COMSM0129: Augmenting the Real World

Lab 2: Keeping track of trackables and images

Overview

The purpose of this lab is to understand how to write scripts to interact with different AR Foundation components. We will do this by implementing a simple functionality in our App: to count and display the number of planes and points being tracked by the AR system.

Tasks

1. Import pre-built AR assets into the scene and modify them.
2. Add a script component to XR Origin and implement callbacks for AR trackable managers.
3. Perform image detection and tracking.

Task 1: Import assets

Firstly, open the Unity project that you worked on in the last lab session, where you added an ARPlaneManager and an ARPointCloudManager component to the scene and attached prefabs to them. Alternatively, you can open a new project and go through all the tasks in the last lab session. The starting point of this lab is that you have an AR app that displays a live video fed from the smartphone camera and visualises the detected ARPlane and ARPointCloud.

We will make use of a Unity package that contains several pre-built assets for our lab projects. Please follow the steps below to download it and import it into your Unity project.

1. Download the package from here (also available on BB - Code and resources: **MobileARCourse.unitypackage**).
2. In your Unity editor, go to `Assets > Import Package > Custom Packages`, and select the `.unitypackage` file you just downloaded.
3. In the pop-up window, keep everything selected and click `Import`. You should then see a folder named `MobileARCourse` in the `Assets` folder.
4. The specific asset we will be using in this lab session is the `ARCourseCanvasPrefab` located under `MobileARCourse/Prefabs`. Locate it in the editor's project folder, and drag it onto the `Main Camera` object in the scene hierarchy.
5. Take a look at the components of the prefab. By clicking on the individual components and looking at the Inspector, we can see that the components include several `TextMesh Pro` text fields and images.
6. In this session, we will only use the `TMPState` and `TMPPlanes` components, so keep them and disable the other components. For example, to disable the `Image` and `Button` components of `FireButton`, click `FireButton` and deselect the checkboxes for those components in the Inspector.

Task 2: Write a script

Now that we have all the components ready in the scene, we need to write some code to implement the App feature: displaying the system state and the number of trackables being tracked. Click `XR Origin` in the scene hierarchy, and in the Inspector window, click `Add Component`, and add a new script; name it "ARUnitLab". Double click the c# script that just appeared in the project folder and edit the script. Follow the following subtasks to complete the script.

Subtask 1: Display system state

The function we want to implement in this task is to display the AR system state on the screen using the `TMPState` object. To do so, we need:

- A mechanism for detecting changes in the system state.
- A callback function in which we specify what to do when the system state changes.
- A reference to the TMP component so that we can update what it displays.

The system lifecycle is controlled by the `ARSession` class. Take a look at the documentation here. Notice that `ARSession` invokes a `stateChanged` event with the argument `ARSessionStateChangedEventArgs`. Also, have a look at the docs for the event args here. After you familiarise yourself with the classes mentioned above, follow the steps below to implement this functionality.

1. Add a reference to the `TMPState` as an attribute of the `ARUnitLab` class.

```
[SerializeField] private TMP_Text _stateText;
```

2. Since we are using the class `TMP_Text` in the package `TMPro`, we also need to include it at the top of our script:

```
using TMPro;
```

3. Implement a callback function named `OnARSessionStateChanged` with the declaration below. The function should update `_stateText.text` with the current session state.

```
private void OnARSessionStateChanged(
    ARSessionStateChangedEventArgs args)
{
    // Do something with args...
    // e.g. _stateText.text = args.[XXX]
    // Hint: you may want to look at what
    properties the object args have. Once you
     found the interesting property, look at
    what functions that property has.
}
```

4. Include the ARFoundation package

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

5. Add the callback function to the ARSessionStateChanged event in the Start() function:

```
ARSession.stateChanged +=
    OnARSessionStateChanged;
```

6. Go back to the Unity editor, and the editor will reload the script. While doing so, if there are any bugs in your script, they will be reported.

7. Click XR Origin in the scene hierarchy and look at the script component you added previously. You should see that it now has a new attribute called State Text. Drag the TMPState component from the scene hierarchy to the State Text field of your script.

8. Build and test the functionality on your device.

Subtask 2: Display the number of tracked planes

In the previous task, we used the ARSession class to detect and manage system state changes. The new task now is to keep track of the number of planes and points being tracked by the system and display the count on the screen using the TMPPlanes object. To do so, we now need to interact with two manager classes: ARPlaneManager and ARPointCloudManager. We need to keep track of every new plane/point they detect and remove it from our storage when the managers decide to stop tracking it. Follow the steps below to implement this functionality in your "ARUnitLab.cs" script.

1. Take a look at the docs of ARPlaneManager and ARPointCloutManager and find out
   a) the names of the events they trigger,
   b) what the event argument contains

2. Add [SerializeField] variables for ARPlaneManager and ARPointCloudManager as attributes for your ARUnitLab class (the same way you did for your _stateText variable). E.g.

```
[SerializeField] private ARPlaneManager
    _arPlaneManager;
```

3. Add a callback for the planesChanged event. The procedure is the same as how you added the callback for ARSession, but, of course, with a different callback implementation. Also, note that the stateChanged event is a static event, while the events for our plane and point cloud managers are not. Therefore, we need to add callbacks to their instances, e.g.,

```
_arPlaneManager.planesChanged +=
    OnPlanesChanged;
```

4. Implement the callback function you just added. E.g., the OnPlanesChanged function above:

```
private void OnPlanesChanged(
    ARPlanesChangedEventArgs args)
{
    // Do something with args
}
```

5. Write code in the callback function you just added to keep track of the number of ARPlanes as they are added and removed. [**Hint**: you may need to instantiate a List<ARPlane> object as a member of your class to keep a record of the tracked planes.]

6. Add a [SerializeField] for the TMPPlanes object as you did for TMPState:

```
[SerializeField] private TMP_Text _planeText;
```

7. In the Update() function, write code to update the text displayed by _planeText with the number of planes being tracked.

8. Go back to the Unity editor and look at the Inspector window for XR Origin in the scene hierarchy. You should see that the Lab 2 script component now has two additional fields: Plane Text and AR Plane Manager. Drag the AR Plane Manager component above to the appropriate empty field of Lab 2. Also, drag the TMPPlanes in the scene hierarchy to the appropriate field in Lab 2 component.

9. Build the App and test the functionality.

10. (Optional) When you are running the App, you may notice that the planes sometimes get merged. If you have only handled cases where planes are added or removed, then your count of the tracked planes will be incorrect! Think about how you can modify the OnPlanesChanged callback to handle the cases where planes are merged. [**Hint**: the ARPlanesChangedEventArgs class also has a property named updated. An ARPlane object has a property called subsumedBy.]

Task 3: Image Tracking

Image tracking is a process in which a computer vision system detects and locates a given image. Given the actual size of the image and the camera parameters, the system can compute the relative pose of the image in the real world using a single camera frame; this pose allows us to create different 3D effects, such as projecting a 3D model onto the detected image or accurately locating the camera pose. In this task, we will first perform simple image tracking with Android phones and then visualise the tracked image pose by projecting a 3D object onto it.

Subtask 1: Prepare target images

1. Load your previous project.

2. In the scene hierarchy, click `XR Origin`, then add a component named `AR Tracked Image Manager`. The `AR Tracked Image Manager` will automatically perform image tracking with a given set of target images.

3. Download sample images from the internet that you will use as the tracking targets; e.g., you can download playing cards from here. In the following, we will use the files from this link as an example.

4. Right click the empty space in the `Assets` window and select `Import New Asset`. Find a PNG image (e.g., `king_of_clubs2`) in the downloaded files and import it. As image tracking requires the target image to contain sufficient features that can be tracked, if selected images do not contain enough features, e.g., the `king_of_clubs` image, Unity will raise an error when you build your project.

5. Right click the empty space in `Assets` window and select `Reference Image Library` in `Create > XR`. Click the created asset to open the inspector, choose `select` and find the imported image. Enable `Specify Size`, and input the physical size `(X:0.086 and Y: 0.124872)`, which is a typical size for playing cards.

6. You can also add more tracked images by repeating the above steps.

Subtask 2: Track and visualise the images

1. In the `Scene Hierarchy` window, select XR Origin. In the `AR Tracked Image Manager` box, configure the created `Reference Image Library` as the `Serialized Library`.

2. The application is now ready to perform tracking. To visualise the tracking, we can project some 3D objects onto the tracked images. Here, we use a 3D robot model as an example, which is included in the package we are using.

3. First, search for a prefab called `robot` in the project window. You will see two `robot` files; the prefab has the file named `robot.prefab`. Drag it to the `Scene Hierarchy` to create an object, and then drag the object to the `Assets` window to create a new prefab. Choose 'Original Prefab' if you see a pop up prompt. Delete the object created in the `Scene Hierarchy`.

4. Choose the created prefab in the `Assets` window and adjust the scale (x,y,z) by 0.25 so that the robot will not be too large when visualising it.

5. Select `XR Origin` in the Scene Hierarchy; under the box of AR Tracked Image Manager, click `Tracked Image Prefab` and select the created robot prefab. This allows the AR Tracked Image Manager to generate objects when an image is detected and update the objects pose when the image is moved.

6. Now, let us build and run the application. You can simply use the desktop monitor to show the playing card to test the tracking. You may need to move the phone closer to the target image to detect it. Once the image has been detected, it can be tracked even if you move your phone further away from it.

Conclusion

In this lab, we learnt how to implement features for an AR App using C# scripting and the ARFoundation package. We moved beyond simple visualisation to **interaction**, learning how to listen to system events and track specific 2D images. ARFoundation provides a comprehensive API, and we only need to know which components to use for our purposes and how to use them. Fortunately, ARFoundation also comes with extensive documentation, so we should get used to constantly referring to the documentation when developing our AR App.