Application.CaptureScreenshot(name);
}
```

## 1.6 Creating and Destroying GameObject.

GameObject can be created by *Instantiate()* to make new copy of a gameObject, usually from prefab. *Destroy()* is used to destroy a object with an allowed lifetime. Destroy() be used to destroy component .

```
Destroy(this); //This is used to destroy the script itself
```

## 1.7 Coroutine

**Coroutine** is function that has the ability to pause execution and return control to Unity but then continue where it left off on the following frame. It's essential a function to be declared with a return type of *IEnumerator* and function will pause at the point of *yield return line*. It will resume on the frame after it's yield by default but you can set a delay time with *WaitForSeconds()*. The coroutine function can be used to spread an effect over a period of time and also can perform regular check and update which has a bigger interval than Update() interval. Coroutine functions will finally stop when all *yield instructions* finishes.

```
void Start() {
    StartCoroutine(Spawn);
}

IEnumerator Spawn() {
    ...
    yield return WaitForSeconds(waitPlayer);
    ...
}
```

Coroutine are not stopped if the Monobehaviour is disabled. Coroutine can be stopped by either *Monobehaviour.StopCoroutines* or *Monobehaviour.StopAll-Coroutine*. It's also stopped when Monobehaviour is destroyed.

## 1.8 Namespaces

Just like c++, the class in a namespace can be accessed by adding a prefix. It allows same class name within different namespaces. You can avoid typing the prefix by *using* directive at the top of the script.

## 1.9 Attributes

**Attributes** are makers that can be placed over a class, property, or a function in a script to indicate special behavior like [HideInspector], [System.Serialization];

## 1.10 Execution order of Events Functions

### 1.10.1 Editor

**Reset()**: Reset is called to initialize the script's property when it's first attached to the object all Reset() is called.

### 1.10.2 First Scene Loaded

**Awake()**: It's called always before Start() functions and also just after a prefab is instantiated and set active.

**OnEnable()**(active required): This function is called just after the object is enabled when level is loaded or instanciated.

They can't be enforced when an object is instantiated during gameplay.

### 1.10.3 Before the First Frame Update

**Start()**: Start is called before the first frame update on if the script instance is enabled.

### 1.10.4 In Between Frames

**OnApplicationPause(bool)**: This is called at the end of frame where the pause is detected, effectively between the normal frame updates. One extra frame will be issued indicating obvious pause state. It is used in mobile game.

### 1.10.5 Update Order

**FixedUpdate()**: It's often called more frequently than Update(), all physical updates will be done right after FixedUpdate(). There is no need for Time.timeDelta because FixedUpdate() has a constant fresh rate.

**Update()**: It's called once per frame.

**LateUpdate()** It's called after Update() is done. The most common use of it is third-person camera.

### 1.10.6 Rendering

**OnPreCull()**:Called before the camera culls the scene. Culling determines which an object becomes visible/invisible to the camera.

**OnBecomeVisible()/ OnBecomeInvisible()**: Called when an object becomes visible/invisible to any camera.

**OnWillRenderObject()**: Called once for each camera if the object is visible.

**OnPreRender()**: Called before the cameras starts to render the scene.

**OnRenderObject()**:Called after all regular scene rendering is done with GL class or Graphics.DrawMeshNow.

**OnPostObject()**:Called after a camera finishes rendering the scene.

**OnRenderImage()**:Called after scene is complete to allow for post-processing of the image. *Post Processing* is the process of applying full-screen filters and effects to the camera's image buffer.

**OnGUI()**:Called multiple times per frame in the response to *GUI events*, the *layout* and *repaint* is the first, followed by a layout and keyboard/mouse event for each input event.

**OnDrawGizmos**:Used for drawing Gizmos in the scene view for visualization.

#### 1.10.7 Coroutine

Normal Coroutines updates are run after the Update function returns. There are different types of yield instructions:

**yield**:The coroutine will continue after all Update functions have been called on the next frame.

**yield return WaitForSeconds()**:Continue after a specific delay after all Update() have been called on the next frame.

**yield return WaitForFixedUpdate()**: Continue after all FixedUpdate has been called on the next frame.

**yield return WWW**:Continue after a WWW download complete.

**yield StartCoroutine()**: chains the routine, and will wait for the MyFun coroutine to complete first.

#### 1.10.8 When the Object is destroyed

**OnDestroy()**:The function is called after all frame updates for the last frame of the object's existence.

#### 1.10.9 When Quitting

These functions get called on all the active objects in the scene:

**OnApplicationQuit()**: This is called before all the application is quit and when user stops play mode.

**OnDisable()**: This function is called when the behavior becomes disable or inactive.

#### 1.10.10 Script LifeCycle Flowchart

Please refer to https://docs.unity3d.com/Manual/ExecutionOrder.htm.

### 1.11 Understanding Automatic Memory

The memory required to store it is allocated from a central pool called *heap*. When it is no longer used, they will be reclaimed and used for something else.

Nowadays automatic management has decreases the coding effort and reduce the opportunity of memory leakage.

### 1.11.1 Value and Reference

The pass of the parameter can be done by creating a instance in a memory block or a pointer to the location of the value. Type that directly stored with parameter is called *value type*. Type that are allocated on the heap and then accessed via a pointer are called reference types.

### 1.11.2 Allocation and Garbage Collection

The memory manager will track all the unused memory and allocate it to incoming request. A reference item on the heap can only be accessed as long as there are still reference to it. If all references to a memory block has gone, it will be reallocated.

**Garbage Collection**(GC) is the process that search through all active references and locate unused memory.

### 1.11.3 Optimization

Improper coding will make automatic GC a disaster. For example :

```
void Example() {
    string line = inArray[0].Tostring();
    for(int i = 0; i < inArray.Length(); ++ i) {
        line += "," + inArray[i].ToString();
    }
}
```

In the example above, a completely new string wil be allocated which may cause hundreds of bytes of memory blocks to be freed every loop. It's better to use *System.Text.StringBuilder()*.

Another example is that the Text object to be updated every Update(). In order to optimize, we will update the Text when its content is changed.

```
float[] RandomList() {
    var result = new float[[numElements];
    for(int i = 0; i < numElements; ++ i) {
        result[i] = Random.value;
    }
    return result;
}
```

In order to reduce the time of reallocate, it's recommended to use reference array rather than creating a new one as above.

### 1.11.4 Requesting a Collection

It's best to avoid allocations as far as possible. As they can't be eliminated, there are two strategies: *Small heap with fast and frequent GC*; *Large heap with slow but infrequent GC*.

The first strategy is often best for games that have long periods of gameplay where a smooth framerate in the main concern. It may make more collections than necessary, but still ha