SRM Institute of Science and Technology

Department of Computer Applications

Delhi – Meerut Road, Sikri Kalan, Ghaziabad, Uttar Pradesh – 201204

Circular – 2020-21

MCA GAI 2nd semester

Augmented Reality and Virtual Reality for Game Development Lab Manual

Lab 1: Set up a VR development environment using Unity

Title

Set up a VR development environment using Unity: Install VR development tools, import VR SDKs, Set up a new VR project

Aim

To successfully set up a Unity development environment configured for Virtual Reality (VR) application development, including the installation of necessary tools, integration of VR SDKs, and creation of a new VR-ready project.

Procedure

1. **Install Unity Hub and Unity Editor:**
    - Download Unity Hub from the official Unity website.
    - Install Unity Hub.
    - Through Unity Hub, install a recommended stable version of the Unity Editor (e.g., a Long Term Support - LTS version). Ensure to include the "Android Build Support" and "iOS Build Support" modules if you plan to develop for mobile VR platforms (like Meta Quest or Google Cardboard).
2. **Create a New Unity Project:**
    - Open Unity Hub.
    - Click "New Project".
    - Select the "VR" template (if available and suitable for your Unity version) or a "3D Core" template.
    - Give your project a meaningful name (e.g., "MyVRProject") and choose a location.
    - Click "Create Project".
3. **Import VR SDKs (e.g., OpenXR Plugin):**
    - Once the project opens in Unity Editor, go to `Window > Package Manager`.
    - In the Package Manager, select "Unity Registry" from the dropdown.
    - Search for and install the "XR Plugin Management" package.
    - After installation, go to `Edit > Project Settings`.
    - Navigate to "XR Plugin Management" on the left sidebar.

    - Under the "XR Plugin Providers" tab, check the box for the VR runtime you intend to use (e.g., "OpenXR" for cross-platform VR, "Oculus" for Meta Quest, "SteamVR" for Valve Index/HTC Vive).
    - If using OpenXR, click on "OpenXR" in the left sidebar under "XR Plugin Management" and ensure the correct feature groups are enabled (e.g., "Meta Quest Support" for Meta Quest development).
4. **Configure Project Settings for VR:**
    - In `Edit > Project Settings > Player`, under "XR Settings" (or similar, depending on Unity version and XR Plugin), ensure "Virtual Reality Supported" is checked.
    - Confirm that the correct SDKs are listed in the "Virtual Reality SDKs" list.
    - For mobile VR, ensure the correct "Company Name" and "Product Name" are set under `Project Settings > Player > Other Settings`.

Source Code

N/A (This lab focuses on environment setup and configuration within the Unity Editor, not coding.)

Input

- Unity Hub and Unity Editor installers.
- Selection of VR SDKs within Unity's Package Manager and Project Settings.

Expected Output

- A fully installed Unity Editor.
- A new Unity project created with the "VR" template or configured for VR.
- The chosen VR SDKs (e.g., OpenXR, Oculus) successfully imported and enabled in Project Settings.
- The Unity Editor is ready to build and run VR applications on the target VR hardware.

Lab 2: Create a simple VR scene

Title

Create a simple VR scene

Aim

To create a basic virtual reality scene within Unity, demonstrating fundamental scene creation and VR camera setup.

Procedure

1. **Open the VR Project:**
    - Open the Unity project created in Lab 1.
2. **Create a New Scene:**
    - Go to `File > New Scene`.
    - Select "Basic (Built-in)" or "Empty" template.
    - Save the scene (`File > Save Scene As...`) with a descriptive name (e.g., "MyFirstVRScene").
3. **Configure the VR Camera:**
    - In the Hierarchy window, delete the default "Main Camera" if it exists.
    - If using OpenXR, right-click in the Hierarchy and select `XR > XR Origin (VR)`. This prefab provides the camera and tracking space.
    - If using another SDK (e.g., Oculus Integration), import the relevant prefabs (e.g., `OVRCameraRig`) from the SDK's assets and drag them into the scene.
    - Position the `XR Origin` (or equivalent VR camera rig) at `(0, 0, 0)` for a starting point.
4. **Add Basic 3D Objects:**
    - Right-click in the Hierarchy window, go to `3D Object`, and add a "Cube", "Sphere", or "Plane".
    - Position these objects around the `XR Origin` to be visible from the VR camera's perspective. For example, place a plane at `(0, -1, 0)` to act as a floor, and a cube at `(0, 0, 3)` to be in front of the player.
    - Add different materials or colors to the objects for better visual distinction.
5. **Build and Run (Optional, but Recommended):**
    - Ensure your VR headset is connected and configured.
    - Go to `File > Build Settings...`.
    - Add your current scene to "Scenes In Build".
    - Select your target platform (e.g., PC, Mac & Linux Standalone for desktop VR; Android for mobile VR).
    - Click "Build And Run".

Source Code

N/A (This lab primarily involves scene setup and object placement within the Unity Editor, not extensive coding.)

Input

- N/A (Scene creation is done through the Unity Editor GUI).

Expected Output

- A new Unity scene containing a configured VR camera rig.
- Several basic 3D objects (e.g., cubes, spheres, planes) placed within the scene.
- When run on a VR headset, the user should be able to see the created objects from a first-person perspective, responding to head movements.

Lab 3: Experiment with different audio and visual effects for immersive experience

Title

Experiment with different audio and visual effects for immersive experience

Aim

To enhance the immersion of a VR scene by incorporating various audio and visual effects, understanding their impact on user presence and realism.

Procedure

1. **Open an Existing VR Scene:**
    - Open the simple VR scene created in Lab 2.
2. **Implement Spatial Audio:**
    - **Add an Audio Source:** Create an empty GameObject (`GameObject > Create Empty`) and name it "SpatialAudioSource". Add an `Audio Source` component to it (`Add Component > Audio > Audio Source`).
    - **Import Audio Clip:** Import a sound file (e.g., a simple ambient sound or a distinct sound effect) into your Unity project (`Assets > Import New Asset...`). Drag this audio clip into the `AudioClip` field of the `Audio Source` component.
    - **Configure Spatialization:** In the `Audio Source` component, set `Spatial Blend` to `1 (3D)`. This enables 3D audio. Experiment with `Volume Rolloff` settings (e.g., `Logarithmic Rolloff`) and `Min Distance`/`Max Distance` to control how the sound fades with distance.
    - **Add an Audio Listener:** Ensure your VR camera rig (e.g., `XR Origin`'s camera) has an `Audio Listener` component. (Unity usually adds this automatically to the main camera).
3. **Apply Post-Processing Effects:**
    - **Install Post Processing Package:** Go to `Window > Package Manager`, select "Unity Registry", and install the "Post Processing" package.
    - **Create Post-Process Volume:** Create an empty GameObject (`GameObject > Create Empty`) and name it "PostProcessVolume". Add a `Post-process Volume` component to it.
    - **Create Post-Process Profile:** In the `Post-process Volume` component, click "New" to create a new `Post-process Profile`. Save it in your Assets folder.
    - **Add Effects:** Check "Is Global" on the `Post-process Volume` component if you want effects to apply everywhere. Click "Add Effect..." and experiment with effects like `Bloom` (for glow), `Vignette` (for darkened edges), `Color Grading` (for color adjustments), and `Ambient Occlusion` (for subtle shading). Adjust their properties.
    - **Add Post-Process Layer to Camera:** Select your VR camera (the actual camera component within your `XR Origin`). Add a `Post-process Layer` component to it. Set the `Layer` to `Default` (or a custom layer if you've set one for post-processing) and assign the `Post-process Volume` to the `Volume` field.
4. **Incorporate Particle Systems:**
    - Right-click in the Hierarchy, go to `Effects > Particle System`.

- Experiment with different particle system presets (e.g., `Fire`, `Smoke`, `Explosion`) or customize properties like `Start Lifetime`, `Start Speed`, `Shape`, `Color over Lifetime`, and `Size over Lifetime` to create various visual effects.
- Position the particle system in your scene.

Source Code (Illustrative for triggering effects)

```
// Example: A simple script to toggle a Post-Process Volume's enabled state
// Attach this to an empty GameObject in your scene.

using UnityEngine;
using UnityEngine.Rendering.PostProcessing; // Make sure to include this
namespace

public class EffectToggler : MonoBehaviour
{
    public PostProcessVolume targetVolume; // Drag your PostProcessVolume
GameObject here in the Inspector
    public KeyCode toggleKey = KeyCode.T; // Key to toggle the effect

    void Update()
    {
        if (Input.GetKeyDown(toggleKey))
        {
            if (targetVolume != null)
            {
                targetVolume.enabled = !targetVolume.enabled;
                Debug.Log("Post-Process Volume Toggled: " +
targetVolume.enabled);
            }
            else
            {
                Debug.LogError("Target Post-Process Volume not assigned!");
            }
        }
    }
}

// Example: A simple script to play a sound effect when an object is clicked
// Attach this to an object with a Collider.

using UnityEngine;

public class SoundOnInteraction : MonoBehaviour
{
    private AudioSource audioSource;
    public AudioClip interactionSound; // Drag your audio clip here in the
Inspector

    void Start()
    {
        audioSource = GetComponent<AudioSource>();
        if (audioSource == null)
        {
            // Add an AudioSource component if it doesn't exist
            audioSource = gameObject.AddComponent<AudioSource>();
        }
        audioSource.playOnAwake = false; // Don't play on start
        audioSource.clip = interactionSound;
        audioSource.spatialBlend = 1f; // Make it 3D sound
    }

    void OnMouseDown() // Or use an XR interaction system equivalent for VR
    {
        if (interactionSound != null)
```

```
        {
            audioSource.Play();
            Debug.Log("Playing interaction sound.");
        }
    }
}
```

Input

- Audio clips (e.g., .wav, .mp3).
- Adjustments to component properties in the Unity Inspector.
- Keyboard input (if using the example `EffectToggler` script).

Expected Output

- The VR scene should incorporate spatialized audio, where sounds appear to originate from specific points in 3D space and their volume changes with distance.
- Visual effects like bloom, vignette, or color grading should be visible, enhancing the scene's mood or atmosphere.
- Particle systems should animate, creating effects like smoke, fire, or sparks within the VR environment.
- The overall VR experience should feel more immersive and visually/audibly rich.

Lab 4: Set up AR Foundation & ARKit Package

Title

Set up AR Foundation & ARKit Package

Aim

To configure a Unity project for Augmented Reality (AR) development using Unity's AR Foundation, specifically targeting Apple's ARKit platform for iOS devices.

Procedure

1. **Create a New Unity Project (or use existing):**
    - Open Unity Hub.
    - Click "New Project".
    - Select the "3D Core" template.
    - Name your project (e.g., "MyARProject") and choose a location.
    - Click "Create Project".
2. **Install AR Foundation:**
    - Once the project opens, go to `Window > Package Manager`.
    - In the Package Manager, select "Unity Registry" from the dropdown.
    - Search for and install the "AR Foundation" package. This is the core framework.
3. **Install Platform-Specific AR Packages (ARKit for iOS):**
    - While still in the Package Manager, search for and install the "ARKit XR Plug-in" package. This provides ARKit-specific functionalities.
    - (Optional: If targeting Android, also install "ARCore XR Plug-in").
4. **Configure XR Plug-in Management:**
    - Go to `Edit > Project Settings`.
    - Navigate to "XR Plug-in Management" on the left sidebar.
    - Under the "Plug-in Providers" tab, ensure that "ARKit" is checked for the iOS platform tab (and "ARCore" for Android if applicable).
5. **Configure Player Settings for iOS:**
    - Go to `Edit > Project Settings > Player`.
    - Select the iOS tab (iPhone icon).
    - Under "Other Settings":
        - Set a unique "Bundle Identifier" (e.g., `com.YourCompanyName.MyARProject`).
        - Ensure "Camera Usage Description" is filled out (e.g., "This app uses the camera for augmented reality."). This is mandatory for iOS AR apps.
        - Set "Minimum iOS Version" to a version that supports ARKit (e.g., iOS 11.0 or higher).
        - Set "Architecture" to "ARM64".
6. **Add AR Session and AR Session Origin:**
    - In your scene, delete the default "Main Camera".
    - Right-click in the Hierarchy window and select `XR > AR Session`. This GameObject manages the AR lifecycle.
    - Right-click in the Hierarchy window and select `XR > AR Session Origin`. This GameObject transforms AR content from the device's coordinate space into Unity's world space. It also contains the AR Camera.

Source Code

N/A (This lab focuses on environment setup and configuration within the Unity Editor, not coding.)

Input

- Unity Editor installation.
- Selection of AR Foundation and ARKit packages in Unity's Package Manager.
- Configuration settings in Project Settings.

Expected Output

- A Unity project configured with AR Foundation and ARKit XR Plug-in.
- The project settings for iOS are correctly set up for AR deployment.
- The scene contains `AR Session` and `AR Session Origin` GameObjects, indicating readiness for AR content.
- The Unity Editor is prepared to build and run AR applications on compatible iOS devices.

Lab 5: Creating and Scripting a Placement Indicator in Unity

Title

Creating and Scripting a Placement Indicator in Unity

Aim

To develop a visual indicator in an AR scene that helps the user understand where a virtual object can be placed by detecting real-world surfaces (planes) and updating its position accordingly.

Procedure

1. **Open an AR Foundation Project:**
    - Open the Unity project configured with AR Foundation (from Lab 4).
2. **Create a New Scene:**
    - Go to `File > New Scene`. Save it (e.g., "ARPlacementScene").
    - Ensure `AR Session` and `AR Session Origin` are in the scene (as done in Lab 4).
3. **Add AR Plane Manager:**
    - Select the `AR Session Origin` GameObject in the Hierarchy.
    - Add an `AR Plane Manager` component (`Add Component > AR > AR Plane Manager`). This component detects and tracks planes in the real world.
    - (Optional: Add an `AR Plane` prefab to the `AR Plane Manager`'s `Plane Prefab` slot to visualize detected planes. You can use Unity's default `ARPlanePrefab` or create a simple transparent material for it).
4. **Create the Placement Indicator Visual:**
    - Create a simple 3D object to serve as the indicator. Right-click in the Hierarchy, `3D Object > Cylinder` or `Quad`. Scale it down to be small and flat (e.g., `Scale: 0.1, 0.01, 0.1` for a cylinder).
    - Name it "PlacementIndicator".
    - Create a distinct material for it (e.g., a semi-transparent green or blue) to make it easily visible.
    - Initially, disable this GameObject in the Inspector (uncheck the box next to its name) as it will only appear when a plane is detected.
5. **Create the Placement Indicator Script:**
    - Create a new C# script (`Assets > Create > C# Script`) and name it "PlacementIndicator".
    - Attach this script to an empty GameObject in your scene (e.g., create an empty GameObject named "ARManager" and attach the script to it).
6. **Implement Script Logic:**
    - The script will use `ARRaycastManager` to perform raycasts from the center of the screen into the real world.
    - If a plane is hit by the raycast, the indicator will be enabled and positioned at the hit point, aligned with the plane's normal.

Source Code (PlacementIndicator.cs)

```
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.XR.ARFoundation; // Required for AR classes
using UnityEngine.XR.ARSubsystems; // Required for TrackableType

public class PlacementIndicator : MonoBehaviour
```

```
{
    [SerializeField]
    private GameObject placementIndicatorVisual; // Drag your
PlacementIndicator visual here in the Inspector

    private ARRaycastManager arRaycastManager;
    private Pose placementPose; // Represents the position and rotation of
the indicator
    private bool placementPoseIsValid = false; // True if a valid plane is
detected

    void Start()
    {
        arRaycastManager = FindObjectOfType<ARRaycastManager>();
        if (arRaycastManager == null)
        {
            Debug.LogError("ARRaycastManager not found in scene! Please add
AR Session Origin and AR Raycast Manager.");
            enabled = false; // Disable this script if manager is missing
        }

        if (placementIndicatorVisual == null)
        {
            Debug.LogError("Placement Indicator Visual not assigned! Please
drag your visual GameObject to the inspector.");
            enabled = false;
        }
        else
        {
            placementIndicatorVisual.SetActive(false); // Hide indicator
initially
        }
    }

    void Update()
    {
        UpdatePlacementPose();
        UpdatePlacementIndicatorVisual();
    }

    private void UpdatePlacementPose()
    {
        // Raycast from the center of the screen
        var screenCenter = Camera.current.ViewportToScreenPoint(new
Vector3(0.5f, 0.5f));
        var hits = new List<ARRaycastHit>();

        // Perform raycast against detected planes
(TrackableType.PlaneWithinPolygon)
        if (arRaycastManager.Raycast(screenCenter, hits,
TrackableType.PlaneWithinPolygon))
        {
            // Get the first hit (closest plane)
            placementPose = hits[0].pose;
            placementPoseIsValid = true;

            // Optional: Adjust rotation to align with camera if desired
(e.g., for UI elements)
            // var cameraForward = Camera.current.transform.forward;
            // var cameraBearing = new Vector3(cameraForward.x, 0,
cameraForward.z).normalized;
            // placementPose.rotation =
Quaternion.LookRotation(cameraBearing);
        }
        else
        {
```

```
            placementPoseIsValid = false;
        }
    }

    private void UpdatePlacementIndicatorVisual()
    {
        if (placementPoseIsValid)
        {
            placementIndicatorVisual.SetActive(true);

placementIndicatorVisual.transform.SetPositionAndRotation(placementPose.posit
ion, placementPose.rotation);
        }
        else
        {
            placementIndicatorVisual.SetActive(false);
        }
    }

    // Public method to get the current valid placement pose
    public Pose GetPlacementPose()
    {
        return placementPose;
    }

    // Public method to check if a valid placement pose exists
    public bool IsPlacementPoseValid()
    {
        return placementPoseIsValid;
    }
}
```

Input

- Movement of the mobile device to scan the real-world environment.
- Detection of horizontal or vertical planes by AR Foundation.

Expected Output

- A small, distinct visual indicator (e.g., a green circle or square) appears on the screen.
- The indicator moves and snaps to detected real-world surfaces (floors, tables, walls) as the user points their device.
- The indicator disappears when no valid plane is detected.
- This visual feedback confirms that the AR system is tracking and ready for object placement.

Lab 6: Create an AR game by importing 3D objects

Title

Create an AR game by importing 3D objects

Aim

To develop a simple Augmented Reality (AR) game in Unity where users can import and place 3D objects into their real-world environment using the placement indicator developed previously.

Procedure

1. **Open AR Project with Placement Indicator:**
   - Open the Unity project from Lab 5, ensuring the scene contains `AR Session`, `AR Session Origin`, `AR Plane Manager`, and the `PlacementIndicator` script and visual.
2. **Import 3D Objects:**
   - Import one or more 3D model files (e.g., .fbx, .obj, .gltf) into your Unity project (`Assets > Import New Asset...`). You can find free 3D models online (e.g., Sketchfab, Poly Pizza).
   - Drag the imported model(s) into your scene to ensure they are correctly imported and scaled. Then, drag them from the Hierarchy back into your Project window to create prefabs. Delete the instances from the scene.
3. **Create a Spawner Script:**
   - Create a new C# script (`Assets > Create > C# Script`) and name it "ARObjectSpawner".
   - Attach this script to the same GameObject as your `PlacementIndicator` script (e.g., "ARManager").
4. **Implement Spawner Logic:**
   - The `ARObjectSpawner` script will reference the `PlacementIndicator` script.
   - When the user taps the screen (or uses another input method), the script will check if the `PlacementIndicator` has a valid placement pose.
   - If valid, it will instantiate one of the imported 3D object prefabs at the `placementPose` position and rotation.
5. **Add Input Handling:**
   - For mobile AR, input is typically handled via touch. Use `Input.touchCount` and `Input.GetTouch(0).phase == TouchPhase.Began` to detect a tap.
6. **Build and Run on Device:**
   - Go to `File > Build Settings...`.
   - Ensure the iOS platform is selected (or Android if you configured ARCore).
   - Add your scene to "Scenes In Build".
   - Click "Build" or "Build And Run" and deploy to a compatible AR device.

Source Code (ARObjectSpawner.cs)

```
using UnityEngine;
using UnityEngine.XR.ARFoundation; // Required for AR classes
using UnityEngine.XR.ARSubsystems; // Required for TrackableType

public class ARObjectSpawner : MonoBehaviour
{
    [SerializeField]
```

```
    private GameObject objectToPlacePrefab; // Drag your 3D object prefab
here in the Inspector

    private PlacementIndicator placementIndicator; // Reference to the
PlacementIndicator script

    void Start()
    {
        placementIndicator = FindObjectOfType<PlacementIndicator>();
        if (placementIndicator == null)
        {
            Debug.LogError("PlacementIndicator script not found! Please
ensure it's in the scene.");
            enabled = false;
        }

        if (objectToPlacePrefab == null)
        {
            Debug.LogError("Object To Place Prefab not assigned! Please drag
your 3D model prefab to the inspector.");
            enabled = false;
        }
    }

    void Update()
    {
        // Check for touch input (for mobile AR)
        if (Input.touchCount > 0 && Input.GetTouch(0).phase ==
TouchPhase.Began)
        {
            TryPlaceObject();
        }
        // Optional: For testing in editor with mouse click
        // #if UNITY_EDITOR
        //         if (Input.GetMouseButtonDown(0))
        //         {
        //             TryPlaceObject();
        //         }
        // #endif
    }

    private void TryPlaceObject()
    {
        if (placementIndicator.IsPlacementPoseValid())
        {
            // Get the valid pose from the placement indicator
            Pose pose = placementIndicator.GetPlacementPose();

            // Instantiate the object at the detected pose
            Instantiate(objectToPlacePrefab, pose.position, pose.rotation);
            Debug.Log("Object placed at: " + pose.position);
        }
        else
        {
            Debug.Log("Cannot place object: No valid plane detected.");
        }
    }
}
```

Input

- Imported 3D model files.
- Touch input on the mobile device screen.
- Real-world environment for plane detection.

Expected Output

- The placement indicator appears on detected surfaces.
- When the user taps the screen, an instance of the imported 3D object appears at the location indicated by the placement indicator.
- The 3D object is correctly scaled and oriented within the real-world environment through the AR view.
- The user can place multiple instances of the object by tapping repeatedly.

Lab 7: Create a simple scene with a controllable character (e.g., a cube) that can move forward, backward, left, and right using keyboard input.

Title

Create a simple scene with a controllable character (e.g., a cube) that can move forward, backward, left, and right using keyboard input.

Aim

To implement a basic character controller in Unity that allows a simple 3D object (e.g., a cube) to be moved using standard keyboard inputs (W, A, S, D).

Procedure

1. **Create a New Unity Project:**
    - Open Unity Hub and create a new "3D Core" project (or use an existing one).
2. **Create the Playable Character:**
    - In the Hierarchy window, right-click and select `3D Object > Cube`. Name it "Player".
    - Position the cube at `(0, 0.5, 0)` to ensure it's above the ground.
    - Add a `Rigidbody` component to the "Player" Cube (`Add Component > Physics > Rigidbody`). This allows physics-based movement and collision. Uncheck "Use Gravity" if you want to control vertical movement manually or disable it for simple top-down movement. For standard character movement, leave "Use Gravity" checked.
    - (Optional: Add a `Capsule Collider` and adjust its size if you want a more appropriate character collider for a humanoid-like character, instead of the default cube collider).
3. **Create a Ground Plane:**
    - Right-click in the Hierarchy, select `3D Object > Plane`. Name it "Ground".
    - Position it at `(0, 0, 0)` and scale it up (e.g., `Scale: 10, 1, 10`) to provide a large walkable surface.
4. **Create the Player Movement Script:**
    - Create a new C# script (`Assets > Create > C# Script`) and name it "PlayerMovement".
    - Attach this script to the "Player" Cube GameObject.
5. **Implement Movement Logic:**
    - In the `PlayerMovement` script, use `Input.GetAxis("Horizontal")` and `Input.GetAxis("Vertical")` to get input from the A/D and W/S keys (or arrow keys).
    - Apply movement using `transform.Translate()` for simple non-physics movement or `Rigidbody.velocity` / `Rigidbody.MovePosition()` for physics-based movement. For a basic character, `transform.Translate()` is often sufficient.

Source Code (PlayerMovement.cs)

```
using UnityEngine;

public class PlayerMovement : MonoBehaviour
{
    public float moveSpeed = 5.0f; // Speed at which the character moves
```

```
    void Update()
    {
        // Get input for horizontal and vertical axes
        // "Horizontal" maps to A/D keys and Left/Right arrow keys
        // "Vertical" maps to W/S keys and Up/Down arrow keys
        float horizontalInput = Input.GetAxis("Horizontal");
        float verticalInput = Input.GetAxis("Vertical");

        // Calculate movement direction based on input
        // Vector3.right is (1,0,0), Vector3.forward is (0,0,1)
        Vector3 movement = transform.right * horizontalInput +
transform.forward * verticalInput;

        // Normalize movement vector to prevent faster diagonal movement
        // and multiply by speed and Time.deltaTime for frame-rate
independent movement
        transform.Translate(movement.normalized * moveSpeed * Time.deltaTime,
Space.World);

        // Optional: If you want the character to face the direction of
movement
        // if (movement.magnitude > 0.1f) // Only rotate if there's
significant movement
        // {
        //     Quaternion targetRotation =
Quaternion.LookRotation(movement.normalized);
        //     transform.rotation = Quaternion.Slerp(transform.rotation,
targetRotation, Time.deltaTime * 10f);
        // }
    }
}
```

Input

- Keyboard keys: W (forward), S (backward), A (left), D (right).

Expected Output

- A cube (or chosen character object) visible in the scene.
- When the W key is pressed, the cube moves forward.
- When the S key is pressed, the cube moves backward.
- When the A key is pressed, the cube moves left.
- When the D key is pressed, the cube moves right.
- Movement should be smooth and consistent regardless of frame rate.

Lab 8: Implement a script that allows the player to interact with objects in the scene (e.g., picking up and dropping objects, triggering events).

Title

Implement a script that allows the player to interact with objects in the scene (e.g., picking up and dropping objects, triggering events).

Aim

To create a system that enables a player character to interact with specific objects in the game world, demonstrating concepts like object pickup/drop and event triggering.

Procedure

1. **Set up the Scene:**
    - Use the scene from Lab 7 with the controllable character.
    - Create several new 3D objects (e.g., "Sphere", "Capsule") that will be interactable. Position them near the player.
    - Add a `Box Collider` or `Sphere Collider` to each interactable object. Ensure "Is Trigger" is **unchecked** if you want physical interaction (picking up) or **checked** if you want to trigger an event on overlap.
    - Add a `Rigidbody` component to objects that can be picked up.
2. **Create an "Interactable" Tag:**
    - Select one of your interactable objects. In the Inspector, click the "Tag" dropdown and select "Add Tag...".
    - Click the + button and type "Interactable". Save.
    - Now, select your interactable objects again and set their Tag to "Interactable".
3. **Create the Interaction Script:**
    - Create a new C# script (`Assets > Create > C# Script`) and name it "PlayerInteraction".
    - Attach this script to your "Player" character GameObject.
4. **Implement Interaction Logic:**
    - The `PlayerInteraction` script will use raycasting from the player's camera (or a point in front of the player) to detect "Interactable" objects.
    - When the interaction key (e.g., 'E') is pressed and an interactable object is detected:
        - **For pickup/drop:** Parent the object to the player's hand/holding point, disable its Rigidbody physics, and then re-enable physics and unparent on drop.
        - **For event triggering:** Call a public method on the interactable object's script.
5. **Create a Simple Interactable Object Script (Example: Door Opener):**
    - Create another C# script named "DoorOpener" and attach it to a Cube GameObject. This cube will represent a door.
    - Add a `Box Collider` to the "Door" cube. Make sure "Is Trigger" is **checked**.
    - Implement a public method that can be called by the `PlayerInteraction` script.

Source Code (PlayerInteraction.cs)

```
using UnityEngine;

public class PlayerInteraction : MonoBehaviour
```

```
{
    public float interactionDistance = 3f; // How far the player can interact
    public KeyCode interactKey = KeyCode.E; // Key to trigger interaction
    public Transform holdPoint; // An empty GameObject child of the player,
where picked-up objects will be held

    private GameObject heldObject = null; // Reference to the object
currently held

    void Update()
    {
        if (Input.GetKeyDown(interactKey))
        {
            if (heldObject != null)
            {
                DropObject();
            }
            else
            {
                TryInteract();
            }
        }
    }

    void TryInteract()
    {
        RaycastHit hit;
        // Cast a ray from the camera (or player's forward direction)
        // Adjust origin if needed, e.g., Camera.main.transform.position
        if (Physics.Raycast(transform.position, transform.forward, out hit,
interactionDistance))
        {
            Debug.DrawRay(transform.position, transform.forward *
interactionDistance, Color.red, 1f); // For debugging
            if (hit.collider.CompareTag("Interactable"))
            {
                Debug.Log("Hit interactable object: " + hit.collider.name);
                // Check if the object has a Rigidbody (can be picked up)
                Rigidbody hitRb = hit.collider.GetComponent<Rigidbody>();
                if (hitRb != null)
                {
                    PickupObject(hitRb);
                }
                else
                {
                    // If no Rigidbody, assume it's an event trigger
                    // Try to find a script with an interact method
                    IInteractable interactable =
hit.collider.GetComponent<IInteractable>();
                    if (interactable != null)
                    {
                        interactable.Interact();
                    }
                    else
                    {
                        Debug.LogWarning("Interactable object " +
hit.collider.name + " has no Rigidbody and no IInteractable script.");
                    }
                }
            }
        }
    }

    void PickupObject(Rigidbody rb)
    {
        if (holdPoint == null)
```

```
        {
            Debug.LogError("Hold Point not assigned for PlayerInteraction
script!");
            return;
        }

        heldObject = rb.gameObject;
        heldObject.transform.SetParent(holdPoint); // Parent to hold point
        heldObject.transform.localPosition = Vector3.zero; // Reset local
position
        heldObject.transform.localRotation = Quaternion.identity; // Reset
local rotation

        rb.isKinematic = true; // Disable physics
        rb.useGravity = false; // Disable gravity
    }

    void DropObject()
    {
        if (heldObject != null)
        {
            Rigidbody rb = heldObject.GetComponent<Rigidbody>();
            if (rb != null)
            {
                rb.isKinematic = false; // Re-enable physics
                rb.useGravity = true; // Re-enable gravity
                // Add a small force to make it drop naturally
                rb.AddForce(transform.forward * 2f + Vector3.up * 1f,
ForceMode.Impulse);
            }
            heldObject.transform.SetParent(null); // Unparent
            heldObject = null;
        }
    }
}

// Interface for objects that can be interacted with to trigger an event
public interface IInteractable
{
    void Interact();
}

// Source Code (DoorOpener.cs) - Example of an event-triggering interactable
using UnityEngine;

public class DoorOpener : MonoBehaviour, IInteractable
{
    public bool isOpen = false;
    public Vector3 openPositionOffset = new Vector3(0, 3, 0); // How much to
move when opened
    private Vector3 initialPosition;

    void Start()
    {
        initialPosition = transform.position;
    }

    public void Interact()
    {
        if (!isOpen)
        {
            transform.position = initialPosition + openPositionOffset;
            Debug.Log("Door Opened!");
        }
        else
        {
```

```
            transform.position = initialPosition;
            Debug.Log("Door Closed!");
        }
        isOpen = !isOpen;
    }
}
```

Input

- Keyboard key 'E' (or chosen `interactKey`).
- Player character positioned within `interactionDistance` of an "Interactable" object.

Expected Output

- When the player presses 'E' while looking at a pick-up-able object, the object should attach to the player (e.g., appear in front of them) and stop responding to physics.
- Pressing 'E' again should drop the object, re-enabling its physics.
- When the player presses 'E' while looking at an event-triggering object (like the "Door" cube), a predefined event should occur (e.g., the "Door" cube moves to its open position).
- Debug messages in the console confirming interaction.

Lab 9: Design a simple user interface (UI) with buttons, sliders, and text elements to display information or control aspects of the game (e.g., health bar, score display).

Title

Design a simple user interface (UI) with buttons, sliders, and text elements to display information or control aspects of the game (e.g., health bar, score display).

Aim

To create a basic graphical user interface (GUI) in Unity using its UI system, incorporating common elements like buttons, sliders, and text to display game information and allow player control.

Procedure

1. **Create a New Scene:**
    - Start with a new empty scene (`File > New Scene`).
2. **Create a UI Canvas:**
    - Right-click in the Hierarchy window, select `UI > Canvas`. This automatically creates an EventSystem as well.
    - Select the `Canvas` GameObject. In the Inspector, set its `Render Mode` to `Screen Space - Overlay` (for a UI that always appears on top of the game world) or `Screen Space - Camera` (if you want the UI to scale with camera distance). For `Screen Space - Camera`, drag your `Main Camera` to the `Render Camera` slot.
    - Set `UI Scale Mode` to `Scale With Screen Size` and set a `Reference Resolution` (e.g., 1920x1080) to ensure the UI scales correctly on different screen sizes.
3. **Add Text Elements (e.g., Score Display, Health Bar Text):**
    - Right-click on the `Canvas` in the Hierarchy, select `UI > Text - TextMeshPro`. (If prompted, import the TMP Essentials).
    - Rename it "ScoreText". Position it at the top-right of the screen using the Rect Transform's anchor presets. Set its `Text` property to "Score: 0".
    - Repeat for "HealthText", positioning it at the top-left, with text "Health: 100".
4. **Add a Slider (e.g., Health Bar):**
    - Right-click on the `Canvas`, select `UI > Slider`. Rename it "HealthSlider".
    - Position it near the "HealthText".
    - In the Slider component, set `Min Value` to 0, `Max Value` to 100.
    - (Optional: Customize the `Background` and `Fill Area` Rect Transforms to make it look like a health bar).
5. **Add Buttons (e.g., Restart Button, Action Button):**
    - Right-click on the `Canvas`, select `UI > Button - TextMeshPro`. Rename it "ActionButton".
    - Position it at the bottom-center. Change the button's text to "Perform Action".
    - (Optional: Add another button, "RestartButton", for restarting the game).
6. **Create a UI Manager Script:**
    - Create a new C# script (`Assets > Create > C# Script`) and name it "UIManager".
    - Create an empty GameObject in the Hierarchy named "GameManager" and attach the `UIManager` script to it.
7. **Implement UI Logic:**

- In the `UIManager` script, create public variables to reference your UI elements (Text, Slider).
- Create public methods that can be called by button clicks or to update text/slider values.
- Connect the UI elements to the script:
  - For buttons: Select the button, in the Inspector, find the `On Click()` event, click +, drag the "GameManager" GameObject into the slot, and select the appropriate public method from the `UIManager` script.
  - For sliders: You can read its value directly or use its `On Value Changed` event.

Source Code (UIManager.cs)

```
using UnityEngine;
using TMPro; // Required for TextMeshPro
using UnityEngine.UI; // Required for Slider and Button

public class UIManager : MonoBehaviour
{
    [Header("UI Elements")]
    public TextMeshProUGUI scoreText; // Drag your ScoreText (TMP) here
    public TextMeshProUGUI healthText; // Drag your HealthText (TMP) here
    public Slider healthSlider; // Drag your HealthSlider here

    private int currentScore = 0;
    private float currentHealth = 100f;

    void Start()
    {
        // Initialize UI values
        UpdateScore(0);
        UpdateHealth(100f);

        // Add listener to the action button (if not already set in
Inspector)
        Button actionButton =
GameObject.Find("ActionButton")?.GetComponent<Button>();
        if (actionButton != null)
        {
            actionButton.onClick.AddListener(OnActionButtonClicked);
        }
    }

    // Public method to update the score
    public void UpdateScore(int newScore)
    {
        currentScore = newScore;
        if (scoreText != null)
        {
            scoreText.text = "Score: " + currentScore;
        }
    }

    // Public method to update health (and slider)
    public void UpdateHealth(float newHealth)
    {
        currentHealth = Mathf.Clamp(newHealth, 0, 100); // Clamp health
between 0 and 100
        if (healthText != null)
        {
            healthText.text = "Health: " + Mathf.RoundToInt(currentHealth);
        }
        if (healthSlider != null)
        {
```

```
            healthSlider.value = currentHealth;
        }
    }

    // Method called when the Action Button is clicked
    public void OnActionButtonClicked()
    {
        Debug.Log("Action Button Clicked!");
        // Example: Increase score and decrease health
        UpdateScore(currentScore + 10);
        UpdateHealth(currentHealth - 5);
    }

    // Method called when the Slider value changes (if configured in
Inspector)
    public void OnHealthSliderChanged(float value)
    {
        Debug.Log("Health Slider changed to: " + value);
        // You might use this to set a game's difficulty or other settings
    }
}
```

Input

- Mouse clicks on buttons.
- Dragging the slider handle with the mouse.
- Changes in game state (simulated by `OnActionButtonClicked` in the example).

Expected Output

- A UI canvas appears on the screen, overlaying the game view.
- Text elements display initial score and health values.
- The health slider visually represents the health value.
- Clicking the "Perform Action" button updates the score text and decreases the health slider/text.
- Dragging the health slider updates its visual representation and logs the new value (if `OnHealthSliderChanged` is connected).
- The UI elements scale appropriately with different screen resolutions.

Lab 10: Develop a script that rotates an object (e.g., a sphere) around its axis when the player presses certain keys (e.g., Q and E).

Title

Develop a script that rotates an object (e.g., a sphere) around its axis when the player presses certain keys (e.g., Q and E).

Aim

To create a Unity script that allows a designated 3D object to rotate around its local axis in response to specific keyboard inputs (e.g., 'Q' for one direction, 'E' for the other).

Procedure

1. **Create a New Scene:**
    - Start with a new empty scene (`File > New Scene`).
2. **Create the Rotatable Object:**
    - In the Hierarchy window, right-click and select `3D Object > Sphere`. Name it "RotatableSphere".
    - Position it at `(0, 1, 0)` to make it clearly visible.
3. **Create the Rotation Script:**
    - Create a new C# script (`Assets > Create > C# Script`) and name it "ObjectRotator".
    - Attach this script to the "RotatableSphere" GameObject.
4. **Implement Rotation Logic:**
    - In the `ObjectRotator` script, use `Input.GetKey()` to detect when 'Q' or 'E' keys are pressed.
    - Use `transform.Rotate()` to apply rotation to the object. `transform.Rotate(Vector3.up * speed * Time.deltaTime)` would rotate around the local Y-axis.

Source Code (ObjectRotator.cs)

```
using UnityEngine;

public class ObjectRotator : MonoBehaviour
{
    public float rotationSpeed = 100f; // Speed of rotation in degrees per
second

    void Update()
    {
        // Rotate clockwise when 'E' is pressed
        if (Input.GetKey(KeyCode.E))
        {
            // Rotate around the object's local Y-axis (up)
            transform.Rotate(Vector3.up * rotationSpeed * Time.deltaTime);
            Debug.Log("Rotating clockwise...");
        }

        // Rotate counter-clockwise when 'Q' is pressed
        if (Input.GetKey(KeyCode.Q))
        {
            // Rotate around the object's local Y-axis (up) in the opposite
direction
            transform.Rotate(Vector3.up * -rotationSpeed * Time.deltaTime);
```

```
            Debug.Log("Rotating counter-clockwise...");
        }
    }
}
```

Input

- Keyboard keys: 'Q' and 'E'.

Expected Output

- A sphere (or chosen object) is visible in the scene.
- When the 'E' key is held down, the sphere continuously rotates around its vertical axis in one direction.
- When the 'Q' key is held down, the sphere continuously rotates around its vertical axis in the opposite direction.
- Rotation should be smooth and frame-rate independent.

Lab 11: Implement a timer script that counts down from a specified time (e.g., 60 seconds) and displays the remaining time on the screen.

Title

Implement a timer script that counts down from a specified time (e.g., 60 seconds) and displays the remaining time on the screen.

Aim

To create a countdown timer in Unity that decrements from a set duration and displays the remaining time on a UI text element, notifying the player when the time runs out.

Procedure

1. **Create a New Scene:**
    - Start with a new empty scene (`File > New Scene`).
2. **Set up UI for Timer Display:**
    - Create a UI Canvas (`Right-click in Hierarchy > UI > Canvas`).
    - Right-click on the `Canvas`, select `UI > Text - TextMeshPro`. (Import TMP Essentials if prompted).
    - Rename it "TimerText". Position it prominently on the screen (e.g., top-center).
    - Set its `Text` property to "Time: 00:00".
    - Adjust font size and color for readability.
3. **Create the Timer Script:**
    - Create a new C# script (`Assets > Create > C# Script`) and name it "CountdownTimer".
    - Create an empty GameObject in the Hierarchy named "GameManager" and attach the `CountdownTimer` script to it.
4. **Implement Timer Logic:**
    - In the `CountdownTimer` script:
        - Declare a float variable for the `currentTime`.
        - Declare a boolean `timerIsRunning`.
        - In `Start()`, initialize `currentTime` and set `timerIsRunning` to true.
        - In `Update()`, if `timerIsRunning` is true, decrement `currentTime` using `Time.deltaTime`.
        - Check if `currentTime` has reached zero or less. If so, stop the timer and trigger an "End Game" event (e.g., log a message).
        - Format and display the `currentTime` to the `TimerText` UI element.

Source Code (CountdownTimer.cs)

```
using UnityEngine;
using TMPro; // Required for TextMeshPro

public class CountdownTimer : MonoBehaviour
{
    public float startTime = 60f; // The time to count down from (in seconds)
    public TextMeshProUGUI timerText; // Drag your TimerText (TMP) here in
the Inspector

    private float currentTime;
    private bool timerIsRunning = false;

    void Start()
```

```
    {
        if (timerText == null)
        {
            Debug.LogError("TimerText (TextMeshProUGUI) not assigned! Please
assign it in the Inspector.");
            enabled = false; // Disable script if UI element is missing
            return;
        }

        currentTime = startTime;
        timerIsRunning = true;
        UpdateTimerDisplay(); // Initial display
    }

    void Update()
    {
        if (timerIsRunning)
        {
            if (currentTime > 0)
            {
                currentTime -= Time.deltaTime; // Decrement time
                UpdateTimerDisplay();
            }
            else
            {
                currentTime = 0; // Ensure it doesn't go negative
                timerIsRunning = false;
                UpdateTimerDisplay(); // Final display (00:00)
                OnTimerEnd(); // Call method when timer ends
            }
        }
    }

    void UpdateTimerDisplay()
    {
        // Calculate minutes and seconds
        int minutes = Mathf.FloorToInt(currentTime / 60);
        int seconds = Mathf.FloorToInt(currentTime % 60);

        // Format and display the time
        timerText.text = string.Format("Time: {0:00}:{1:00}", minutes,
seconds);
    }

    void OnTimerEnd()
    {
        Debug.Log("Timer has ended! Game Over!");
        // Here you would typically trigger game over logic,
        // e.g., show a game over screen, reset the level, etc.
    }

    // Optional: Method to reset the timer
    public void ResetTimer()
    {
        currentTime = startTime;
        timerIsRunning = true;
        UpdateTimerDisplay();
        Debug.Log("Timer reset.");
    }

    // Optional: Method to stop the timer
    public void StopTimer()
    {
        timerIsRunning = false;
        Debug.Log("Timer stopped.");
    }
```

```
}
```

Input

- Initial `startTime` set in the Inspector.
- N/A (timer runs automatically).

Expected Output

- A UI text element on the screen displays "Time: 00:00" initially.
- The timer starts counting down from the `startTime` (e.g., 60 seconds).
- The displayed time updates every frame, showing the remaining minutes and seconds.
- When the timer reaches 00:00, it stops, and a "Timer has ended! Game Over!" message is logged to the console.

Lab 12: Create a script that controls the playback of animations on a character or object (e.g., idle, walk, jump).

Title

Create a script that controls the playback of animations on a character or object (e.g., idle, walk, jump).

Aim

To implement a Unity script that manages and transitions between different animations (e.g., idle, walk, jump) on a character or object using an Animator Controller and keyboard input.

Procedure

1. **Import an Animated Model:**
    - Import a 3D model with multiple animations (e.g., a character with idle, walk, and jump animations) into your Unity project (`Assets > Import New Asset...`). Free animated models can be found on sites like Mixamo (after downloading, drag the .fbx into Unity).
    - Drag the imported model into your scene.
2. **Create an Animator Controller:**
    - Right-click in the Project window (`Assets` folder), select `Create > Animator Controller`. Name it "CharacterAnimatorController".
    - Select your animated model in the Hierarchy. In the Inspector, find the `Animator` component and drag your "CharacterAnimatorController" into the `Controller` slot.
3. **Set up Animator States and Transitions:**
    - Double-click the "CharacterAnimatorController" to open the Animator window.
    - Drag your animation clips (e.g., "Idle", "Walk", "Jump") from the imported model's asset into the Animator window to create new states.
    - **Create Parameters:** In the Animator window, go to the "Parameters" tab (`+` icon). Create:
        - A `Bool` parameter named "IsWalking".
        - A `Trigger` parameter named "Jump".
    - **Create Transitions:**
        - Right-click "Idle" state, select `Make Transition`, and drag the arrow to "Walk".
        - Select the transition arrow. In the Inspector, uncheck "Has Exit Time". Under "Conditions", add `IsWalking` and set it to `true`.
        - Right-click "Walk" state, select `Make Transition`, and drag the arrow to "Idle".
        - Select this transition. Uncheck "Has Exit Time". Under "Conditions", add `IsWalking` and set it to `false`.
        - Right-click "Any State", select `Make Transition`, and drag the arrow to "Jump".
        - Select this transition. Uncheck "Has Exit Time". Under "Conditions", add `Jump`.
        - Right-click "Jump" state, select `Make Transition`, and drag the arrow back to "Idle".
        - Select this transition. Check "Has Exit Time" (so it plays the full jump animation before returning).

4. **Create the Animation Control Script:**
    - Create a new C# script (`Assets > Create > C# Script`) and name it "CharacterAnimationController".
    - Attach this script to your animated character GameObject.
5. **Implement Animation Logic:**
    - In the `CharacterAnimationController` script, get a reference to the `Animator` component.
    - In `Update()`, check for keyboard input (e.g., W, A, S, D for walking, Space for jumping).
    - Use `animator.SetBool("IsWalking", true/false)` to control the walk animation.
    - Use `animator.SetTrigger("Jump")` to trigger the jump animation.

Source Code (CharacterAnimationController.cs)

```
using UnityEngine;

public class CharacterAnimationController : MonoBehaviour
{
    private Animator animator; // Reference to the Animator component
    public float movementThreshold = 0.1f; // Minimum movement to trigger
walk animation

    void Start()
    {
        animator = GetComponent<Animator>();
        if (animator == null)
        {
            Debug.LogError("Animator component not found on this GameObject!
Please add one.");
            enabled = false; // Disable script if Animator is missing
        }
    }

    void Update()
    {
        // Get horizontal and vertical input for movement
        float horizontalInput = Input.GetAxis("Horizontal");
        float verticalInput = Input.GetAxis("Vertical");

        // Calculate the magnitude of movement input
        Vector3 movementInput = new Vector3(horizontalInput, 0,
verticalInput);
        bool isMoving = movementInput.magnitude > movementThreshold;

        // Set the "IsWalking" boolean parameter in the Animator
        animator.SetBool("IsWalking", isMoving);

        // Check for Jump input
        if (Input.GetKeyDown(KeyCode.Space))
        {
            // Trigger the "Jump" animation
            animator.SetTrigger("Jump");
            Debug.Log("Jump animation triggered!");
        }
    }
}
```

Input

- Imported 3D model with animation clips.

- Keyboard keys: W, A, S, D (for movement leading to walk animation), Space (for jump).

Expected Output

- The character model is visible in the scene.
- When no movement keys are pressed, the character plays its "Idle" animation.
- When W, A, S, or D keys are pressed, the character transitions to and plays its "Walk" animation.
- When the Space key is pressed, the character briefly plays its "Jump" animation and then returns to "Idle" or "Walk" depending on other input.
- Transitions between animations are smooth based on Animator Controller settings.

Lab 13: Create your first unreal engine project

Title

Create your first Unreal Engine project

Aim

To successfully install Unreal Engine and create a new project, familiarizing with the initial setup process of the Unreal Editor.

Procedure

1. **Install Epic Games Launcher:**
    - Download the Epic Games Launcher from the official Epic Games website.
    - Install the launcher on your computer.
2. **Install Unreal Engine:**
    - Open the Epic Games Launcher.
    - Navigate to the "Unreal Engine" tab on the left sidebar.
    - Go to the "Library" section.
    - Click the + icon next to "ENGINE VERSIONS" to add a new engine slot.
    - Select the desired Unreal Engine version (e.g., the latest stable release or an LTS version).
    - Click "Install" and follow the prompts. This may take a significant amount of time and disk space.
3. **Launch Unreal Engine and Create a New Project:**
    - Once Unreal Engine is installed, click "Launch" next to the installed engine version in the Epic Games Launcher.
    - The "Unreal Project Browser" will open.
    - Select the "Games" category.
    - Choose a suitable template:
        - "Blank" for a completely empty project.
        - "Third Person" or "First Person" for a project with a pre-built character and basic controls (recommended for beginners).
        - "Virtual Reality" or "Augmented Reality" if you plan to jump directly into XR.
    - Select "Blueprint" as the project type (for visual scripting) or "C++" (for programming). Blueprint is recommended for beginners.
    - Choose "Maximum Quality" and "With Starter Content" (recommended to have basic assets).
    - Choose a location and name your project (e.g., "MyFirstUnrealProject").
    - Click "Create".

Source Code

N/A (This lab focuses on environment setup and project creation within the Unreal Editor, not coding.)

Input

- Epic Games Launcher installer.
- Unreal Engine version selection within the launcher.

- Project template, type, quality, and content selections in the Unreal Project Browser.

Expected Output

- Epic Games Launcher is installed and functional.
- A chosen version of Unreal Engine is successfully installed.
- The Unreal Editor launches, displaying a new project based on the selected template.
- The user is presented with the Unreal Editor interface, ready to begin development.

Lab 14: create a simple environment, author basic materials, explore the lighting system, and add basic Landscape and Foliage to bring the scene to life.

Title

Create a simple environment, author basic materials, explore the lighting system, and add basic Landscape and Foliage to bring the scene to life.

Aim

To construct a foundational game environment in Unreal Engine by creating a landscape, applying simple materials, configuring lighting, and adding foliage, thereby understanding core environment art workflows.

Procedure

1. **Open an Unreal Engine Project:**
    - Open the Unreal Engine project created in Lab 13 (e.g., "MyFirstUnrealProject").
2. **Create a New Level:**
    - Go to `File > New Level`.
    - Select "Basic" or "Empty Level".
    - Save the level (`File > Save Current As...`) with a descriptive name (e.g., "MyEnvironmentLevel").
3. **Create a Landscape:**
    - Go to `Modes > Landscape` (or press `Shift + 2`).
    - Click "Manage" and then "Create New". Adjust `Section Size` and `Number of Components` if needed.
    - Click "Create". This generates a large terrain.
    - Use the "Sculpt" tools (e.g., "Sculpt", "Flatten", "Ramp") to shape your terrain (hills, valleys).
4. **Author Basic Materials:**
    - In the Content Browser, right-click and select `Material`. Name it "M_Ground".
    - Double-click "M_Ground" to open the Material Editor.
    - Hold `3` and click to create a `Vector3` node (color). Connect it to `Base Color`. Choose a brown or green color.
    - Hold `1` and click to create a `Scalar Parameter` node (for roughness). Connect it to `Roughness`. Set its default value (e.g., 0.8 for rough ground).
    - Click "Apply" and "Save".
    - Drag "M_Ground" from the Content Browser onto your Landscape.
    - Create another material (e.g., "M_Rock") with a different color and roughness and apply it to some static meshes later.
5. **Explore the Lighting System:**
    - In the `Place Actors` panel (left sidebar), search for and drag these into your scene:
        - `Directional Light` (simulates the sun). Adjust its rotation for time of day.
        - `Sky Light` (captures distant lighting from the sky).
        - `Sky Atmosphere` (for realistic sky and atmospheric scattering).
        - `Exponential Height Fog` (for atmospheric depth).
    - Adjust properties of these lights in the Details panel (e.g., `Intensity`, `Color`).

    - (Optional: Add a `Post Process Volume` and set `Infinite Extent (Unbound)` to true. Experiment with `Exposure` and other settings).
6. **Add Basic Foliage:**
    - Go to `Modes > Foliage` (or press `Shift + 4`).
    - In the Content Browser, find some simple static mesh assets (e.g., from Starter Content: `Props > SM_Bush`, `SM_Tree`). Drag these into the `Foliage` panel.
    - Select the foliage type in the panel. Adjust properties like `Paint Density`, `Scale X/Y/Z`.
    - Use the "Paint" brush to paint instances of your foliage onto the landscape.

Source Code

N/A (This lab primarily involves using the Unreal Editor's built-in tools for environment creation, material authoring, and lighting setup.)

Input

- Unreal Editor.
- Built-in static meshes (from Starter Content) or imported 3D models.
- User interaction with various editor modes and panels.

Expected Output

- A new Unreal Engine level with a sculpted landscape.
- The landscape has a basic material applied, giving it color and surface properties.
- The scene is lit by a directional light and sky light, creating shadows and ambient illumination.
- Atmospheric effects like sky atmosphere and fog are visible.
- Instances of foliage (e.g., bushes, trees) are scattered across the landscape.
- The overall scene looks like a simple, believable outdoor environment.

Lab 15: Create classes with Blueprints in Unreal Engine.

Title

Create classes with Blueprints in Unreal Engine.

Aim

To understand and utilize Unreal Engine's Blueprint visual scripting system by creating custom Blueprint classes, adding components, and implementing basic interactive logic.

Procedure

1. **Open an Unreal Engine Project:**
    - Open an existing Unreal Engine project (e.g., from Lab 14).
2. **Create a New Blueprint Class:**
    - In the Content Browser, right-click and select `Blueprint Class`.
    - The "Pick Parent Class" window appears. This determines what kind of object your Blueprint will be.
        - Select `Actor` for a general-purpose object that can be placed in the world (e.g., a door, a collectible item).
        - Select `Pawn` or `Character` if you want to create a player-controlled entity.
    - Name your Blueprint (e.g., "BP_InteractiveCube", "BP_Door").
3. **Open and Explore the Blueprint Editor:**
    - Double-click your newly created Blueprint in the Content Browser to open the Blueprint Editor.
    - **Components Tab:** Add new components (`+ Add` button) like `Static Mesh`, `Point Light`, `Box Collision`, etc.
    - **Viewport Tab:** Visualize your Blueprint's components.
    - **Event Graph Tab:** This is where you add visual script logic using nodes.
4. **Add a Static Mesh Component:**
    - In the Blueprint Editor's `Components` tab, click `+ Add > Static Mesh`.
    - Select the `Static Mesh` component. In the `Details` panel, under `Static Mesh`, choose a mesh (e.g., `Cube`, `Sphere` from Starter Content).
5. **Implement Basic Interaction Logic (Example: Toggle Light on Click):**
    - In the `Components` tab, click `+ Add > Point Light`.
    - Go to the `Event Graph` tab.
    - Right-click in the empty graph area and search for `Event BeginPlay`. This node executes when the game starts.
    - Right-click and search for `Event ActorOnClicked`. This node executes when the Blueprint actor is clicked in the game.
    - From the `Point Light` component in the `Components` tab, drag it into the Event Graph to get a reference.
    - From the `Point Light` reference node, drag a wire and search for `Toggle Visibility`. Connect the `Event ActorOnClicked` execution pin to the `Toggle Visibility` execution pin.
    - **Compile** and **Save** your Blueprint (buttons in the top-left of the Blueprint Editor).
6. **Place Blueprint in Scene and Test:**
    - Drag your Blueprint from the Content Browser into your game level.
    - Play the game (`Play` button in the main toolbar).
    - Click on your Blueprint actor in the game. The light should toggle on/off.

Source Code (Blueprint Visual Scripting - Described)

- **Event Graph for `BP_InteractiveCube` (Toggle Light on Click):**
  - **Nodes:**
    - `Event BeginPlay` (Optional: Can be used to set initial state)
    - `Event ActorOnClicked` (Triggered when the actor is clicked)
    - `Point Light` (Reference to the Point Light component)
    - `Toggle Visibility` (Function node to toggle the visibility of a component)
  - **Connections:**
    - The execution pin of `Event ActorOnClicked` connects to the execution pin of `Toggle Visibility`.
    - The `Point Light` reference node connects to the `Target` input pin of `Toggle Visibility`.

Input

- User clicks on the Blueprint actor in the game world.
- Selection of components and nodes within the Blueprint Editor.

Expected Output

- A new Blueprint class is created, inheriting from the chosen parent class (e.g., Actor).
- The Blueprint contains a static mesh component, making it visible in the world.
- The Blueprint contains a point light component.
- When the Blueprint actor is placed in the scene and the game is run, clicking on the actor with the mouse toggles the visibility of the internal point light.
- The Blueprint is compiled and saved successfully.