**List out Unity LifeCycle**

* Awake
* OnEnable
* Onstart
* FixedUpdate
* Update
* LateUpdate
* OnGUI
* OnDisable
* OnDestroy

**List out UI Elements**

* Image
* Panel
* Button
* Slider
* Scollerview
* Text
* RawImage
* Canvas
* DropDown
* InputField

## **Awake**

* Awake is called either when an active GameObject that contains the script is initialized when a Scene loads
* Awake is called even if the script is a disabled component of an active GameObject.
* Unity calls Awake only once during the lifetime of the script instance.
* Awake is always called before any Start functions.
* Awake can not act as a coroutine.

**Start**

* Start is called on the frame when a script is enabled just before any of the Update methods are called the first time.
* Start is called exactly once in the lifetime of the script.
* Start may not be called on the same frame as Awake if the script is not enabled at initialisation time.
* The Start function can be defined as a [Coroutine](https://docs.unity3d.com/ScriptReference/Coroutine.html), which allows Start to suspend its execution (yield).

## **Different Start and Awake**

| Awake | Start |
| --- | --- |
| Awake is always called before any Start functions | Start is called on the frame when a script is enabled just before any of the Update methods is called the first time**.** |
| Awake is called when script is disable. | Start may not be called on the same frame as Awake if the script is not enabled at initialisation time. |

# 

# **FixedUpdate**

## Frame-rate independent FixedUpdate message for physics calculations.

* it is called every fixed frame-rate frame.
* FixedUpdate is usually used for physics calculations since it has the same frequency as the physics system.
* Compute Physics system calculations after FixedUpdate. 0.02 seconds (50 calls per second) is the default time between calls.

# **Update**

* Update is called every frame, if the MonoBehaviour is enabled.
* If the frame rate is 60 FPS, it will execute 60 times a second; if it’s 30, it will be 30 times a second.
* Update is the most commonly used function to implement any kind of game script. Not every MonoBehaviour script needs Update.

# **LateUpdate**

* LateUpdate is called every frame, if the Behaviour is enabled.
* It will execute after all the update functions are called, and it’s recommended to use LateUpdate on camera scripts instead of Update since it can keep track of objects that have already been moved in an Update function.
* Order of execution between Update and LateUpdate has nothing to do with speed.

# **Transform**

* Position, rotation and scale of an object.
* Every object in a scene has a Transform. It's used to store and manipulate the position, rotation and scale of the object.
* Every Transform can have a parent, which allows you to apply position, rotation and scale hierarchically.

# **RectTransform**

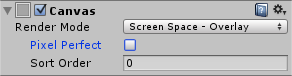
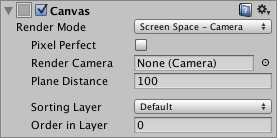
* Position, size, anchor and pivot information for a rectangle.
* RectTransforms are used for GUI but can also be used for other things.
* It's used to store and manipulate the position, size, and anchoring of a rectangle and supports various forms of scaling based on a parent RectTransform.

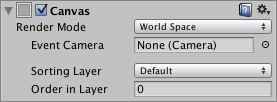
**Difference between Transform and RectTransform**

| **Transform** | **RectTransform** |
| --- | --- |
| Transform is a component for defining a GameObject’s relative position, rotation, and scale. | RectTransform is a Transform, but with extra info for managing an object as a UI element. |
| Most objects added to a scene get this assigned automatically. | Objects added to the scene through the Unity menu creating a UI object will have RectTransform automatically, as will any empty objects added as children to that object. |

## **Canvas**

* The **Canvas** component represents the abstract space in which the UI is laid out and rendered. All UI elements must be children of a GameObject that has a Canvas component attached.
* The **Canvas** is the area that all UI elements should be inside. The Canvas is a Game Object with a Canvas component on it, and all UI elements must be children of such a Canvas.
* Creating a new UI element, such as an Image using the menu **GameObject > UI > Image**, automatically creates a Canvas, if there isn't already a Canvas in the scene. The UI element is created as a child to this Canvas.
* The Canvas area is shown as a rectangle in the Scene View. This makes it easy to position UI elements without needing to have the Game View visible at all times.



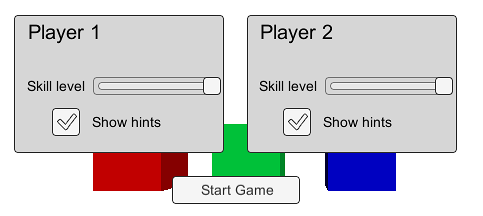


## 

## 

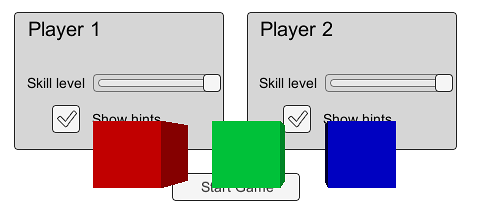
## **Screen Space - Overlay**

* The Canvas is scaled to fit the screen and then rendered directly without reference to the scene or a camera (the UI will be rendered even if there is no camera in the scene at all).
* If the screen's size or resolution are changed then the UI will automatically rescale to fit. The UI will be drawn over any other graphics such as the camera view.



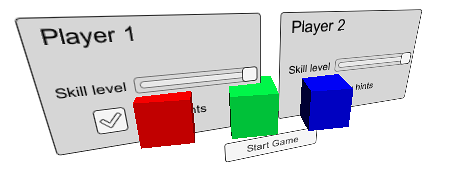
**Screen Space - Camera**

* The Canvas is rendered as if it were drawn on a plane object some distance in front of a given camera.
* The onscreen size of the UI does not vary with the distance since it is always rescaled to fit exactly within the camera frustum.
* If the screen's size or resolution or the camera frustum are changed then the UI will automatically rescale to fit.
* Any 3D objects in the scene that are closer to the camera than the UI plane will be rendered in front of the UI, while objects behind the plane will be obscured.



**World Space**

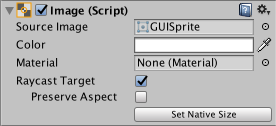
* This mode renders the UI as if it were a plane object in the scene. Unlike *Screen Space - Camera* mode, however, the plane need not face the camera and can be oriented however you like.
* The size of the Canvas can be set using its Rect Transform but its onscreen size will depend on the viewing angle and distance of the camera. Other scene objects can pass behind, through or in front of the Canvas.



## 

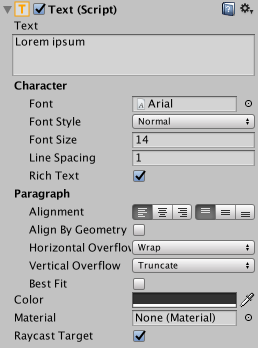
## **Image**

* The **Image** control displays a non-interactive image to the user.
* You can use this for purposes such as decorations or icons, and you can change the image from a script to reflect changes in other controls.



# **Text**

* The **Text** control displays a non-interactive piece of text to the user.
* This can be used to provide captions or labels for other GUI controls or to display instructions or other text.



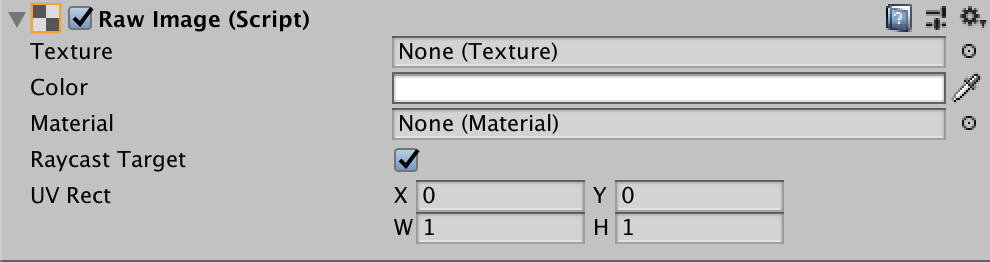
# 

# **TextMesh Pro**

* TextMesh Pro is an easy-to-use system for high-quality text.
* It has many text appearance and formatting options, and is an easy way to add a professional touch to any project’s user interface.
* TextMeshPro provides Improved Control over text formatting and layout with features like character, word, line and paragraph spacing, kerning, justified text, Links, over 30 Rich Text Tags available, support for Multi Font & Sprites, Custom Styles and more.

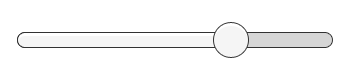
# **Raw Image**

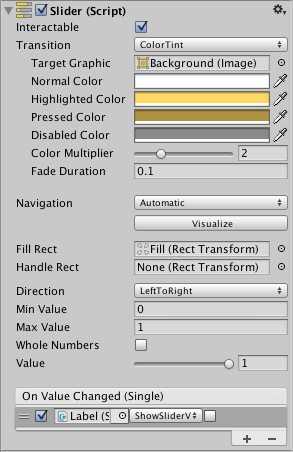
* The **Raw Image** control displays a non-interactive image to the user.
* You can use this for purposes such as decorations or icons, and you can change the image from a script to reflect changes in other controls.
* The control is similar to the Image control, but offers more options for animating the image and accurately filling the control rectangle.



**Slider**

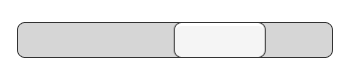
* The **Slider** control allows the user to select a numeric value from a predetermined range by dragging the mouse.

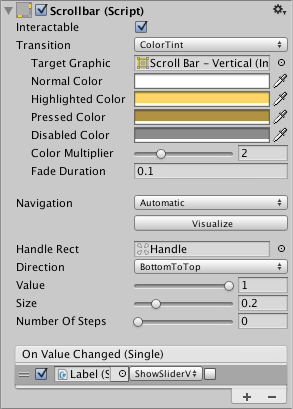




# **Scrollbar**

* The **Scrollbar** control allows the user to scroll an image or other view that is too large to see completely.



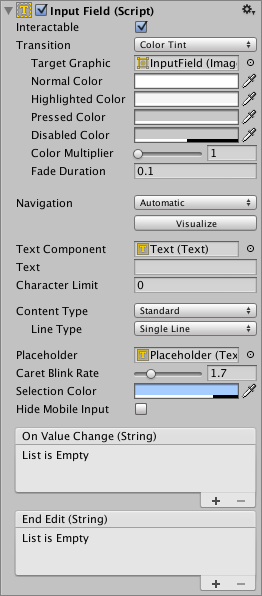


# **Input Field**

* An **Input Field** is a way to make the text of a Text Control editable.
* It's not a visible UI element in itself and must be combined with one or more visual UI elements in order to be visible.

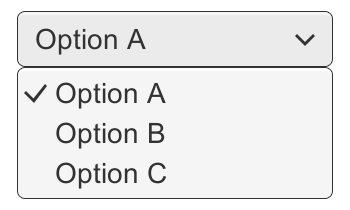


# 



# **Dropdown**

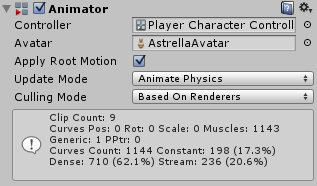
* The **Dropdown** can be used to let the user choose a single option from a list of options.
* The control shows the currently chosen option. Once clicked, it opens up the list of options so a new option can be chosen.

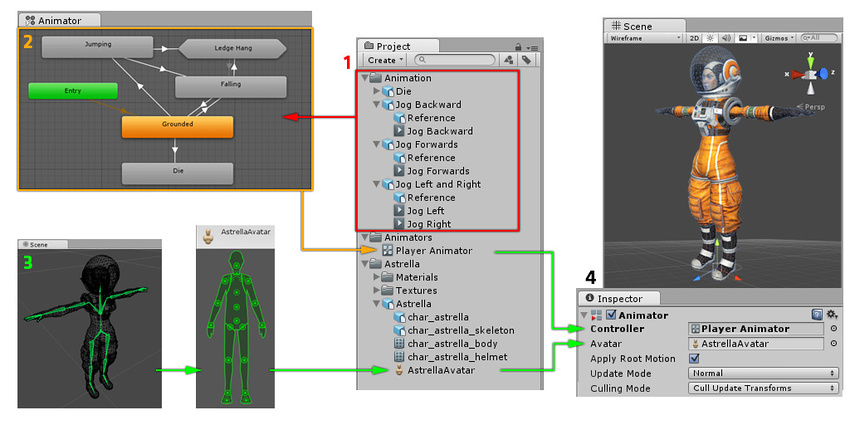


## 

# **Animator**

* Interface to control the Mecanim animation system.
* Use the **Animator component** to assign animation to a **GameObject**  in your **Scene**.
* The Animator component requires a reference to an Animator Controller which defines which **animation clips** to use, and controls when and how to blend and transition between them.



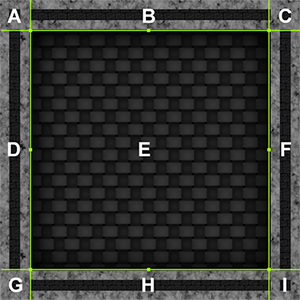
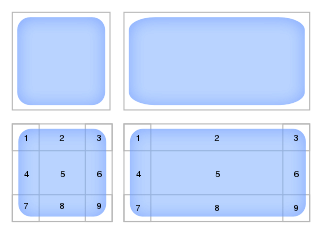


# **Sprite**

* Sprites are 2D graphic objects used for characters, props, projectiles and other elements of 2D gameplay.
* The graphics are obtained from bitmap images - Texture2D.
* The Sprite class primarily identifies the section of the image that should be used for a specific sprite.

# **9-slicing Sprites**

* 9-slicing is a 2D technique which allows you to reuse an image at various sizes without needing to prepare multiple Assets.
* It involves splitting the image into nine portions, so that when you resize the Sprite the different portions scale or tile (that is, repeat in a grid formation) in different ways to keep the Sprite in proportion.

# **Camera**

* A Camera is a device through which the player views the world.
* A screen space point is defined in pixels. The bottom-left of the screen is (0,0); the right-top is (pixelWidth,pixelHeight). The z position is in world units from the Camera.
* A viewport space point is normalized and relative to the Camera. The bottom-left of the Camera is (0,0); the top-right is (1,1). The z position is in world units from the Camera.
* A world space point is defined in global coordinates.

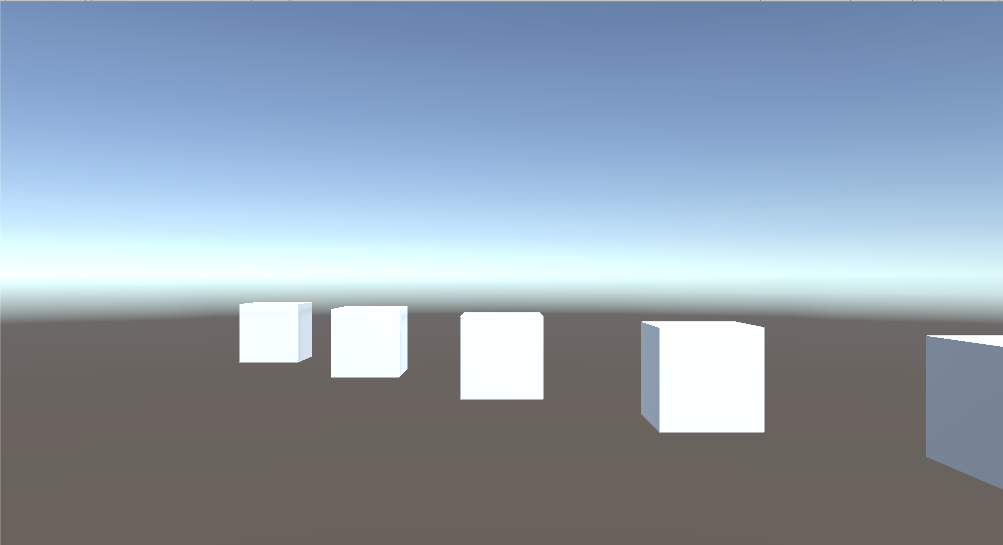
# **Orthographic Cameras**

* An object viewed in an Orthographic Camera will always appear the same size.
* A two-unit-high object will appear twice as tall as a one-unit-high object, regardless of where they are in relation to each other and to the Camera.
* Orthographic camera is well-suited for 2D games, isometric views, and certain UI elements. If you are creating a 3D game that requires perspective, you would typically use a perspective camera instead.
* The size of an Orthographic Camera specifies how many Unity units fit vertically between the Camera and either the top or bottom edge of the screen.

# 

# **Perspective Cameras**

* A perspective camera is a type of camera that simulates real-world perspective by making objects that are farther away from the camera appear smaller.
* It is commonly used for most 3D games and provides a sense of depth and realism to the scenes.
* Where the Orthographic Camera’s coverage is specified by a size, the Perspective Camera has a field of view.
* Perspective cameras are most often used in 3D projects, such as games or walkthroughs, or anywhere the view needs to resemble that of a real camera.
* Perspective cameras are also sometimes used in 2D projects for the automatic parallax effect they offer.



# **Playerpref**

* PlayerPrefs is a class that stores Player preferences between game sessions.
* It can store string, float and integer values into the user’s platform registry.
* Unity stores PlayerPrefs in a local registry, without encryption. Do not use PlayerPrefs data to store sensitive data.
* This can be extremely useful for saving and loading basic sets of data, such as option settings or simplified game saves.
* The Player Prefs values are stored locally on the device in a preferences file, meaning that they will exist after the game is closed.

# **ScriptableObject**

* A ScriptableObject is a data container that you can use to save large amounts of data, independent of class instances.
* Scriptable Objects are a type of class in Unity that can be used to create multiple instances of a script that each contain their own unique values.
* Once the asset is created, you can access and modify its properties just like any other asset in your project. Any changes you make to the ScriptableObject will be saved as part of the asset.
* One of the main use cases for ScriptableObjects is to reduce your Project’s memory usage by avoiding copies of values. This is useful if your Project has a Prefab that stores unchanging data in attached MonoBehaviour scripts
* Every time you instantiate that Prefab, it will get its own copy of that data. Instead of using the method, and storing duplicated data, you can use a ScriptableObject to store the data and then access it by reference from all of the Prefabs. This means that there is one copy of the data in memory.

# **Mouse events**

* Mouse events occur when you interact with the **UI** using a mouse.
* Touch, pens, or other pointing devices generate other events, not mouse events.
* Mouse events are always preceded by the corresponding PointerEvent.
* There are set of Mouse Event Methods as part of the MonoBehaviour class; which allow us to handle simple operations or logics based on Mouse inputs.

| **Events** | **Description** |
| --- | --- |
| OnMouseEnter() | is called as soon as the mouse pointer enters a game object's bounding area |
| OnMouseOver() | is called as long as the mouse pointer is over a game object's bounding area |
| OnMouseDown() | is called as soon as the left mouse button is pressed on a game object's bounding area |
| OnMouseDrag() | is called as long as the left mouse button is pressed and held down over the screen |
| OnMouseUp() | is called as soon as the left mouse button is released on the same object or anywhere else on the screen |
| OnMouseUpAsButton() | is called as soon as the left mouse button is released on the same object on which left mouse button was pressed |
| OnMouseExit() | is called as soon as the mouse pointer exits a game object's bounding area |

**2D collider**

* A 2D collider is a component that you can add to a GameObject to define its physical shape for 2D physics interactions.
* Colliders are essential for detecting collisions between objects and for applying physics-based behaviours like rigidbody interactions, collision responses, and triggers.

Here are the most common 2D colliders:

| **2D colliders** | **Description** |
| --- | --- |
| Box Collider 2D | The Box Collider 2D creates a rectangular collider around the GameObject.It's ideal for objects with simple rectangular shapes, like platforms or walls. |
| Circle Collider 2D | The Circle Collider 2D creates a circular collider around the GameObject.  It's suitable for objects with round or circular shapes, like coins or balls. |
| Edge Collider 2D | The Edge Collider 2D creates a collider along the edges of the GameObject.  It's used for objects with long and thin shapes, like platforms with irregular shapes. |
| Polygon Collider 2D | The Polygon Collider 2D allows you to create a custom collider shape by defining a set of points. |
| Capsule Collider 2D | Use Capsule Collider2D for circular or lozenge-shaped collision areas. |
| Custom Collider 2D | Use Custom Collider 2D for collision areas with customizable geometry. |

**3D collider**

* A 3D collider is a component that you can add to a GameObject to define its physical shape for 3D physics interactions.
* Colliders are crucial for detecting collisions between objects and for applying physics-based behaviours like rigidbody interactions, collision responses, and triggers in 3D scenes.

Here are the most common 3D colliders:

| **3D colliders** | **Description** |
| --- | --- |
| Box Collider | The Box Collider creates a rectangular prism-shaped collider around the GameObject.It's ideal for objects with simple rectangular shapes, like walls, buildings, or crates. |
| Sphere Collider | The Sphere Collider creates a spherical collider around the GameObject.It's suitable for objects with round or circular shapes, like balls or globes. |
| Capsule Collider | The Capsule Collider creates a capsule-shaped collider, which is a cylinder with hemispherical ends.It's useful for character controllers, characters, or objects with a cylindrical shape like barrels or poles. |
| Mesh Collider | The Mesh Collider uses the object's actual 3D mesh as the collider.It's versatile and can be used for complex or irregular shapes that don't fit into the other collider types. |

**Joints**

* A Joint component connects a Rigidbody to another Rigidbody or a fixed point in space.
* physics joints are components that allow you to create physical connections between GameObjects, enabling them to interact with each other in a realistic manner based on the laws of physics.
* Unity provides various types of physics joints that can be used to simulate different types of physical connections, such as hinges, springs, and fixed joints.

Here are some of the common types of physics joints available in Unity:

| **Joints** | **Description** |
| --- | --- |
| Hinge Joint | The HingeJoint simulates a rotating connection between two Rigidbody objects. It restricts movement to a single axis, allowing objects to swing around a fixed point as if connected by a hinge, like a door on its hinges. |
| Fixed Joint | The FixedJoint completely fixes the relative position and rotation of two Rigidbody objects, making them act as if they are rigidly attached together, like welding two objects in place. |
| Spring Joint | The SpringJoint creates a spring-like connection between two Rigidbody objects, allowing them to move closer or farther apart while applying a spring force to return them to their original distance. |
| Configurable Joint | The ConfigurableJoint is a more advanced joint that allows you to configure various properties to create a wide range of joint behaviours. You can specify the motion axes, angular limits, and more. |
| Character Joint | The CharacterJoint is specifically designed for use with humanoid characters and helps simulate realistic articulations like shoulders and hips. |

# **Rigidbody**

* In Unity, a "Rigidbody" is a component that can be added to GameObjects to give them physics-based behaviour.
* Rigidbody allows the GameObject to interact with the physics engine, making it move, rotate, and respond to forces like gravity, collisions, and other physical interactions.
* When a GameObject has a Rigidbody component attached, it becomes subject to the laws of physics, and you can apply forces to it or manipulate its position and rotation through physics-based methods.
* After adding the Rigidbody, you can adjust its properties in the Inspector, such as:

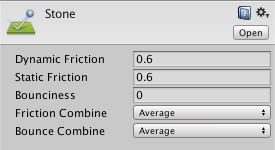
| **Properties** | **Description** |
| --- | --- |
| Mass | The mass of the object, which affects how it responds to forces and collisions. |
| Drag | Air resistance affecting the object's movement. |
| Angular Drag | Air resistance affecting the object's rotation. |
| Use Gravity | Determines whether the object is affected by the global gravity setting in Unity. |
| Is Kinematic | If enabled, the object will not be affected by physics forces and collisions and must be moved manually through scripts. |

# **Materials, Shaders & Textures**

* Rendering in Unity is done with **Materials**, **Shaders** and **Textures**.
* There is a close relationship between Materials, Shaders and Textures in Unity.
* **Materials** are definitions of how a surface should be rendered, including references to textures used, tiling information, colour tints and more. The available options for a material depend on which shader the material is using.
* **Shaders** are small scripts that contain the mathematical calculations and algorithms for calculating the colour of each pixel rendered, based on the lighting input and the Material configuration.
* **Textures** are bitmap images. A Material may contain references to textures, so that the Material’s shader can use the textures while calculating the surface colour of an object. In addition to basic colour (albedo) of an object's surface, textures can represent many other aspects of a material’s surface such as its reflectivity or roughness.

# **Physic Material**

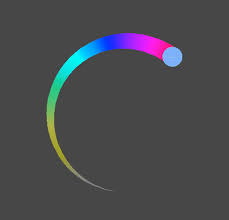
* The **Physic Material** is used to adjust friction and bouncing effects of colliding objects.
* To create a Physic Material select **Assets > Create > Physic Material** from the menu bar. Then drag the Physic Material from the Project View onto a **Collider** in the scene.



* Friction is the quantity which prevents surfaces from sliding off each other.
* **Bounciness:=**How bouncy is the surface? A value of 0 will not bounce. A value of 1 will bounce without any loss of energy.

# **Trail Renderer**

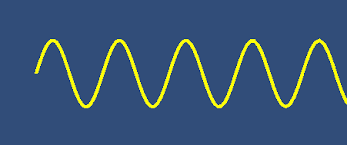
* The trail renderer is used to make trails behind objects in the Scene as they move about.
* The **Trail Renderer** component renders a trail of polygons behind a moving **GameObject**. over time. This can be used to give an emphasised feeling of motion to a moving object, or to highlight the path or position of moving objects.



# 

# **Line Renderer**

* The **Line Renderer** component takes an array of two or more points in 3D space, and draws a straight line between each one.
* The line renderer is used to draw free-floating lines in 3D space.
* A single Line Renderer component can therefore be used to draw anything from a simple straight line to a complex spiral.
* Line renderer can also help in targeting systems where you want to draw a raycast of where a bullet might shoot to in a first person game.



# **StartCoroutine**

* The execution of a coroutine can be paused at any point using the yield statement. When a yield statement is used, the coroutine pauses execution and automatically resumes at the next frame. See the Coroutines documentation for more details.
* Coroutines are excellent when modelling behaviour over several frames. The StartCoroutine method returns upon the first yield return, however you can yield the result, which waits until the coroutine has finished execution. There is no guarantee coroutines end in the same order they started, even if they finish in the same frame.
* Yielding of any type, including null, results in the execution coming back on a later frame, unless the coroutine is stopped or has completed.

IEnumerator **MyCoroutine**()

{

**while**(true)

{

**yield** **return** null;

}

}

# **Invoke**

* If time is set to 0 and Invoke is called before the first frame update, the method is invoked at the next Update cycle before MonoBehaviour.Update. In this case, it's better to call the function directly.
* Setting time to negative values is identical to setting it to 0.
* The order of execution of the method depends on the timing of the invocation.

public void **Invoke**(string **methodName**, float **time**);

## **Invoke**

invoke is a public method of MonoBehaviour class (so you can access it from your game scripts). It takes method name and time as parameters and it can be used in that way:

**Invoke("LaunchProjectile", 2);**

## **InvokeRepeating**

Invoke*Repeating* is very similar to Invoke method but it takes repeat interval as a third parameter, so there can be a difference between the first execution time and any other.

**InvokeRepeating("LaunchProjectile", 2, 0.3f);**