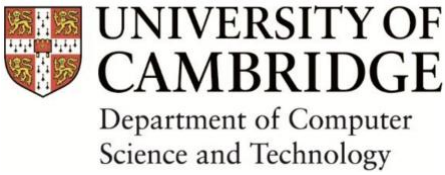

Extended Reality
Course Project Report

[Interactive AR block tower]

Authors: Peter Hu, zh369; 2200C
Group number: 6

Contents

Acronyms

AR Augmented Reality. 1, 2

BRIEF Binary Robust Independent Elementary Features. 5

FAST Features from Accelerated Segment Test. 5

HDR High Dynamic Range. 1, 6, 16

LDR Low Dynamic Range. 6

MR Mixed Reality. 2

N/A Not Applicable. 13, 20

ORB Oriented FAST and Rotated BRIEF. 5

SIFT Scale-Invariant Feature Transform. 5

STOA State-of-the-Art. 6

SURF Speeded-Up Robust Features. 5

UI User Interface. vi, 11

VR Virtual Reality. 2

XR Extended Reality. 2, 16

List of Figures

List of Tables

1 Abstract

Block-stacking games are popular as they encourage creativity and problem-solving. The project aims to provide accessible and inclusive Augmented Reality (AR) gameplay experience. I utilize a series of non-trivial techniques in the project gameplay, e.g. *plane detection, relighting estimation, physical simulation, human occlusion, object detection and virtual object placements*. By integrating such AR features, it further enhances the experience by overlaying digital content that interacts dynamically with the physical world.

To make the world of block tower interactive and joyful, I also focus on how user behaviours can affect the virtual objects. For example, the user can blow the blocks away because of collision detection. Moreover, the block itself is evolving with various unexpected size, geometry, and material, which adds more spice to the experience.

Checkpoints and roll-back are crucial for long-term gameplay. I also introduce a new feature that allows the user to save the current game state and resume any one of them later. This feature is useful for users who want to take a break and continue later.

In this project, I explore beyond the original proposal with more exciting features. High Dynamic Range (HDR) lighting estimation is used to make the virtual objects more realistic, according to the real-world lighting condition. I also introduce a new game mode similar with **bowling**, where different spheres are shot from the camera to knock down the blocks. A new scoring system is designed to reward players for achieving goals. The game is more goal-oriented and engaging.

2 Introduction

§ 2.1 Motivation

Extended Reality (XR) applications can bridge physical and digital play to create a unique and immersive experience. XR technology is an umbrella term that encompasses Augmented Reality (AR), Virtual Reality (VR), and Mixed Reality (MR). In this project, I am exploring AR's potential in enhancing traditional games and creating novel interactive experiences.

AR features of plane detection, relighting estimation, physical simulation, object detection and virtual object placements can be integrated into a block-stacking game to enhance the gameplay experience. By overlaying digital content that interacts dynamically with the physical world, the game can be more engaging and joyful.

The potential of this project is to encourage learning through play by visualizing complex concepts like physics or engineering in a hands-on environment. I will also make gameplay more accessible and inclusive, allowing customization for different age groups and skill levels.

For a visual demonstration, please refer to the below Figure 2.1.

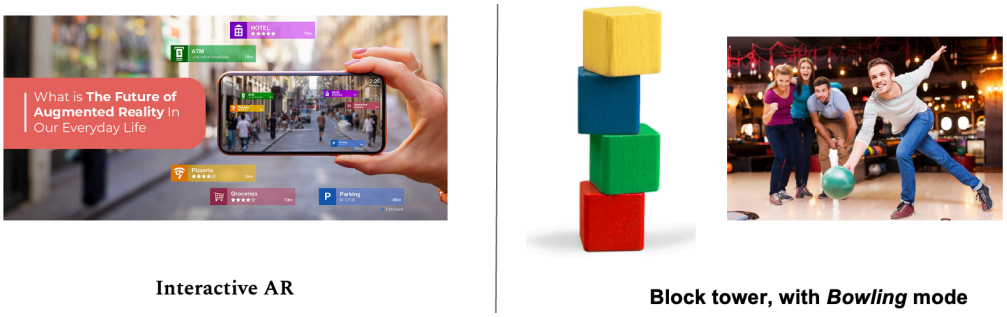

Figure 2.1: Motivation of the project

§ 2.2 Project Proposal

I aim to develop accessible and inclusive AR gameplay with great user experience, which is designed for all age groups and skill levels. It includes the following intended functions,

- merging physical and virtual worlds by applying augmented reality (AR) technology with a physical block-stacking game, i.e., players are able to change the layout according to the rule given.
- proper camera calibration, which ensures the system can detect and track physical blocks accurately and overlay virtual elements in the correct positions.
- implementing gameplay challenges like stability tests or real-time scoring.
- adding at least one educational or creative feature, such as guiding users to recreate specific structures or offering visualizations of physics concepts like balance, force, and center of gravity.

In this project, I explore beyond the original proposal with more exciting features. HDR lighting estimation is used to make the virtual objects more realistic, according to the real-world lighting

condition. I also introduce a new game mode similar with **bowling**, where different spheres are shot from the camera to knock down the blocks. A new scoring system is designed to reward players for achieving goals. The game is more goal-oriented and engaging.

§ 2.3 Screenshots

The below screenshot (Figure 2.2) show the AR block tower with bowling mode in AR Simulator.

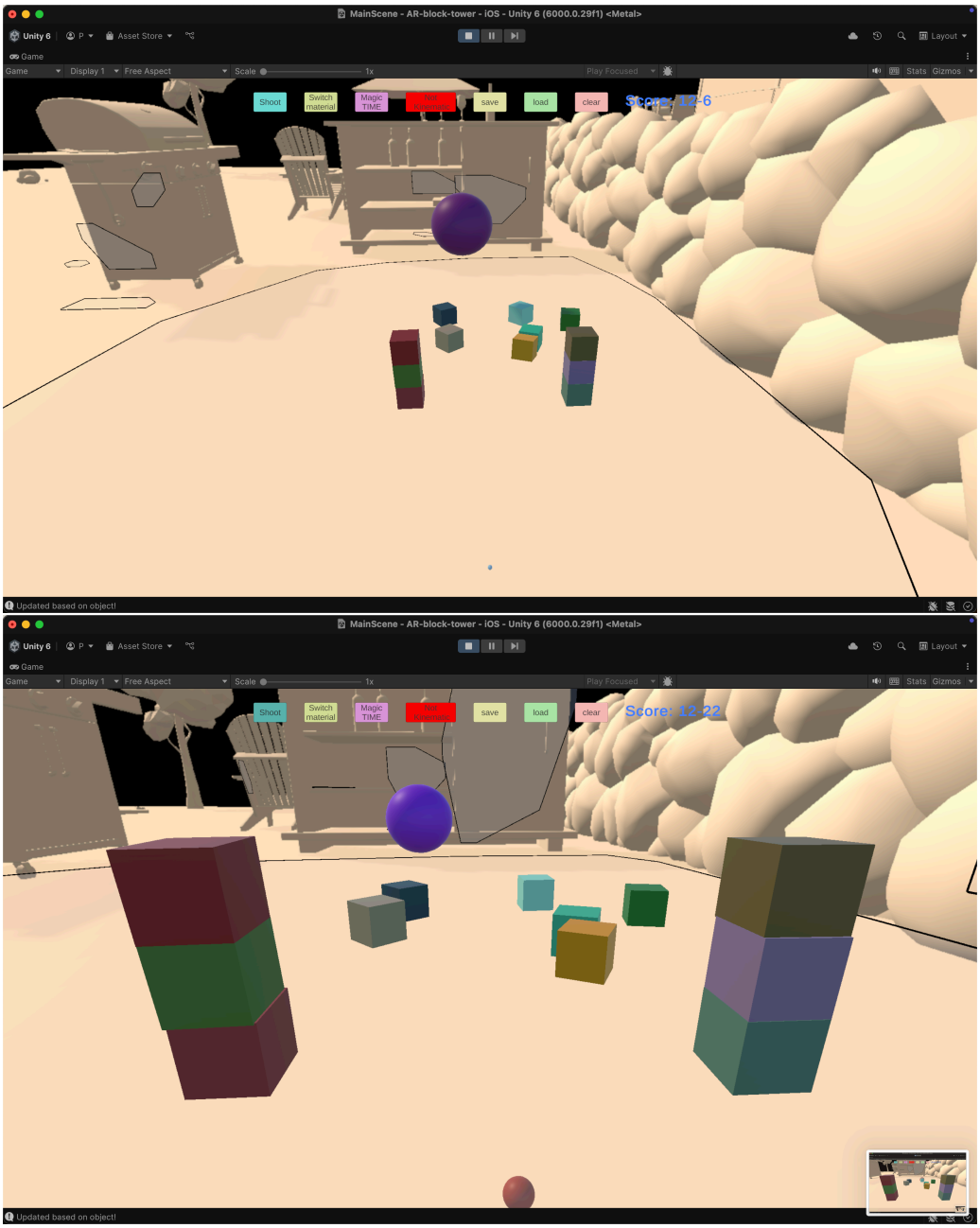

Figure 2.2: Demo of the AR block tower with bowling mode in AR Simulator.

§ 2.4 Report Structure

The report starts with an abstract that summarizes the project. Followed by the introduction, method, experiments, and conclusion chapters, with a list of references and appendices,

- the introduction chapter motivates the project and outlines the project proposal.
- the method chapter describes the features design and implementation of the project.

- the experiments chapter details the coding and testing of the project.
- the evaluation chapter analyzes the project's success and limitations.
- the conclusion summarizes the project and discusses future extensions.

3 Methodology

§ 3.1 Overview

The application is built with Unity and AR Foundation, with the below flowchart illustrates the high-level design of the AR application. The application starts with plane detection, where the user places the virtual object on the detected plane.The user can interact with the scene by clicking on buttons, saving, clearing, and loading the scene. The user can also fire spheres to knock down the blocks. The application provides user feedback through a scoring system and physics simulation. For a visual demonstration, please refer to the flowchart (Figure 3.1).

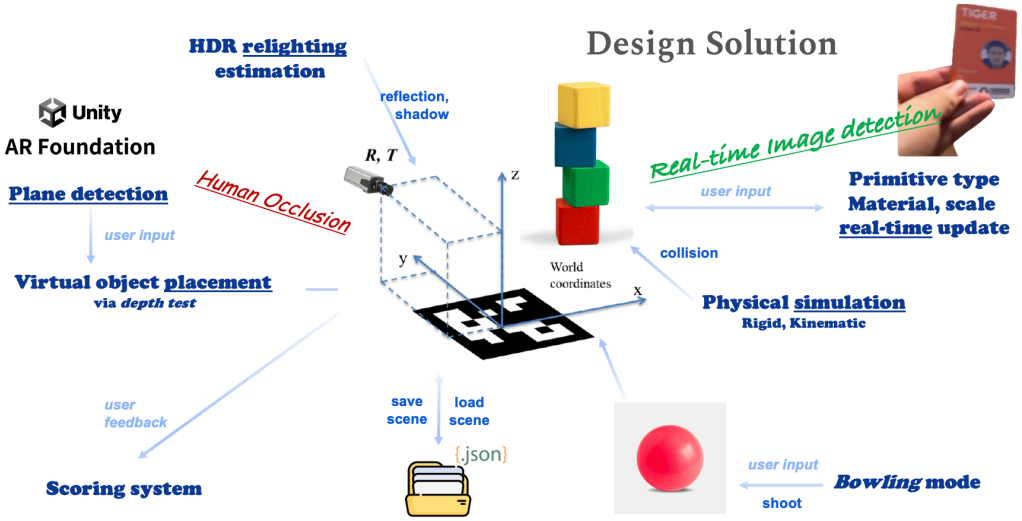

Figure 3.1: Flowchart of the AR application

In the following subsections, I will discuss the AR features and supporting features of the application in details. The AR features deployed includes plane detection, virtual object placement, and lighting estimation. The supporting features include physical simulation, graphics, UI, and audio. The interaction with the application is through user input and feedback.

§ 3.2 AR features

§ 3.2.1 Plane detection

Plane detection [KCS+17] is a feature that allows the application to detect the real-world surfaces and therefore place the virtual objects on the detected plane.

Feature based detection Scale-Invariant Feature Transform (SIFT) [Low04] recognizes local features and is invariant to image scaling and rotation, and partially invariant to affine distortion and illumination changes. Speeded-Up Robust Features (SURF) [BTVG06] is a feature detector and descriptor that is faster than SIFT by image convolutions. Features from Accelerated Segment Test (FAST) [RD06] is a corner detection algorithm that is used to detect the plane. Binary Robust Independent Elementary Features (BRIEF) [CLÖ+11] is a feature descriptor using binary strings. It is general for any feature detector. Oriented FAST and Rotated BRIEF

(ORB) [RRKB11] utilized the BRIEF descriptor and an adapted FAST feature detector. It is rotation invariant and noise-resistant.

Deep Learning detection Recent advances in deep learning have shown that deep learning models can be used to detect planes in real-time. For example, PlaneRCNN [LKG+19] is a deep learning model that can detect planes in real-time, which outperforms State-of-the-Art (STOA) methods in plane detection, segmentation, and reconstruction metrics.

§ 3.2.2 Object detection

Object detection [ZZtXW19] is a widely researched field in Computer Vision, due to its relationship with image and video understanding. Traditional methods are based on manually-selected features (SIFT), which are replaced by deep learning models in recent years (SVM, CNN, transformer, etc). The deep learning models are able to learn semantic, high-level, deeper features, which are more powerful than traditional methods. For downstream tasks, face, body, pedestrian, image detection and segmentation are developed.

§ 3.2.3 Relighting estimation

Relighting estimation [EGH21] is a feature that allows the application to estimate the lighting conditions of the real-world environment and adjust the virtual objects. It is particularly useful for different weather conditions, times of day, and indoor/outdoor scenes. From the survey, a series of prior works have been proposed to estimate the lighting conditions, including High Dynamic Range (HDR) or Low Dynamic Range (LDR). Models, either parametric or deep learning, are trained from images with ground truth lighting conditions, to estimate the hidden relationship. The features implemented by the AR Foundation's light estimation API are in Table 4.1.

§ 3.2.4 Virtual object placement

To my best knowledge, there's no free-to-use virtual object placement features available in the built-in AR Foundation. Hence, inspired by the depth test [Gro], I implemented this feature from scratch and tested it extensively. Please refer to § 4.1.2 for implementation details and experiments.

§ 3.2.5 Human occlusion

The AR human occlusion is done by per-frame images representing depth or stencil images. It is a feature for more realistic incorporation of virtual and real-world content.

The depth test [Gro] is a technique that compares the depth of the virtual object with the depth of the real-world object. If the depth of the virtual object is greater than the depth of the real-world object, the virtual object is occluded by the real-world object. Likewise, the stencil test is a technique that compares the stencil value of the virtual object with the stencil value of the real-world object.

§ 3.3 Supporting features

I implement the following supporting features to enhance the AR experience,

- Physical simulation
- Real-time virtual object update (primitive type, material and scale)
- UI (canvas, buttons, on-click events)

- Checkpoints (saving and loading all the placed objects of the scene in the `json` file)
- audio, graphics resources
- scoring system

In the following section, I will discuss the physical simulation. For details of the other features, please refer to § 4.

§ 3.3.1 Physical simulation

Physical simulation, including rigid body dynamics, collision detection, and gravity, is a feature that allows the user to experience the real-world physics in the AR environment. The rigid body dynamics [MMC+20] is a technique that simulates the motion of objects under the influence of forces, which is essential for the physical simulation. Collision detection [KHI+07] is a technique that detects the intersection of two objects, which is essential for the physical simulation. Gravity is a force that pulls objects towards the center of the Earth, which is essential for the physical simulation.

Physics equations are adopted in the physical simulation domain, including Newton's laws of motion, conservation of momentum, and conservation of energy [JS98]. Alternatively, artists design fake physics to simulate the real-world physics, which is more flexible and controllable.

§ 3.4 Interaction

§ 3.4.1 User input

To place virtual objects in the scene, the user can interact with the application by clicking on screen. The virtual object is placed on the detected plane if the hit point is not on any existing object, or stacked on the closest object with proper location update if the hit point is on the existing object. Likewise, users can fire sphere with another button to knock down the blocks. The physical simulation allows the user to blow the blocks away because of collision detection.

§ 3.4.2 User action recording

The user can save, clear, and load the scene with buttons. In the background, the scene is saved in separate `json` files, which can be loaded back to the scene. The overall assumption is that users are familiar with the button interaction in the AR application. Otherwise, a tutorial or guide can be provided to help users understand the interaction modality.

§ 3.4.3 User feedback

The scoring system is a direct feedback to the user, rewarding players for achieving goals. The user can also interact with the UI elements, e.g. the Physical simulation is a toggle button, which allows the user to visualize the current state by checking the color of the button. Physical simulation is on when the button is green, and off when the button is red.

Physical simulation itself is surprising to the user, as the user can blow the blocks away because of collision detection. The real-time virtual object update (primitive type, material and scale) is another surprise to the user, adding more spice to the experience.

4 Implementation

§ 4.1 Codebase and feature integration

I adopt the class hierarchy in Unity, which is an object-oriented programming language. The main classes include `ARRaycastPlace` is a subset of the given `MonoBehaviour` class, which is the base class from which every Unity script derives. Similarly, all different features are implemented in their classes. The class members are encapsulated by the access modifiers, such as `public`, `private`, and `protected`. The class members include fields, properties, methods, and events. When they are accessed by other classes, the access modifiers are used to control the visibility of the members.

§ 4.1.1 Plane detection

Plane detection is the first step in my AR application, where the user places the virtual object on the detected plane. The plane detection feature is implemented using AR Foundation, which provides a set of APIs for plane detection. AR plane manager is used to detect planes in the environment, and AR plane visualizer is used to visualize the detected planes. In particular, "Plane Prefab" allows different material (e.g. a dot or uniform-color texture).

§ 4.1.2 Virtual object placement

AR raycast manager encapsulates the raycasting logic for AR applications. There are existing solutions in the third-party libraries, such as AR Magic Bar [Ali]. However, they are expensive, not open-source and not customizable. Hence, I **implement the script from scratch myself** inspired by depth test [Gro].

The `C#` code snippet `Update()` is my implementation of the virtual object placement feature in the AR application, by testing the depth of the hit point, with a ray cast from the camera plane pointed by the user. The main goals are to (1) check if the hit point is on the detected plane or other virtual objects via loop, (2) stack the new object on the closest object with proper location update if the hit point is on the existing object, and (3) stack the new object on the detected plane if the hit point is not on any existing object.

For detailed implementation, please refer to Appendix B.2. I include the main pseudocode below,

```
if (Input.GetMouseButtonDown(0)){
    Ray ray = arCamera.ScreenPointToRay(Input.mousePosition);
    hits = new List<ARRaycastHit>();
    // ray cast to detected planes
    if (raycastManager.Raycast(ray, hits, TrackableType.Planes)){
        Pose hitPose = hits[0].pose;
        Vector3 stackPosition = hitPose.position;
        GameObject closestObject = null;
        // default set to the distance to the hit point on the plane
        float closestDist = Vector3.Distance(ray.origin,
```

```
                                        hitPose.position);

        // Find the closest object to the camera along the ray
        foreach (GameObject obj in placedObjects){
            Vector3 objScreenPos = arCamera.WorldToScreenPoint(
                                        obj.transform.position);
            // Assert: the object is in front of the camera
            // with correct direction; hitResult = True, i.e.,
            // the ray hits the object. Otherwise, continue.
            float distance = Vector3.Distance(ray.origin,
                                        obj.transform.position);
            Debug.Log("Distance to object: " + distance);
            if (distance < closestDist)
            {
                // Update the closest distance and object
            }
        }
        if (closestObject != null){
            // Stack on the closest object
            closestObjRend = closestObject.GetComponent<Renderer>();
            float objectHeight = closestObjRend.bounds.size.y;
            stackPosition.y += objectHeight;
            // update position and rotation
        }
        else{ // Stack on the plane
        }
        GameObject newObject = Instantiate(objectToPlace,
                                        stackPosition,
                                        hitPose.rotation);
        placedObjects.Add(newObject);
    }
}
```

The placed virtual objects and fired spheres in the bowling mode are store by different lists of Game Object `List<GameObject>`. By tracking them in the `List` data structure, I can easily manipulate them, such as updating, clearing, saving, and loading the scene.

§ 4.1.3 Human Occlusion

In Unity, I apply the `AROcclusion Manager` component to the main camera with the `ARCamera Background` to automatically enable the background rendering pass to incorporate any available depth information when rendering the depth buffer. I add the Human depth and stencil, besides environment occlusion, which the speed is preferred against the quality for game experience. For results, please refer to Figure 5.1.

§ 4.1.4 Object detection

In Unity, it's hard to implement hand detection, due to Apple lack of software compatibility. Hence, I utilize the AR tracked image manager instead. To track the target images, multiple reference images are added to the library. The AR tracked image manager is used to detect the images in the camera perspective.

The tracked images are updated in real-time, where a virtual object is updated right above the image. In addition, a separate class, subscribing to the `ARTracked ImageManager`'s event, will be notified whenever an image is added (that is, first detected), updated, or removed. When the image exists, i.e., `ImagesChanged` is triggered, a virtual object magic-time function is called for enhancing the user experience. Please refer to Figure 5.2 for the object detection feature results.

§ 4.1.5 Relighting estimation

Relighting estimation is particularly useful for different weather conditions, times of day, and indoor/outdoor scenes. Unity Light Estimation Mode is provided by the AR Camera Manager script, including `Ambient Intensity`, `Ambient Color`, `Ambient Spherical Harmonics`, `Main Light Direction`, `Main Light Intensity` (Table 4.1).

Table 4.1: Relighting estimation features.

Features	Purpose
Ambient Intensity	ambient brightness of light.
Ambient Color	ambient light RGB value.
Ambient Spherical Harmonics	multi-directional ambient light estimation.
Main Light Direction	direction of the dominant light source.
Main Light Intensity	brightness of the dominant light source.

It is used to adjust the virtual object's lighting to match from different real-world lighting representations. I choose the former three as the mode used in this project, where an alternative choice is `Everything`.

§ 4.1.6 Physical simulation

The physical state of the objects in the scene is controlled by the rigid body component in Unity. The rigid body component is used to simulate the physics of the object, such as gravity, mass, and velocity. The rigid body component is also used to detect collisions between objects in the scene. I implement a toggle for whether kinematic state of the rigid body is enabled.

```
public void ToggleKinematicState()
{
    targetRb.isKinematic = !targetRb.isKinematic;
    // debug log to show the state of the rigidbody
    Debug.Log("Kinematic state: " + targetRb.isKinematic);

    // change button color and text
```

```
    if(targetRb.isKinematic)
    {
        GetComponent<UnityEngine.UI.Button>().image.color = Color.green;
        buttonText.text = "Kinematic";
    }
    else
    {
        GetComponent<UnityEngine.UI.Button>().image.color = Color.red;
        buttonText.text = "Not\n Kinematic";
    }
}
```

§ 4.1.7 Other features

For **checkpoints**, `json` files are appended to the persistent data path, which stores the current game state, including the number of placed objects and fired spheres by the format of,

```
{
    objectName: " ",
    position.x: " ", position.y: " ", position.z: " ",
    rotation.x: " ", rotation.y: " ", rotation.z: " ", rotation.w: " ",
    scale.x: " ", scale.y: " ", scale.z: " "
}
```

where the `objectName` is the name of the object, and the `position`, `rotation` (quaternion), and `scale` are the transform properties of the object. The `json` files are loaded and saved by the script.

For **User Interface (UI)**, I use the Unity built-in UI system, including canvas, buttons, and `on-click()` events. The UI system is used to display the game state, interactive events, and scoring system (Figure 4.1). To access the main class functions, I use the `private` access modifier to control the visibility of them properly in the UI script.

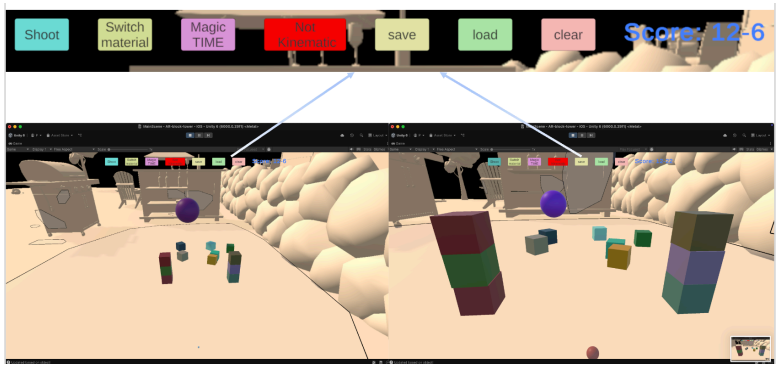

Figure 4.1: UI of the AR application

For example, the `clear` button is implemented by `on-click()` ⇒ `clearObjects()`,

```
private void clearObjLists( List<GameObject> objList){
        // clear all the objects
    foreach (GameObject obj in objList)
    {
        Destroy(obj);
    }
    objList.Clear();
}

public void clearObjects(){
    // clear all the objects
    clearObjLists(placedObjects);
    clearObjLists(firedSpheres);
    Debug.Log("Clearing objects done!");
}
```

For **scoring** system, it is based on the number of placed objects and fired spheres stored in the `ARRaycastPlace` class, please refer to Appendix B.1 for detailed implementation.
For **audio**, I use the Unity built-in audio source, which are loaded from the `Resources` folder, to play the background music and sound effects.

§ 4.2 Meshes and assets

Unity is under the MIT license, which is free to use for personal and commercial projects. Unity provides a wide range of built-in assets, including meshes, materials, and textures.
For **meshes**, I adopted various primitive types of Unity: cube, sphere, cylinder, plane, etc. They form the basic building blocks of tower or fired spheres.
For **assets**, I used the Unity Asset Store, providing a wide range of free and paid assets. In particular, the AR setup is based on the AR Foundation package, which contains the default assets for ARCore and ARKit plugins on Android and iOS devices.
For **materials**, I use a combination of Unity's built-in materials, including Lit, Unlit, and Standard materials for various purposes. I also created custom materials for the plane detection feature, which uses a white transparent material.

§ 4.3 Testing strategies

The main testing strategies include unit testing, integration testing, and user testing. In particular, unit test involves function-level testing, integration test involves module-level testing, and user test involves end-to-end testing on the target device.
Testing by assertion or logging (printing and debugging software), such as `Debug.Log()` and breakpoints, is the most common way to test the code. For example, I use `Debug.Log()` to print the depth of the hit point on the plane (Figure 4.2), the distance between the camera and the object, and the state of the rigid body. These help me to narrow down the potential bugs and fix them.

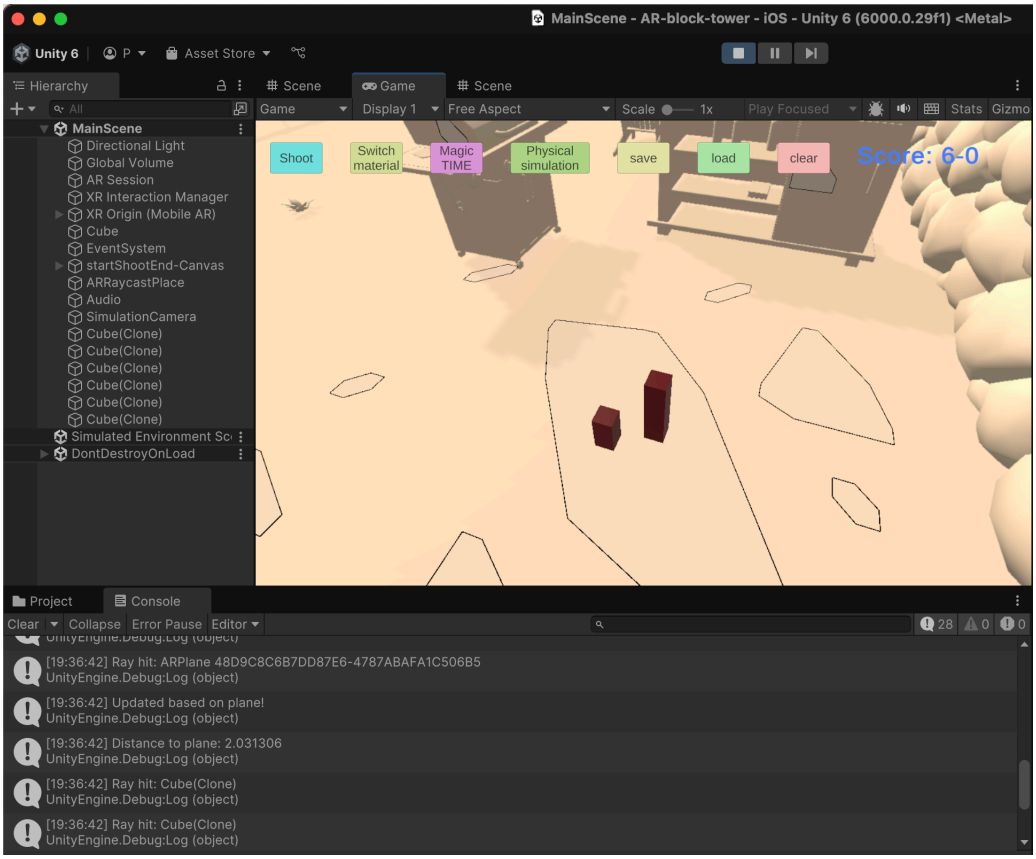

Figure 4.2: Logging my virtual object placement feature in Unity.

§ 4.4 Milestones

In the chronological order, the project progressed as summarized in the Table 4.2.

Table 4.2: Milestones and testing strategies

week	milestone	testing
1	Writing proposal	N/A
2	Learning AR	N/A
3	Plane detection	unit testing
4	Object placement	logging, assertion
5	Physical simulation	unit testing
6	Other features	user testing
7	Improvements	all the above tests
8	Demo and report	N/A

For milestone details, please refer to the previous chapters. The logging or assertion are included in the code snippet in the previous sections and appendices. The user testing is conducted by myself and my friends, who are not familiar with AR applications. The feedback is collected and used to improve the application design.

5 Critical evaluation

§ 5.1 Success analysis

For this project, I am most proud of integrating various AR techniques into the AR block-stacking game, making it accessible and inclusive and providing an engaging and joyful AR gameplay experience, including:

- the virtual object placements allow the user to stack the new object on the closest object with proper location update if the hit point is on the existing object, or stack the new object on the detected plane if the hit point is not on any existing object.
- the relighting estimation gives a huge contrast to the virtual objects, making them more realistic according to the real-world lighting condition.
- the physical simulation allows the user to blow the blocks away because of collision detection.
- the human occlusion, where the virtual objects are occluded by the real-world human or objects, making the virtual objects more realistic.
- the real-time virtual object update (primitive type, material and scale) is a surprise to the user, adding more spice to the experience.
- the scoring system is a plus to user experience, rewarding players for achieving goals.
- the analysis of techniques and underlying theories.

From the technical perspective, my virtual object placement feature works particularly well. The feature verifies the depth of the hit point with a ray cast from the camera plane pointed by the user, which is extensively asserted and tested. It achieves a similar result as the real-world object stacking, which is the main goal of the feature.
Human occlusion is an important feature for AR project, which seemly works well in the project. For instance, I capture the real-time results in Figure 5.1, where the virtual objects are occluded by the real-world human or objects, making the virtual objects more realistic.

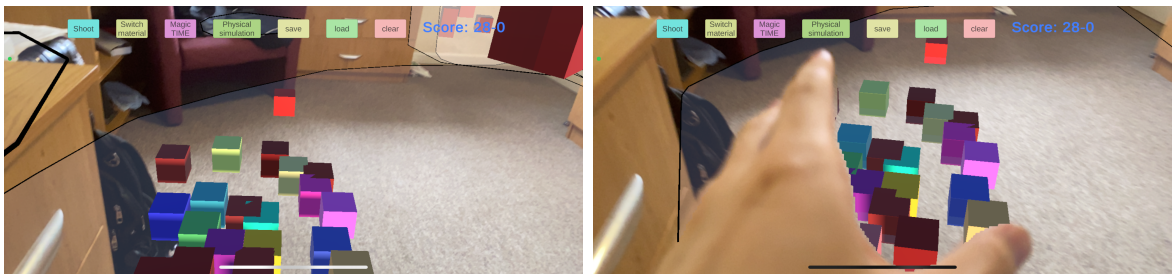

Figure 5.1: Human occlusion.

Despite the challenge of hand tracking described in the next section, I use an alternative approach via the image detection. It works well in the project, where the user can the target image to control the scene without touching the screen. It's beneficial for smoother and pleasant interactions. The results are shown in Figure 5.2, where users can control the cube's position

and rotation by moving the target image. In addition, the virtual objects information keeps updating in real-time via a default function call, which is a plus to the user experience.

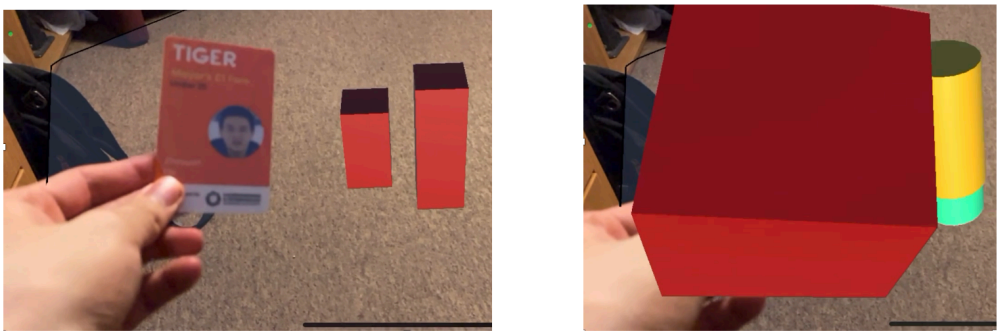

Figure 5.2: Before and after image detection.

§ 5.2 Failure analysis

The most challenging part of the project is the integration of the AR techniques. The AR techniques are non-trivial and require a proper understanding of the AR technology. For example, the relighting estimation is a complex technique that requires a good understanding of the real-world lighting condition and the virtual object's material properties. For **shadowing** on the detected transparent plane, it is difficult to achieve the desired effect. Alternative solution would be to use a different material for the transparent plane, or turn transparent plane into a non-transparent plane.

Hand tracking is directly harder than expected, due to Apple ARKit lack of support for hand tracking. Hence, I utilize an alternative approach via the image detection. In the future, I plan to use an alternative platform or API that supports hand tracking, or use a different technique to detect and track the hand.

§ 5.3 Deviation from project proposal

The scoring system is different from original proposal. It is because of the newly added bowling mode, which requires a new scoring system to reward players for achieving goals. However, such changes are beneficial to the project, making the game more goal-oriented and engaging. Other than that, the project is consistent with the original proposal.

In this project, I also explore **beyond** the original proposal with more exciting features. HDR lighting estimation is used to make the virtual objects more realistic, according to the real-world lighting condition. I also introduce a new game mode similar with **bowling**, where different spheres are shot from the camera to knock down the blocks. A new scoring system is designed to reward players for achieving goals. The game is more goal-oriented and engaging.

6 Conclusions and future extensions

§ 6.1 Conclusion

The project is a success in creating an AR block-stacking game that is accessible and inclusive. I aims to provide an engaging and joyful AR gameplay experience. The project successfully integrates a series of non-trivial AR techniques, such as plane detection, relighting estimation, physical simulation, human occlusion, object detection and virtual object placements. By integrating such AR features, it further enhances the experience by overlaying digital content that interacts dynamically with the physical world (Figure 6.1).

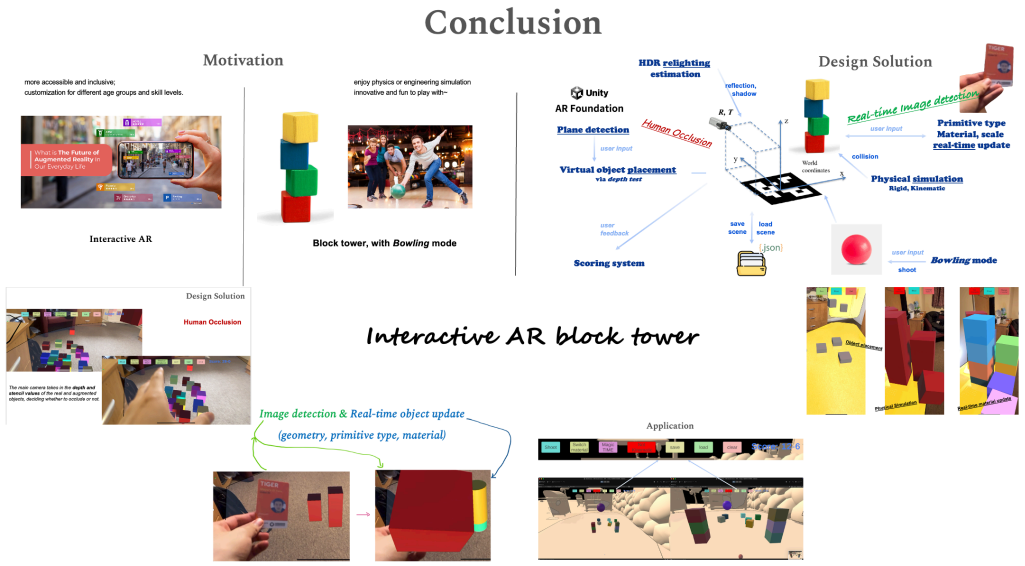

Figure 6.1: Project conclusion.

The project also focuses on how user behaviours can affect the virtual objects. For example, the user can blow the blocks away because of collision detection. Moreover, the block itself is evolving with various unexpected size, geometry, and material, which adds more spice to the experience.

Checkpoints and roll-back are crucial for long-term gameplay. I also introduce a new feature that allows the user to save the current game state and resume any one of them later. This feature is useful for users who want to take a break and continue later.

In this project, I explore beyond the original proposal with more exciting features. High Dynamic Range (HDR) lighting estimation is used to make the virtual objects more realistic, according to the real-world lighting condition. I also introduce a new game mode similar with **bowling**, where different spheres are shot from the camera to knock down the blocks. A new scoring system is designed to reward players for achieving goals. The game is more goal-oriented and engaging.

In conclusion, I have a hand-on experience on how to apply the theory of Extended Reality (XR) to create an AR game. I have learned a lot from the project, including how to design, implement, test, debug and evaluate along the whole process. I am delighted with the outcome

of the project, and I believe the project has achieved its goal of providing an accessible and inclusive AR gameplay experience as proposed.

§ 6.2 Future extensions

Hand tracking Due to Apple Kit lack of support for hand tracking, I am not able to integrate the hand tracking feature in the project, unfortunately. In the next step, it's crucial to implement hand tracking to allow users to interact with the virtual objects or UI elements more naturally.

3D reconstruction With the recent advent of machine learning and computer vision, it is possible to reconstruct the 3D model of the physical world. In the future, I would like to integrate the 3D reconstruction feature into the project. This feature will allow users to scan the physical world and import the 3D model into the virtual world. The user can then interact with the 3D model in the virtual world, e.g. placing virtual objects on the 3D model, or even playing games on the 3D model.

AR Multiplayer mode The current project is designed for single-player mode. In the future, I would like to extend the project to support multiplayer mode. This feature will allow users to play with friends or family members in the same or even different physical space(s).

Story mode with AR features The current project is more like a sandbox game. In the future, I would like to add a story mode to the game, with AR storytelling characters and plot. In addition, there will be a series of levels with different challenges and goals, including puzzles, mazes, and boss fights scattered throughout the virtual world.

Bibliography

[Ali] AliveStudios. Ar magic bar (ar foundation, lightship & 3d). Unity Asset Store package, accessed on February 12, 2025. URL: https://assetstore.unity.com/packages/tools/utilities/ar-magic-bar-ar-foundation-lightship-3d-284853?srsltid=AfmBOoqzi9OQefNyDx2G4_HF68EhHip6XicQo4dkCQ8PauH65vm5uVhw. 8

[BTVG06] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. Surf: Speeded up robust features. In Aleš Leonardis, Horst Bischof, and Axel Pinz, editors, *Computer Vision – ECCV 2006*, pages 404–417, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. 5

[CLÖ+11] Michael Calonder, Vincent Lepetit, Mustafa Özuysal, Tomasz Trzcinski, Christoph Strecha, and Pascal Fua. Brief: Computing a local binary descriptor very fast, 2011. URL: https://infoscience.epfl.ch/handle/20.500.14299/69740, doi:10.1109/TPAMI.2011.222. 5

[EGH21] Farshad Einabadi, Jean-Yves Guillemaut, and Adrian Hilton. Deep neural models for illumination estimation and relighting: A survey. *Computer Graphics Forum*, 40(4), June 2021. JEL classification: Computer graphics → Neural networks. Accessed on February 12, 2025. doi:10.1111/cgf.14283. 6

[Gro] Khronos Group. Depth test - opengl wiki. Accessed on February 12, 2025. URL: https://www.khronos.org/opengl/wiki/Depth_Test. 6, 8

[JS98] Jorge Valenzuela José and Eugene Jerome Saletan. *Classical Dynamics: A Contemporary Approach*. Cambridge University Press, Cambridge UK, 1998. 7

[KCS+17] Dongchul Kim, Seungho Chae, Jonghoon Seo, Yoonsik Yang, and Tack-Don Han. Realtime plane detection for projection augmented reality in an unknown environment. In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5985–5989, 2017. doi:10.1109/ICASSP.2017.7953305. 5

[KHI+07] S. Kockara, T. Halic, K. Iqbal, C. Bayrak, and Richard Rowe. Collision detection: A survey. In *2007 IEEE International Conference on Systems, Man and Cybernetics*, pages 4046–4051, 2007. doi:10.1109/ICSMC.2007.4414258. 7

[LKG+19] Chen Liu, Kihwan Kim, Jinwei Gu, Yasutaka Furukawa, and Jan Kautz. Planercnn: 3d plane detection and reconstruction from a single image. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4445–4454, 2019. doi:10.1109/CVPR.2019.00458. 6

[Low04] David G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2):91–110, 2004. doi:10.1023/B:VISI.0000029664.99615.94. 5

[MMC+20] Matthias Müller, Miles Macklin, Nuttapong Chentanez, Stefan Jeschke, and Tae-Yong Kim. Detailed rigid body simulation with extended position based dynamics. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '20, Goslar, DEU, 2020. Eurographics Association. doi:10.1111/cgf.14105. 7

[RD06] Edward Rosten and Tom Drummond. Machine learning for high speed corner detection. In *Proceedings of the 9th European Conference on Computer Vision (ECCV)*, volume 1, pages 430–443, 2006. 5

[RRKB11] Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary Bradski. Orb: An efficient alternative to sift or surf. In *2011 International Conference on Computer Vision*, pages 2564–2571, 2011. doi:10.1109/ICCV.2011.6126544. 6

[ZZtXW19] Zhong-Qiu Zhao, Peng Zheng, Shou tao Xu, and Xindong Wu. Object detection with deep learning: A review, 2019. URL: https://arxiv.org/abs/1807.05511, arXiv:1807.05511. 6

A Workload Distribution

Not Applicable (N/A), due to the nature of single person project.

B Implementation details

§ B.1 Scoring system

The below C# code snippet is my implementation of the scoring system in the AR application. The main goal is to update the scoring based on the number of placed objects and fired spheres stored in the ARRaycastPlace class.

```
using UnityEngine;
using TMPro;
public class ScoreTMP : MonoBehaviour
{
    private ARRaycastPlace arRaycastPlace;
    public TextMeshProUGUI placedObjectsText;
    void Start()
    {
        arRaycastPlace = FindFirstObjectByType<ARRaycastPlace>();
    }
    void Update()
    {
        // update text to show the number of placed objects and
        // fired spheres from ARRaycastPlace class
        if (arRaycastPlace != null)
        {
            placedObjectsText.text = "Score: " +
             arRaycastPlace.ShowPlacedObjectsNum() +
            "-" + arRaycastPlace.ShowFiredSpheresNum();
        }
    }
}
```

§ B.2 Virtual object placement

The C# code snippet `Update()` is my implementation of the virtual object placement feature in the AR application, by testing the depth of the hit point, with a ray cast from the camera plane pointed by the user. The main goals are to (1) check if the hit point is on the detected plane or other virtual objects via loop, (2) stack the new object on the closest object with proper location update if the hit point is on the existing object, and (3) stack the new object on the detected plane if the hit point is not on any existing object.

```
// ARRaycastPlace.cs
void Update()
{

    if (Input.GetMouseButtonDown(0))
    // and check not at UI button
    {
        Ray ray = arCamera.ScreenPointToRay(Input.mousePosition);
        hits = new List<ARRaycastHit>();
        if (raycastManager.Raycast(ray, hits, TrackableType.Planes))
        {
            Pose hitPose = hits[0].pose;
            Vector3 stackPosition = hitPose.position;

            GameObject closestObject = null;

            // default set to the distance to the hit point on the plane
            float closestDist = Vector3.Distance(ray.origin,
                                                 hitPose.position);
            Debug.Log("Distance to plane: " + closestDist);


            // Find the closest object to the camera along the ray
            foreach (GameObject obj in placedObjects)
            {
                Vector3 objScreenPos = arCamera.WorldToScreenPoint(
                                              obj.transform.position);

                // Check if object is in front of the camera
                if(objScreenPos.z <= 0)
                {
                    Debug.Log("Object is behind the camera!");
                    continue;
                }
```

```
            Ray objRay = arCamera.ScreenPointToRay(objScreenPos);

            // Check if the object is in the same general direction
            if (Vector3.Dot(ray.direction, objRay.direction) <= 0)
            {
                Debug.Log("Object is not in the same direction!");
                continue;
            }

            RaycastHit hit2;
            bool hitResult = Physics.Raycast(ray, out hit2,
                                             Mathf.Infinity);

            if (!hitResult)
            {
                Debug.Log("Ray does not hit anything!");
                continue;
            }

            if(hit2.collider.gameObject != obj){
                Debug.Log("Ray hit: " + hit2.collider.gameObject.name);
                continue;
            }

            float distance = Vector3.Distance(ray.origin,
                                        obj.transform.position);
            Debug.Log("Distance to object: " + distance);

            if (distance < closestDist)
            {
                // Update the closest distance and object
                closestDist = distance;
                closestObject = obj;
            }
        }

        if (closestObject != null)
        {
            // Stack on the closest object
            Debug.Log("Updated based on object!");
```

```
                closestObjRend = closestObject.GetComponent<Renderer>();
                float objectHeight = closestObjRend.bounds.size.y;
                stackPosition = closestObject.transform.position;
                stackPosition.y += objectHeight;
                hitPose.rotation = closestObject.transform.rotation;
            }
            else{
                // Stack on the plane
                Debug.Log("Updated based on plane!");
                stackPosition.y += 0.08f;
            }
            GameObject newObject = Instantiate(objectToPlace,
                                               stackPosition,
                                               hitPose.rotation);
            placedObjects.Add(newObject);
        }
    }
}
```