Unity Script Basics

1. **MonoBehaviour** is a fundamental class in Unity's scripting API. It acts as the base class from every Unity script derived. Scripts that are attached to GameObjects in Unity are typically subclasses of MonoBehaviour. This class provides access to many useful functions and events that are essential for game development, such as Start, Update, and many others.

   Code Example:

   ```
   using UnityEngine;

   public class ExampleScript: MonoBehaviour
   {
       // Start is called before the first frame update
       void Start()
       {
           Debug.Log("Hello, World!");
       }

       // Update is called once per frame
       void Update()
       {
           // Code to be executed every frame
       }
   }
   ```

2. The **Start** Method is called once in the lifetime of the script, just before the first frame where the script is active. It's typically used for initialization tasks.

   Code Example:

   ```
   void Start()
   {
       Debug.Log("The game has started!");
       // Initialization code here. For example, setting initial values.
   }
   ```

3. **Rigidbody** in Unity is a component that allows a GameObject to be affected by the engine's physics system. It adds properties like mass, drag, and allows the object to react to forces, collisions, and gravity. This component is essential for simulating realistic movement and interaction with other objects in the game world.

   Code Example:
   ```
   using UnityEngine;

   public class BallController: MonoBehaviour
   {
       Rigidbody rb;
   ```

```
    void Start()
    {
          // Get the Rigidbody component
            attached to this GameObject
        rb = GetComponent<Rigidbody>();
    }

    void Update()
    {
        // Check for a key press
        if (Input.GetKeyDown(KeyCode.Space))
        {
            // Apply an upward force to the Rigidbody
            rb.AddForce(Vector3.up * 5, ForceMode.Impulse);
        }
    }
}

/*In this script:
   a. Start Method: In the Start method, we get the Rigidbody component
      attached to the same GameObject as this script and store it in the
      rb variable.
   b. Update Method: In the Update method, we check if the spacebar key
      (KeyCode.Space) is pressed.
   c. Applying Force: If the spacebar is pressed, we apply an upward
      force to the Rigidbody using rb.AddForce(). The Vector3.up * 5
      part specifies the direction and magnitude of the force (upward in
      this case), and ForceMode.Impulse applies the force immediately,
      making it suitable for sudden impacts like jumps. */
```

4. A **GameObject** in Unity is the fundamental entity in a scene. It can represent characters, props, scenery, cameras, waypoints, and more. Essentially, a GameObject is a container for components such as meshes, colliders, scripts, and other Unity behaviors. Here are some simple examples to illustrate how GameObject is used in Unity scripting:

   Code Example 1, Creating a GameObject in Code:

```
// Create a new GameObject with a name
GameObject myGameObject = new GameObject("MyGameObject");

// You can now add components to this GameObject, like a Rigidbody or a
custom script
myGameObject.AddComponent<Rigidbody>();

//This code creates a new GameObject named "MyGameObject" and adds a
Rigidbody component to it, making it subject to physics.
```

   Code Example 2, Accessing a GameObject by Name:

```
// Find a GameObject in the scene by its name
GameObject foundGameObject = GameObject.Find("MyGameObject");
```

```
// You can now manipulate this GameObject, like changing its position
foundGameObject.transform.position = new Vector3(0, 0, 0);

//Here, GameObject.Find is used to find a GameObject named "MyGameObject"
in the scene. Once found, its position is set to the origin (0,0,0).
```

Code Example 2, Attaching a Script to a GameObject:

```
public class MyScript : MonoBehaviour
{
    void Start()
    {
        // Access the GameObject that this script is attached to
        GameObject thisGameObject = this.gameObject;

        // Do something with the GameObject, like changing its color
        Renderer renderer = thisGameObject.GetComponent<Renderer>();
        renderer.material.color = Color.blue;
    }
}

//In this script, this.gameObject refers to the GameObject to which
MyScript is attached. The script then changes the color of the GameObject
to blue.
```

5. **Vector3** in Unity is a structure that represents a three-dimensional vector or point in space, used for storing positions, rotations, and scales. It contains three float components: x, y, and z.

   Code Example:

```
using UnityEngine;

public class ExampleScript : MonoBehaviour
{
    void Start()
    {
        // Create a new Vector3 to represent a position.
        Vector3 startPosition = new Vector3(0, 0, 0);

        // Set the position of this GameObject to startPosition.
        transform.position = startPosition;

        // Move the GameObject by adding to its current position.
        Vector3 movement = new Vector3(1, 0, 0); // Move 1 unit along the
x-axis.
        transform.position += movement;
    }
}
/*In this Script:
    i.  A Vector3 named startPosition is created and set to (0, 0, 0),
        representing the origin in 3D space.
   ii.  The transform.position of the GameObject this script is attached
        to is set to startPosition.
  iii.  Another Vector3 named movement is created, representing a movement
        alonthe x-axis.
   iv.  The GameObject's position is then updated by adding movement to
        its current position, effectively moving it 1 unit along the
        x-axis. */
```

6. The **Transform** component in Unity is a fundamental part of every GameObject. It represents the object's position, rotation, and scale in the game world. Here's a simple explanation of how Transform is typically used in Unity scripts.

   Code Examples:
```
using UnityEngine;

public class ExampleScript : MonoBehaviour
{
    void Update()
    {
        // Move the GameObject 1 unit upwards every frame
        transform.position += new Vector3(0, 1, 0) * Time.deltaTime;

        // Rotate the GameObject by 30 degrees around the y-axis every second
        transform.Rotate(new Vector3(0, 30, 0) * Time.deltaTime, Space.World);
    }
}

/* In this
   a. transform.position += new Vector3(0, 1, 0) * Time.deltaTime;: This line
      moves the GameObject upwards at a constant speed. Time.deltaTime is used
      to make the movement frame-rate independent.
   b. transform.Rotate(new Vector3(0, 30, 0) * Time.deltaTime, Space.World);:
      This line rotates the GameObject around its y-axis. We use Space.World
      to rotate it in world space. */
```

7. A **Quaternion** in Unity is a data structure used to represent and manipulate rotations. It's often preferred over traditional Euler angles (which use degrees to represent rotations around the X, Y, and Z axes) because it avoids problems like gimbal lock and provides smoother rotational interpolation.

   Code Example:
```
using UnityEngine;

public class ExampleScript: MonoBehaviour
{
    void Update()
    {
      // Rotate the GameObject by 45 degrees around the Y axis every
      second
      Quaternion rotation = Quaternion.Euler(0, 45 * Time.deltaTime, 0);
```

```
            transform.rotation = transform.rotation * rotation;
        }
    }

    /* In this Script:
       a. Quaternion.Euler(0, 45 * Time.deltaTime, 0) creates a Quaternion
          representing a rotation. The rotation is 45 degrees around the
          Y-axis. Time.deltaTime is used to make the rotation frame-rate
          independent.
       b. transform.rotation = transform.rotation * rotation; applies this
          rotation to the current rotation of the GameObject. The
          multiplication of Quaternions results in a combined rotation.*/
```

8. **deltaTime** in Unity's context refers to the time in seconds it took to complete the last frame. It's a crucial concept in game development as it helps in creating frame rate independent movement and animations. This means that regardless of how fast or slow the game is running, movements will appear consistent.

   Code Example:

```
    void Update()
    {
        float speed = 5.0f; // Speed of the GameObject
        // Move the GameObject forward continuously at speed per second
        transform.Translate(Vector3.forward * speed * Time.deltaTime);
    }
    /* In this Script:
       a. Update() is a method that Unity calls once per frame.
       b. transform.Translate is a method that moves the GameObject in a
          specified direction.
       c. Vector3.forward represents the forward direction (along the Z-axis
          in Unity's 3D space).
       d. speed is a variable that determines how fast the GameObject moves.
       e. Time.deltaTime is multiplied with speed to ensure the movement
          speed is consistent across different frame rates. Without
          Time.deltaTime, the GameObject's speed would vary depending on the
          frame rate, moving faster on higher frame rates and slower on
          lower ones. */
```

9. **Linear interpolation**, often referred to as Lerp in Unity, is a method used to smoothly transition between two values over a certain period. In Unity, Lerp is commonly used for positions, rotations, scales, colors, and more, ensuring that these transitions appear smooth and natural.

   Code Example:

```
    using UnityEngine;

    public class ExampleScript : MonoBehaviour
    {
        public Transform startMarker;
        public Transform endMarker;
        public float speed = 1.0F;
        private float startTime;
        private float journeyLength;
        void Start()
        {
            startTime = Time.time;
           journeyLength = Vector3.Distance(startMarker.position,
           endMarker.position);
        }

        void Update()
        {
            float distCovered = (Time.time - startTime) * speed;
            float fractionOfJourney = distCovered / journeyLength;
           transform.position = Vector3.Lerp(startMarker.position,
           endMarker.position, fractionOfJourney);
        }
    }
    /* In this Script:
       a. startMarker and endMarker are the starting and ending points of
          the movement.
       b. speed determines how fast the object moves from the start to the
          end point.
       c. startTime records the time when the movement started.
       d. journeyLength calculates the distance between the start and end
          points.
       e. In the Update method, the script calculates how far along the
          journey the object should be based on the elapsed time.
       f. Vector3.Lerp is then used to smoothly interpolate the object's
          position between the start and end points based on the calculated
          fraction of the journey. */
```

10. **MovePosition and MoveRotation** are methods in Unity used to move and rotate Rigidbody objects. These methods are particularly useful when you need to move or rotate objects in a physics-based environment, as they allow for smooth movement and rotation while taking into account the physics simulation.

    - MoveRotation is is used to rotate a Rigidbody. Similar to MovePosition, it interacts with the physics engine, making it suitable for physics-based rotations.

      Code Example:

      ```
      public class ExampleScript : MonoBehaviour
      {
          public Rigidbody rb;
          public Quaternion newRotation;

          void Update()
          {
              // Rotate the Rigidbody to the new rotation
              rb.MoveRotation(newRotation);
          }
      }
      //In this example, rb.MoveRotation(newRotation) rotates the
      Rigidbody to newRotation. This rotation is often calculated using
      ```

```
Quaternion.Euler if you want to rotate in degrees, or it might be
a direct assignment from another rotation value.
```

- Move Position is used to move a Rigidbody to a new position. It's a physics-based method, so it moves the object in a way that interacts properly with the physics engine, unlike directly setting the position of the transform which can lead to unrealistic or buggy behavior in physics simulations.

  Code Example:
  ```
  public class ExampleScript : MonoBehaviour
  {
      public Rigidbody rb;
      public Vector3 newPosition;

      void Update()
      {
          // Move the Rigidbody to the new position
          rb.MovePosition(newPosition);
      }
  }
  //In this example, rb.MovePosition(newPosition) is used to move
  the Rigidbody attached to this object to newPosition. This would
  typically be updated in the Update method or within a coroutine
  for smooth movement.
  ```

11. **Mathf.Abs** is a method in Unity's Mathf class that returns the absolute value of a number. The absolute value of a number is its distance from zero on the number line, without considering the direction (positive or negative). This method is often used in scenarios where you need to work with the magnitude of a value irrespective of its sign.

    Code Example:
    ```
    using UnityEngine;

    public class ExampleScript: MonoBehaviour
    {
        void Start()
        {
            float negativeValue = -10f;
            float absoluteValue = Mathf.Abs(negativeValue);

            Debug.Log("The absolute value of " + negativeValue + " is " +
    absoluteValue);
        }
    }
    /* In this Script:
       a. We have a negative float value, -10f.
       b. Mathf.Abs(negativeValue) is used to get the absolute value of
          negativeValue.
       c. The result is 10f, which is the absolute value of -10f.
       d. This value is then printed to the Unity console using Debug.Log.*/
    ```

12. **IEnumerator and StartCoroutine** are key components of Unity's scripting used for creating coroutines. Coroutines in Unity allow you to pause the execution of a function and resume it in the next frames. This is particularly useful for implementing delays, animations, or any sequence of actions that occur over time.

    - **IEnumerator** is an interface that methods (usually referred to as coroutines) must implement to be used with StartCoroutine. A method that returns IEnumerator can yield execution back to Unity and then continue from where it left off in the next frame.
        - Within an IEnumerator method, yield return is used to pause the coroutine. After the specified condition or time, the coroutine resumes.
    - **StartCoroutine** is a method used to start a coroutine. It takes an IEnumerator method as an argument.

    Code Example:

```
using UnityEngine;
using System.Collections;

public class ExampleScript: MonoBehaviour
{
    void Start()
    {
        // Start the coroutine named "ExampleCoroutine".
        StartCoroutine(ExampleCoroutine());
    }

    IEnumerator ExampleCoroutine()
    {
        // Print the time of when the function is first called.
        Debug.Log("Started Coroutine at timestamp : " + Time.time);

       // Yield execution of this coroutine and return to the main loop
       until next frame.
        yield return null;

        // After one frame, print the time again.
        Debug.Log("After one frame, the timestamp is : " + Time.time);

        // Yield execution for 2 seconds.
        yield return new WaitForSeconds(2);

        // After 2 seconds, print the time again.
        Debug.Log("Finished Coroutine at timestamp : " + Time.time);
    }
}
/*In this Script:
   a. The coroutine ExampleCoroutine is started in the Start method
      using StartCoroutine.
   b. Inside ExampleCoroutine, yield return null causes the coroutine to
      pause until the next frame.
   c. yield return new WaitForSeconds(2) pauses the coroutine for 2
      seconds.
   d. After each yield, execution resumes from the point of the last
      yield. */
```

13. **GetComponent** is a method in Unity used to access components attached to a GameObject. Components are essentially parts of a GameObject, like a Rigidbody, Collider, or a custom script. This method is extremely useful in Unity scripting, allowing scripts to interact with and manipulate other components on the same GameObject or on other GameObjects in the scene.

    Code Example:

```
//Imagine you have a GameObject in your Unity scene, like a
player character. This GameObject has a Rigidbody component
attached to it, which allows it to interact with Unity's
physics system.

using UnityEngine;

public class PlayerController : MonoBehaviour
{
    private Rigidbody playerRigidbody;

    void Start()
    {
        // Get the Rigidbody component attached to this GameObject
        playerRigidbody = GetComponent<Rigidbody>();
       // Now you can use playerRigidbody to manipulate the physics of the
       GameObject
    }

    void Update()
    {
        // Example: Add a force to the Rigidbody in the upward direction
        if (Input.GetKeyDown(KeyCode.Space))
        {
            playerRigidbody.AddForce(Vector3.up * 10, ForceMode.Impulse);
        }
    }
}
/* In this Script:
   a. GetComponent<Rigidbody>(): This line gets the Rigidbody component
      attached to the same GameObject the script is attached to.
   b. Start Method: In the Start method (which is called when the game starts),
      the Rigidbody component is fetched and stored in the playerRigidbody
      variable.
   c. Update Method: In the Update method (which is called once per frame),
      we're checking for a key press (space bar in this case) and using the
      playerRigidbody to apply an upward force, making the player "jump". */
```

14. A **Quaternion** in Unity is a data structure used to represent and manipulate rotations. It's often preferred over traditional Euler angles (which use degrees to represent rotations around the X, Y, and Z axes) because it avoids problems like gimbal lock and provides smoother rotational interpolation.

    Code Example:

```
using UnityEngine;

public class ExampleScript: MonoBehaviour
{
    void Update()
    {
       // Rotate the GameObject by 45 degrees around the Y axis every
       second
       Quaternion rotation = Quaternion.Euler(0, 45 * Time.deltaTime, 0);
       transform.rotation = transform.rotation * rotation;
    }
}

/* In this Script:
   a. Quaternion.Euler(0, 45 * Time.deltaTime, 0) creates a Quaternion
      representing a rotation. The rotation is 45 degrees around the
      Y-axis. Time.deltaTime is used to make the rotation frame-rate
      independent.
   b. transform.rotation = transform.rotation * rotation; applies this
      rotation to the current rotation of the GameObject. The
      multiplication of Quaternions results in a combined rotation.*/
```