COMSM0129: Augmenting the Real World

Lab 4: Intelligent Tracking (Learning-based AR)

Overview

In the previous labs, we focused on geometric tracking using existing modules or writing simple C# scripts. In this lab, we will explore "Learning-based AR" in AR Foundation. Specifically, we will focus on

- Using Plane Classification to distinguish floors from tables.
- Performing 3D Human Body Tracking to visualise skeletons.
- Enabling Environment Occlusion using depth estimation.

Tasks

1. **Plane Classification (Semantics):** Colour-code planes based on what they are (Floor vs. Table).
2. **Human Body Tracking:** Track and visualise a human skeleton in 3D.
3. **Environment Occlusion:** Allow real-world objects to conceal virtual content.

Task 1: Plane Classification

Modern AR frameworks (e.g., ARCore on the Google Pixel 8) use Neural Networks to classify surface types, which is referred to **Semantics**. In this task, we will modify our plane prefab to change its colour automatically.

Subtask 1: Set up the Materials

1. Open your project (with the same settings as in Lab 1). In the `Assets` folder, create a new folder named `Materials`.
2. Right-click inside it → `Create` → `Material`. Name it `Mat_Floor`.
3. In the Inspector, set the **Base Map** colour to **Green** (or a colour you prefer).
4. Repeat to create `Mat_Table` (**Blue**) and `Mat_Other` (**Red**).
5. **Important:** Select all three materials. In the Inspector:
   - Set **Surface Type** to `Transparent`.
   - Click the Colour box again and lower the **Alpha (A)** value to `100` (about 40%) so that we can see through the planes.

Subtask 2: Create the Semantic Script

We need a script attached to the individual Plane Prefab which checks its own classification.

1. In `Assets/Scripts`, create a C# script named `SemanticPlane`.
2. Open the script. We need to access the `ARPlane` component.
3. The example code is given below.

```
using UnityEngine;
using UnityEngine.XR.ARFoundation;
using UnityEngine.XR.ARSubsystems;

public class SemanticPlane : MonoBehaviour
{
    [SerializeField] private Material _floorMat;
    [SerializeField] private Material _tableMat;
    [SerializeField] private Material _otherMat;

    private ARPlane _arPlane;
    private MeshRenderer _renderer;

    void Awake()
    {
        _arPlane = GetComponent<ARPlane>();
        _renderer = GetComponent<MeshRenderer>();
    }

    void Update()
    {
        // Check if the plane is classified as a Floor
        if (_arPlane.classifications.HasFlag(PlaneClassifications.Floor))
        {
            _renderer.material = _floorMat;
        }
        // Check if the plane is classified as a Table
        else if (_arPlane.classifications.HasFlag(PlaneClassifications.Table))
        {
            //Write your code here...
        }
        // All other types (Wall, Ceiling, None, etc.)
        else
        {
             //Write your code here...
        }
    }
}
```

Subtask 3: Update the Prefab

1. Locate your `AR Default Plane` prefab in the `Assets` folder. Double-click it to open **Prefab Mode**.
2. Add the `SemanticPlane` script component to it.
3. Drag your materials (`Mat_Floor`, `Mat_Table`, `Mat_Other`) into the corresponding slots.
4. Save and exit Prefab Mode.

Subtask 4: Critical Configurations (Troubleshooting)

Note: If your planes do not change colour or nothing is detected, it is likely that you have missed one of these setup steps. Please follow the procedures carefully:

1. **Assign the Correct Prefab to Manager:**
   - Select **XR Origin** in the Hierarchy.
   - Find the **AR Plane Manager** component.
   - Look at **Plane Prefab**. Is it using the default Unity asset? This is incorrect.
   - **Fix:** Drag your modified **AR Default Plane** from the **Assets** folder (Project window) into the **Plane Prefab** slot.
2. **Check for Empty Material References:**
   - Open your **AR Default Plane** in Prefab Mode.
   - Check the `SemanticPlane` script component.
   - Are `Mat_Floor`, etc., set to **None**? If so, the script will crash.
   - **Fix:** Drag the corresponding materials into the script slots. Also, ensure your Materials have an Alpha > 0 (not invisible).
3. **Fix Yellow/Black Screen (URP) issues:**
   - Go to `Assets > Settings`, select `Mobile_Renderer`.
   - In Inspector, click **Add Renderer Feature** → **AR Background Renderer Feature**.
   - Ensure **Target Architectures** (Player Settings) is set to **ARM64**.
4. **Build and Run:** Now build to your Pixel 8. Scan the floor and tables.

Task 2: Human Body Tracking

In this task, we will use the `ARHumanBodyManager` to track and visualise a person in 3D.

Subtask 1: Scene Configuration

1. Open your main scene. Select the `XR Origin`.
2. In the Inspector, find the `AR Plane Manager`. **Uncheck** the box to disable it. (This isolates the body tracking feature).
3. Click `Add Component` and search for **AR Human Body Manager**.
4. Ensure `Pose 2D` and `Pose 3D` are ticked.

Subtask 2: Create a Custom Visualizer

Since the `Line Renderer` does not know where to draw lines automatically, we need a script to connect the joint data.

1. In `Assets/Scripts`, create a new C# Script named `SkeletonVisualizer`.
2. Write the code with the following:

```
using UnityEngine;
using UnityEngine.XR.ARFoundation;
using UnityEngine.XR.ARSubsystems;
using Unity.Collections;

public class SkeletonVisualizer : MonoBehaviour
{
    private ARHumanBody _arBody;
    private LineRenderer _lineRenderer;

    void Awake()
    {
        _arBody = GetComponent<ARHumanBody>();
        _lineRenderer = GetComponent<LineRenderer>();
    }

    void Update()
    {
        // Check if we are actively tracking a person
        if (_arBody.trackingState != TrackingState.Tracking)
        {
            _lineRenderer.positionCount = 0;
            return;
        }

        var joints = _arBody.joints;

        if (!joints.IsCreated || joints.Length == 0)
        {
            _lineRenderer.positionCount = 0;
            return;
        }

        // Draw lines between all joints
        _lineRenderer.positionCount = joints.Length;

        for (int i = 0; i < joints.Length; i++)
        {
            Vector3 jointLocalPos = joints[i].localPose.position;
            Vector3 worldPos = transform.TransformPoint(jointLocalPos);
            _lineRenderer.SetPosition(i, worldPos);
        }
    }
}
```

Subtask 3: Create the Body Prefab

1. In the Hierarchy, right-click → **Create Empty**. Name it `AR Human Body Prefab`.
2. Add Component → **AR Human Body**.
3. Add Component → **Line Renderer**.
4. Add Component → **Skeleton Visualizer** (the script you just created).
5. **Configure Line Renderer:**
   - Set **Width** to `0.02`.
   - Set **Material** to `Default-Line` (or `Sprites-Default`).
   - Ensure **Use World Space** is checked.
6. **Turn into Prefab:** Drag this object into your `Assets` folder, and then delete it from the Hierarchy.
7. **Assign:** Select `XR Origin`, find **AR Human Body Manager**, and drag your new prefab into the **Human Body Prefab** slot.

Subtask 4: UI Debugging

Here, we implement adaptive monitoring for our detection status.

1. Right-click Hierarchy → **UI** → **Text - TextMeshPro**. Set Font Size to 80 and Colour to **Red**.
2. Create a script named `BodyDebug` and attach it to the Text object:

```
using UnityEngine;
using TMPro;
using UnityEngine.XR.ARFoundation;

public class BodyDebug : MonoBehaviour
{
    public ARHumanBodyManager bodyManager;
    private TMP_Text _text;

    void Start() => _text = GetComponent<TMP_Text>();

    void Update()
    {
        if (bodyManager == null) return;

        int count = bodyManager.trackables.count;
        var subsystem = bodyManager.subsystem;

        if (subsystem != null && !subsystem.running)
            _text.text = "AR Subsystem Stopped!";

        // No body found
        //Write your code here....

        // Body detected.
        //Write your code here....
    Count: " + count;
    }
}
```

3. In the Inspector, drag **XR Origin** into the **Body Manager** slot of the script.
4. **Build and Run**. Read the text on screen:
   - **"Scanning..."**: AR is working, but no person is detected (check distance/light).
   - **"Body Detected!"**: Person found. If lines are missing, check LineRenderer settings.
   - **"Stopped!"**: Device not supported or permission denied.

Subtask 5: Testing

1. **Build and Run.**
2. Point the camera at a person (standing 2-3 metres away).
3. You should see a white line skeleton overlay.

Task 3: Environment Occlusion (Optional)

Occlusion allows the real world to hide virtual objects. **Note:** If you are testing Task 2 (Skeleton), disable this feature first, as the person might occlude their own skeleton!

Subtask 1: Enable Occlusion

1. Select `XR Origin` → `AR Camera` (or Main Camera).
2. Add Component → **AR Occlusion Manager**.
3. Set `Environment Depth Mode` to `Best`.
4. Set `Human Segmentation Stencil/Depth Mode` to `Best`.

Subtask 2: Verify the Effect

1. Right-click Hierarchy → `3D Object` → `Cube`.
2. Set the Position to `(0, 0, 2.0)` (2 metres forward). Scale to `(0.2, 0.2, 0.2)`.
3. **Build and Run**.
4. Ask a friend to stand in front of the virtual cube, or put your hand in front of the camera. The real object should properly obscure the virtual cube.

Conclusion

In this lab, we utilised the powerful AI capabilities of the Pixel 8 and AR Foundation in Unity 6. We implemented **Semantic Classification** (Task 1), **3D Body Tracking** using a custom visualiser (Task 2), and **Depth Occlusion** (Task 3). These "Learning-based" features allow the AR system to understand the context of the real world, not just its geometry.