Project report on "Advanced interactions" subject
Prepared by
Shcherbakova Kateryna
Master Informatique 2, Ingénierie logicielle

Introduction

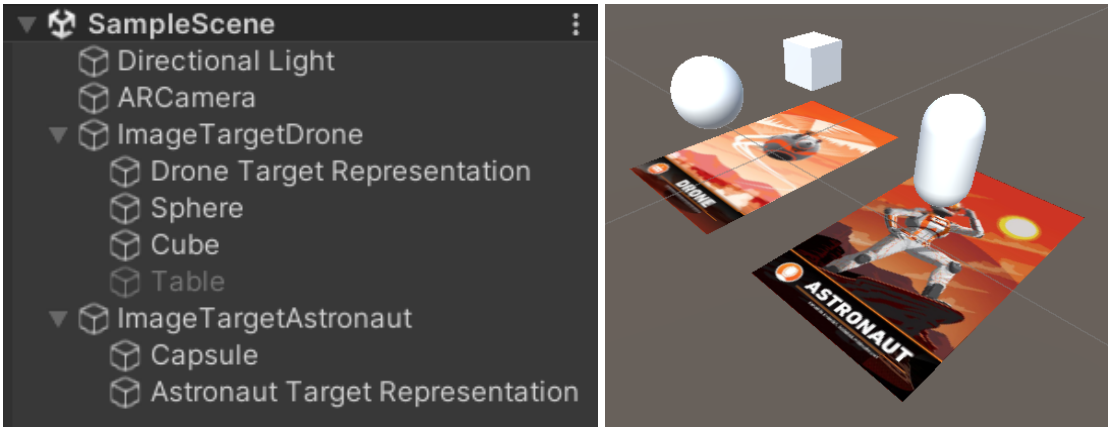
This report provides an overview of Augmented Reality (AR) application development using the Vuforia SDK within Unity 3D. It delves into fundamental concepts and practical exercises to integrate digital content seamlessly with the real world. This report covers ARCamera setup, ImageTarget customization, virtual occlusion, and Android deployment, illustrating key concepts and techniques fundamental to AR integration.

Exercise 1: ARCamera Setup and Configuration

During this exercise, I configured the ARCamera within Unity using the Vuforia SDK for an augmented reality application. The primary steps included creating a new Unity project, integrating the ARCamera GameObject from the Vuforia Engine package, and verifying the correct display of the video feed within the Unity editor. This exercise provided an initial experience in setting up the ARCamera, which served as a foundation for diving deeper into building augmented reality applications.

Exercise 2: ImageTarget Customization

For the second exercise, I explored ImageTarget customization in the AR application. I created multiple simple 3D shapes and associated them as child objects to the Image Targets. Running the application allowed me to verify whether the 3D shapes appeared correctly when the real image was within the camera viewport.



Question 1. What happens when you add the image target representation?

Adding the image target representation involved enabling the visualization of the target during play by pressing the "*Add target representation*" button in the ImageTargetBehaviour component. This action demonstrated a 3D representation of the actual image target, helping to see how the real object and the virtual content

relate in space. It allowed for a clearer understanding of how the AR system links the real and virtual elements.

Question 2. What happens if some of the game objects are affected by the unity physics?

I explored how these physics-based interactions affected the behavior of objects when subjected to forces, collisions, or constraints within the AR space.
During the experimentation, I attached a sphere to the ImageTarget and enabled physics for it by adding a Rigidbody component and activating the gravity via the "Use Gravity" parameter. As a result, the sphere fell infinitely downwards. However, when I placed a virtual table underneath it, the sphere landed on the table surface instead. This showcased how enabling physics affected the behavior of the sphere when interacting with different environmental elements.

Exercise 3. Adding a Secondary Image Target and Understanding "World Center Mode"

In this exercise, I added an additional Image Target to the AR environment. By integrating a second Image Target, I analyzed the AR application's response and functionality with multiple targets enabled simultaneously.

Question 1. Explore and explain the “World Center Mode”. Does it impact the behavior you defined in the second exercise?

During this exercise, I examined the "World Center Mode" functionality within the context of two distinct Image Targets - the Drone and Astronaut. I specifically explored three main modes within Vuforia's "World Center Mode": "*First Target*", "*Specific Target*", and "*Device*".

1. **"First Target" Mode:**

When set to "First Target," Vuforia uses the initial Image Target (Drone in this case) as the reference point or origin for the entire AR scene. All subsequent placements and interactions are relative to this primary target. For instance, placing a virtual object concerning the Drone Image Target creates a spatial relationship starting from that target's position and orientation.

2. **"Specific Target" Mode:**

In "Specific Target" mode, I observed that I could designate a particular Image Target (such as the Astronaut) as the central reference point. The AR elements' positioning and orientation became dependent on this chosen Image Target, disregarding the initial target (Drone). It allows defining a new origin point, potentially altering the entire spatial context within the AR environment.

3. **"Device" Mode:**

When using "Device" mode, Vuforia employs the device's physical position and orientation as the scene's origin. In this scenario, the AR experience becomes more mobile-centric, with the device acting as the reference point. I noticed that

interactions and placements of virtual content were aligned with the device's movements and orientation in the physical space.

Exercise 4. Extended Tracking

Enabling and disabling extended tracking in Vuforia involves activating a feature that allows the system to continue tracking a target even when it's not visible in the camera's view. Here's an explanation of the differences between enabling and disabling extended tracking:

1. **Enabling Extended Tracking:**

When extended tracking is enabled, Vuforia retains information about the target's position and orientation even after it leaves the camera's field of view. Extended tracking allows users to move the camera away from the target and then return to find the virtual content repositioned relative to the remembered location of the target.

2. **Disabling Extended Tracking:**

Disabling extended tracking stops Vuforia from maintaining information about the target's position and orientation beyond its visibility in the camera view. Without extended tracking, if the target moves out of the camera's view and then returns, the virtual content associated with it may not persistently remain in the same location as it would with extended tracking enabled.

Enabling extended tracking provides a more seamless and continuous AR experience by maintaining the spatial relationship between the real-world target and the virtual content even when the target is temporarily not in view of the camera. Disabling extended tracking results in a less persistent AR experience where the virtual content's spatial context may not be retained once the target is no longer visible.

Exercise 5. DefaultObserverEventHandler

The DefaultObserverEventHandler in Vuforia serves to manage the behavior of an Image Target when it becomes visible or invisible within the camera feed. Here's a brief overview of its behavior:

The DefaultObserverEventHandler contains several methods that monitor the status of the Image Target:

1. **HandleTargetStatusChanged**: This method is invoked every time there is a change in the tracking state of the Image Target. It manages the visibility of the objects associated with the target based on its tracking status, like rendering or hiding them.
2. **OnTrackingFound**: This function is called when the Image Target is first detected by the camera feed. It triggers actions that should occur when the target becomes visible, such as displaying associated virtual objects.

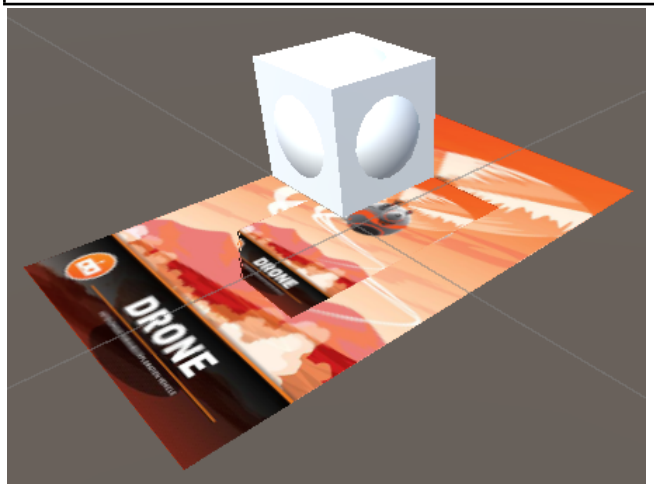
3. **OnTrackingLost**: This method is called when the Image Target, previously visible, is no longer detected in the camera feed. It manages actions or changes in behavior when the target becomes invisible, such as hiding or disabling associated virtual objects.

Exercise 6. Script to Display a Different Object Every Time That the Image Target Becomes Visible

This code extends the functionality of the DefaultObserverEventHandler class to display different 3D objects each time the Image Target becomes visible.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Vuforia;


public class ChangingObjectsEventHandler : DefaultObserverEventHandler
{
    public GameObject[] objectsToDisplay; // all 3D objects

    int currentIndex = 0;

    // when marker is found
    protected override void OnTrackingFound()
    {
        base.OnTrackingFound(); // from default method

        if (objectsToDisplay.Length > 0)
        {
            ShowNextObject();
        }
    }

    void ShowNextObject()
    {
        // Disable all objects
        foreach (var obj in objectsToDisplay)
        {
            obj.SetActive(false);
        }

        // Enable next object
        objectsToDisplay[currentIndex].SetActive(true);

        // index for the next object
        currentIndex = (currentIndex + 1) % objectsToDisplay.Length;
    }
```

```
}
```



This script ensures that every time the Image Target is detected, it triggers the display of the next GameObject from the array, allowing for a sequential presentation of different 3D objects associated with the Image Target.

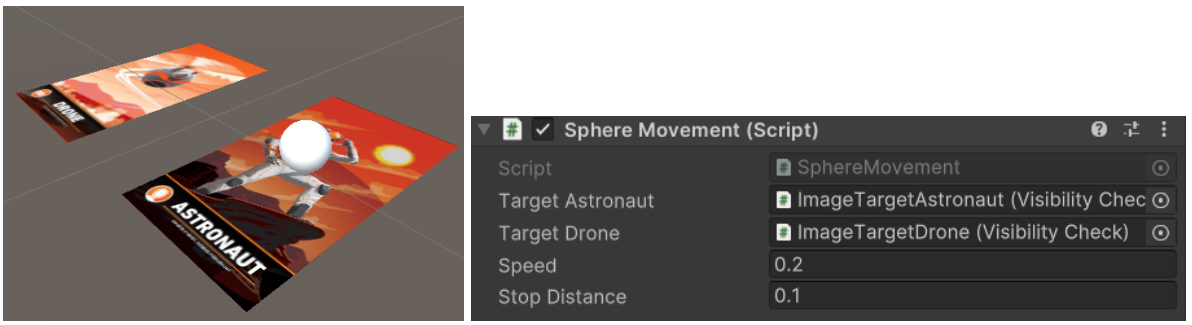
Exercise 7. Android Deployment

Deploying an AR application onto an Android device after testing it within the Unity Editor may present several differences and considerations:

1. **Performance Changes**: Expect variations in performance due to device specs like memory, processor, affecting rendering quality and smoothness.
2. **Device-Specific Factors**: Device features (camera, sensors) impact how AR elements align and behave in the real world.
3. **Real-world Impact**: Lighting changes and surroundings affect object recognition and stability.
4. **User Experience Shifts**: User interactions differ on devices, impacting how users engage with AR content.
5. **Deployment and Debugging**: Deploying differs and debugging might need distinct approaches compared to the Unity Editor.

Exercise 8.1. Rolling Sphere

In this exercise, the objective was to modify the default behavior of the ImageTarget prefab by creating an application scenario where two image targets trigger the display of a virtual ball on one of them. Subsequently, an animation moves the ball towards the other target.



The script *SphereMovement* is attached to a GameObject representing a *Sphere* within the scene.

```
using UnityEngine;

public class SphereMovement : MonoBehaviour
{
    public VisibilityCheck targetAstronaut;
    public VisibilityCheck targetDrone;
    public float speed = 0.3f;

    private Transform currentTarget;
    private bool moving = false;
    private bool hasMoved = false;

    void Update()
    {
        if (targetAstronaut == null || targetDrone == null)
        {
            return;
        }

        bool astronautVisible = targetAstronaut.isVisbile;
        bool droneVisible = targetDrone.isVisbile;

        if (astronautVisible && droneVisible)
        {
            if (!hasMoved)
            {
                // Debug.Log("!!! Sphere position " + transform.position);
                currentTarget = targetDrone.transform;
                hasMoved = true;
            }
            moving = true;
        }
        else
        {
            moving = false;
            hasMoved = false;
```

```
        }

        if (moving)
        {
            MoveTowardsTarget();
        }
    }

    void MoveTowardsTarget()
    {
        // Debug.Log("!!! Moving to target " + transform.position);

        if (currentTarget == null)
            return;

        transform.position = Vector3.MoveTowards(transform.position,
currentTarget.position, speed * Time.deltaTime);


            if (currentTarget == targetAstronaut)
            {
                currentTarget = targetDrone.transform;
            }
            else
            {
                moving = false;
            }
    }
}
```

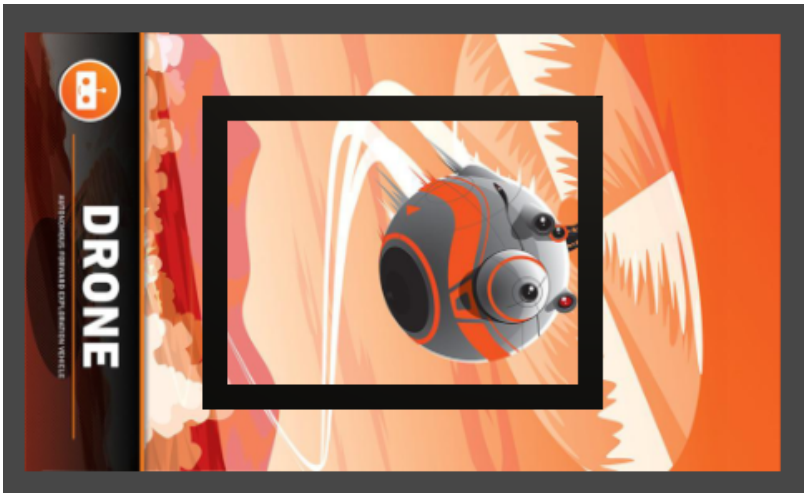
The script *SphereMovement* manages the movement of a virtual sphere between two image targets (*targetAstronaut* and *targetDrone*). When both targets are visible, it triggers the sphere's movement from one target to the other.

1. The script uses the *Update* function to check target visibility and initiate sphere movement.
2. Once both targets are visible, it sets the sphere's initial movement toward the drone target.
3. The *MoveTowardsTarget* function moves the sphere smoothly toward the current target at a specified speed.
4. When the sphere reaches the target, it switches its movement to the other target.

Exercise 8.2. Occlusion

In my demonstration, I created a box by assembling *Cube* primitives, shaping them as walls and positioning them behind the image target. Inside this construct, I placed a *Question Mark* object from *PolygonStarter* asset. To effectively conceal the

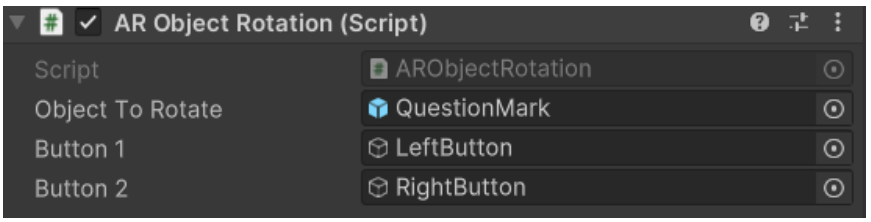
virtual content, I employed a *Plane* GameObject and applied an occlusion shader to its surface using "VR/SpatialMapping/Occlusion". It hides the outside of the box, as it should not be visible behind the real world object. During gameplay, this setup convincingly simulates the question mark being positioned behind the real object, significantly enhancing the integration of virtual and real elements within the scene.



Exercise 8.3. Rotation Control Buttons for Virtual Objects

The script allows two *Cube* GameObjects to function as buttons that enable rotation of a specified object - *Question Mark* - in response to user interactions, toggling between pressed and unpressed states to control the object's rotation direction.

The script is applied to each button object.

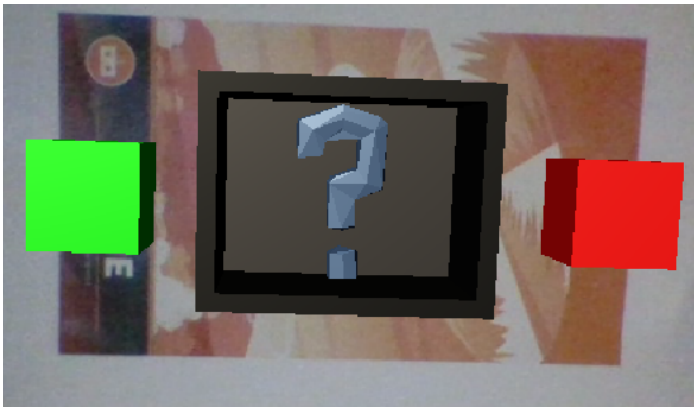


Here's a brief overview of what happens in the code:

1. **Initialization**: Upon start, the script initializes the colors of the two buttons, setting one to green and the other to red.
2. **Button Interaction**: The script detects button clicks through the OnMouseDown() method. When a button is clicked:
    a. Toggles the button's pressed state and ensures only one button is pressed at a time.
    b. Changes the color of the clicked button to indicate its pressed status and resets the color of the other button.
3. **Rotation Control**: In the Update() method:
    a. Checks if rotation is enabled (rotationStarted) and if an object to rotate is specified.
    b. Adjusts the object's rotation based on which button is pressed.

```
using UnityEngine;

public class ARObjectRotation : MonoBehaviour
{
   public GameObject objectToRotate;
   public GameObject button1;
   public GameObject button2;

   private bool button1Pressed = false;
   private bool button2Pressed = false;
   private bool rotationStarted = false;

   private float rotationSpeed = 50f;

   private void Start()
   {
      // init colors
      button1.GetComponent<Renderer>().material.color = Color.green;
      button2.GetComponent<Renderer>().material.color = Color.red;
   }

   private void OnMouseDown()
   {

      if (gameObject == button1)
      {
         button1Pressed = !button1Pressed;
         button2Pressed = false;
         rotationStarted = button1Pressed;

         // colors change
         button1.GetComponent<Renderer>().material.color = button1Pressed ?
new Color(0f, 0.5f, 0f) : Color.green;
         button2.GetComponent<Renderer>().material.color = Color.red;
```

```
        }
        else if (gameObject == button2)
        {
            button2Pressed = !button2Pressed;
            button1Pressed = false;
            rotationStarted = button2Pressed;

            button2.GetComponent<Renderer>().material.color = button2Pressed ?
new Color(0.5f, 0f, 0f) : Color.red;
            button1.GetComponent<Renderer>().material.color = Color.green;
        }
    }

    private void Update()
    {
        if (rotationStarted && objectToRotate != null)
        {
            float rotationStep = rotationSpeed * Time.deltaTime;
            Vector3 rotation = objectToRotate.transform.eulerAngles;

            if (button1Pressed && !button2Pressed)
            {
                rotation.y += rotationStep; // left
            }
            else if (!button1Pressed && button2Pressed)
            {
                rotation.y -= rotationStep; // right
            }

            objectToRotate.transform.eulerAngles = rotation;
        }
    }
}
```



Conclusion

This report offered a comprehensive exploration of augmented reality (AR) application development using Unity 3D and Vuforia SDK. The exercises covered critical aspects such as ARCamera setup, ImageTarget customization, occlusion handling, interactive controls, and real-virtual content integration. This report will greatly benefit my future AR projects.