COMSM0129: Augmenting the Real World

Lab 3: Interacting with AR

Overview

In this lab session, we will learn how to interact with reality through our AR App. The core functionality in AR Foundation that enables interactions is called Raycasting. We will cast rays to the scene when the user touches the screen and instantiate a robot where the cast ray successfully collides with our target.

Note: make sure that you have finished the previous lab tasks as we will need to continue working on the same project.

Tasks

1. Continue working on the script from the previous lab sessions to detect screen touches from the user and perform a Raycast from the touch position.
2. Instantiate a robot prefab where the ray collides with an ARPlane.

Task 1: Raycasting

The purpose of this task is to take the smartphone screen touch events and interpret them in the context of the AR environment. To allow such functionalities, Unity has provided a Raycast API in the ARRaycastManager object. While you can also cast a ray to points in a point cloud, in this session, we will look at raycasting onto planes. For this method, we will use a restricted trackable type called `PlaneWithinPolygon` so that the raycast only matches a plane when the user touches it within the visualised area.

Subtask 1: Detect screen touches

1. Load your previous project into the Unity Editor.
2. In the `Update()` method of the `ARUnitLab` class, write code to detect screen touches and get the coordinates of the touch point [**Hint**: take a look at the `Input` class of `UnityEngine` here, especially those properties and methods related to the keyword “touch”.].

Subtask 2: Cast a ray

1. As the `Raycast` method is within an `ARRaycastManager` object, we need to instantiate such an object first. In your `ARUnitLab` class, add a `SerializeField` to an `ARRaycastManager` object.

```
[SerializeField] private ARRaycastManager
    _arRaycastManager;
```

2. Now, going back to the `Update` function, add code there to call the `Raycast` function using the coordinates of the screen touch that you detected previously. You will notice that the `Raycast` function requires more arguments than simply the position of the raycast. [**Hint**: take a look at the documentation of `ARRaycastManager` here; you may need to instantiate a list at the beginning of the class to store the raycast hits. Also, remember that we would like the ray to hit a **plane** within a polygon.]

```
_arRaycastManager.Raycast(arg1, arg2, arg3);
```

3. Note that `TrackableType` enum is defined in `UnityEngine.XR.ARSubSystems`, so make sure you import this package with a `using` statement.
4. Define a boolean variable that stores whether the raycast was successful. [**Hint**: have a look at what `Raycast` function you used above returns.]

Task 2: Instantiate a Robot

Now that you have cast a ray from the location of the screen touch, let us instantiate a robot when the ray hits a plane.

1. First, we need to let the AR system know what to instantiate, so add a `[SerializeField] GameObject` variable to your class (we will wire this to our robot prefab later):

```
[SerializeField] private GameObject
    _robotPrefab;
```

2. In the `Update()` function, if the raycast is successful, add code to retrieve the information about the hit. Specifically, you should create an `ARRaycastHit` object (see documentation here) and extract the `pose` attribute of the raycast hit. [**Hint**: check the documentation of `ARRaycastManager.Raycast` function carefully to see where the raycast hits are stored.]

```
ARRaycastHit hit = ???;
var hitPose = hit.???;
```

3. Since our goal is to create a robot in the scene, we need a variable to reference the instantiated (spawned) robot so that we can keep track of whether it has been instantiated. Therefore, add an attribute to your class to reference the spawned object and assign an appropriate initial value to it:

```
private GameObject _spawnedObject = ???;
```

4. In the `Update()` function, implement the following.

a) If the robot has not been spawned, instantiate the robot at the location of the raycast hit using the GameObject.Instantiate method, passing in appropriate arguments that correspond to the pose information of the raycast hit (check the documentation of the function here).

b) If the robot has already been instantiated, then update its position (see documentation of GameObject to learn how to do this).

5. Once you have finished writing the code, go back to the Unity Editor. The editor will first try to compile your code and report any errors that occur. Once you finish debugging, click the XR Origin in the scene hierarchy and check your script in the Inspector. You will see that there are new fields that you need to wire components to. To wire an ARRaycastManager, you need to first create an ARRaycastManager by clicking "Add Component".

6. For the robot prefab, drag the robot prefab from the Assets/MobileARCourse/Prefabs/ folder to the field of your script.

7. Build and test your application.

Task 3: Adding 3D Motions

Using image tracking (Lab 2), we can track images and visualise 3D effects. However, using the default Tracked Image Prefab will only generate static objects, which will not allow us to create interesting applications. Like in the previous lab, we can customise the behaviour of the Tracked Image Manager by writing a script.

1. Based on the project created for image tracking (Lab 2), remove the Tracked Image Prefab by setting it to None.

2. Replace the player card image with a QR code, which is more commonly used in AR applications. A sample QR code is provided on Blackboard [Download from here].

3. Open the script from lab 2 or follow the instructions to add an empty script.

4. In the script, add an ARTrackedImageManager object, which will be linked to the AR Tracked Image Manager later.

```
[SerializeField] private ARTrackedImageManager _arTrackeddImageManager;
```

5. Additionally, add a game object prefab, which we will use to initialise the game object once the target image has been detected.

```
[SerializeField] private GameObject robotPrefab;
```

6. Since the created game object will be updated continually, you will need to declare the object variable in the class.

```
private GameObject robot = null;
private float theta = 0f;
```

7. You will also need to add an event handler for handling image detection and tracking updates. For more details on ARTrackedImageManager and ARTrackedImagesChangedEventArgs, see here.

```
void OnEnable(){
    _arTrackeddImageManager.trackedImagesChanged += OnTrackedImagesChanged;
}
void OnDisable(){
    _arTrackeddImageManager.trackedImagesChanged -= OnTrackedImagesChanged;
}

Continue coding....
            ??????
```

8. Once you achieve image detection and obtain tracking updates, you can calculate the pose of the image and perform different animations according to the pose. Try to create a robot when the target image is detected and enable the robot to walk in a circle around the tracked QR code with the following function:

$$position = < 0.25 sin\theta, 0, 0.25 cos\theta >$$

$$angle = < 0, \theta + 180^\circ, 0 >$$

where θ is changing over time.

9. To achieve this, you can either directly update the transform of the robot or use transform.Translate and transform.Rotate. See here for more details.

```
?????
```

10. To update the game object behaviour, you are advised to store the pose of the image and use the function Update instead of OnTrackedImagesChanged to update the object's behaviour, such that it will remain unaffected even if the image is not being tracked in every moment.

11. Lastly, return to Unity Editor. You will see new fields named AR Tracked Image Manager and Robot Prefab. Drag the AR Tracked Image Manager component to the AR Tracked Image Manager field and select the robot prefab in the Robot Prefab field.

12. Your code is ready to run.

13. When you complete this task and still have time, you can develop code to generate more complicated 3D motions/effects following the Unity Animation manual here.

Conclusion

In this session, we learnt how to interact with the user of our AR App via screen touches. We also learnt how to perform raycasting and detect successful raycast hits. We saw how a virtual object can be instantiated in the scene. Moreover, we learnt how to create simple 3D motions with ARFoundation. These are highly relevant to the coursework of this unit.