EyeSimVR

VR Maintenance Manual

EyeSim 1.1.1

Alex Arnold
2nd November 2018

Table of Contents

My Contributions

During the last two semesters, I contributed to the following sections of the project:

1. **Team management**
   With teammate Michael Finn taking the lead in feature development for SubSim integration, I focused my efforts on making sure the rest of the team had work to do, were up to speed on what was happening, and were motivated and contributing. I also made sure that I was one of the main points of contact with the client, alongside Michael.
2. **Submarine Physics**
   I implemented water buoyancy and drag physics for the submarines and various objects by coding a mesh-based water physics calculation. By researching Unity engine capabilities and API functions, I was able to produce realistic water physics with little impact on performance.
3. **Scripted Builds**
   With several separate repositories depended upon by the EyeSim simulator, and multiple deployment platforms, I developed and maintained build scripts to ensure the process was accurate and rapid. I also researched automated build tools to run these scripts (such as Jenkins) but was limited by cost and setup time.
4. **Testing/Validation**
   Most of the testing of such a large, UI-heavy system such as EyeSim had to be done manually, and I often helped teammates Michael Finn and Nicholas Johnson validate their features during pull requests and even during weekly meetings.
5. **Oculus Go VR**
   I worked closely with teammate Michael Finn to produce the Oculus Go version of EyeSim for the UWA Open Day, completing the task in under a week. While Michael produced the core of the minigames, I implemented the Main Menu and created the controller systems for each game.
6. **RoBIOS Python**
   I worked with teammate Nicholas Johnson to produce the Python wrapper of the RoBIOS C API, and then closely with teammates Alex Zhong and Jayesh Joshi to produce the Python versions of the C examples in the *eyesimX* repository.

1. Introduction

This document covers the design and implementation of the VR versions of EyeSim. There are currently two VR platforms supported by EyeSim – Oculus Go and HTC Vive. Both platforms differ considerably in price and functionality, so two separate versions of EyeSim were created to take advantage of the capabilities of each platform.

This document is intended for developers new to the project to gain an understanding of the intentions behind each platform, and to understand where features are implemented within each platform/scene.

2. Oculus Go

The Oculus Go is a wireless, Android-based, self-contained VR headset based on the Gear VR platform. It comes with a controller and features 3 degrees of freedom. It’s convenience and low cost compared to other mobile-based VR headsets (such as Gear VR, Google Daydream VR and Google Cardboard) make it a viable platform for demonstrating the VR capabilities of EyeSim to a wider audience.

2.1 Overview

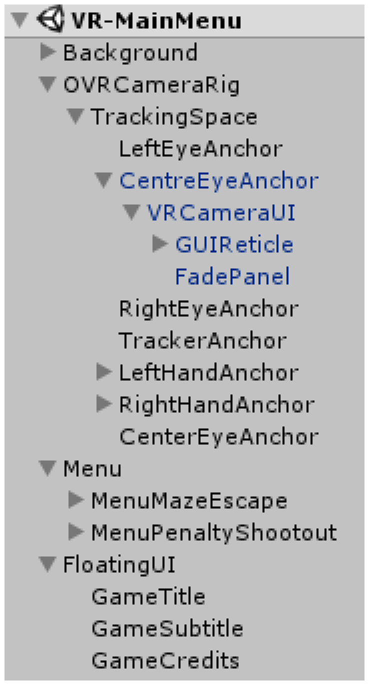
Due to the limited compute power compared to PC-based VR platforms, and the inability to run C and Python scripts within the locked-down Android environment, the Oculus Go version of EyeSim contains a series of minigames built with EyeSim components to showcase the capabilities of the simulator. It is designed to quickly show users what is available in EyeSim and what can be done with the system in VR. To this end, a custom application was created, with a menu/loading room which leads to two different minigames (Penalty Shootout and Maze Escape).

2.2 Architecture

The Oculus Go version of EyeSim utilises the OVR (Oculus VR) libraries from the Unity Asset Store, as well as Unity’s own VR Samples asset pack. These provide simple components such as the camera model, visual representation of the controllers, and various VR-related scripts, all of which are contained in the *Oculus* and *VRSampleScenes* asset folders. This allowed rapid development of the VR system in time for the UWA Open Day of semester 2, 2018.

2.2.1 Main Menu

The main menu for Oculus Go is located in the *VR-MainMenu* scene. The hierarchy of the scene is shown below in Figure 1:



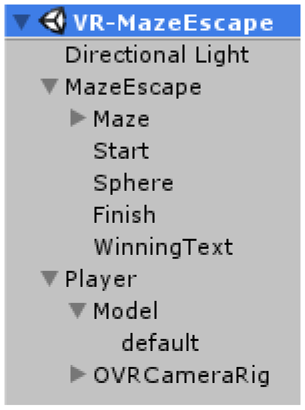
```
▼ VR-MainMenu
  ▶ Background
  ▼ OVRCameraRig
    ▼ TrackingSpace
        LeftEyeAnchor
      ▼ CentreEyeAnchor
        ▼ VRCameraUI
          ▶ GUIReticle
            FadePanel
        RightEyeAnchor
        TrackerAnchor
      ▶ LeftHandAnchor
      ▶ RightHandAnchor
        CenterEyeAnchor
  ▼ Menu
    ▶ MenuMazeEscape
    ▶ MenuPenaltyShootout
  ▼ FloatingUI
      GameTitle
      GameSubtitle
      GameCredits
```

Figure 1: VR-MainMenu scene layout

The Background and OVRCameraRig objects are prefabs from the *VRSampleScenes* and *Oculus* asset packages respectively, with the latter containing the logic and models for the Oculus Go headset. However, the OVRCameraRig prefab was extended with the VRCameraUI element from *VRSampleScenes*, which allows existing UI components from that asset package to be utilised to rapidly build an interactive UI from existing prefabs and scripts. These scripts were attached to the MenuMazeEscape and MenuPenaltyShootout components under the Menu object, allowing impressive visual effects such as the menu items "popping out" when gazed at, and the menu scene fading in and out when loaded (or when entering a minigame).

2.2.2 Maze Escape

The Maze Escape minigame is located in the *VR-MazeEscape* scene. The hierarchy of the scene is shown below in Figure 2:



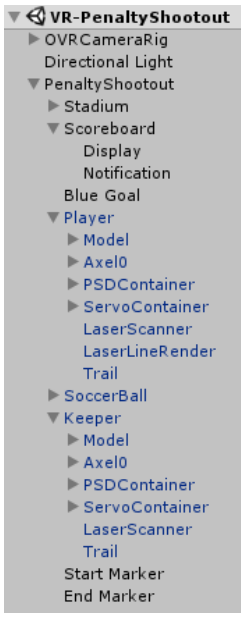
```
▼ VR-MazeEscape
    Directional Light
  ▼ MazeEscape
    ▶ Maze
      Start
      Sphere
      Finish
      WinningText
  ▼ Player
    ▼ Model
        default
    ▶ OVRCameraRig
```

Figure 2: VR-MazeEscape scene layout

The MazeEscape object simply contains the MazeEscape script, which runs the simple logic for the game – display the winning text and return to the main menu if the Player collides with the Finish object. Additionally, the MazeEscape object contains a reference to the Player object, which contains a Labbot prefab (the object labelled as *default* in Figure 2) attached to the rotation of the Oculus Go headset (OVRCameraRig). The Sphere object is used to direct the players towards the goal and matches the colour of the Finish object, while the Maze object simply contains all the walls for the maze.

2.2.3 Penalty Shootout

The Penalty Shootout minigame is located in the *VR-PenaltyShootout* scene. The hierarchy of the scene is shown below in Figure 3:



```
▼ VR-PenaltyShootout
  ▶ OVRCameraRig
    Directional Light
  ▼ PenaltyShootout
    ▶ Stadium
    ▼ Scoreboard
        Display
        Notification
      Blue Goal
    ▼ Player
      ▶ Model
      ▶ Axel0
      ▶ PSDContainer
      ▶ ServoContainer
        LaserScanner
        LaserLineRender
        Trail
    ▶ SoccerBall
    ▼ Keeper
      ▶ Model
      ▶ Axel0
      ▶ PSDContainer
      ▶ ServoContainer
        LaserScanner
        Trail
      Start Marker
      End Marker
```

Figure 3: VR-PenaltyShootout scene layout

Of note here is the PenaltyShootout object – it contains all the logic of the soccer match within a single script, also called PenaltyShootout. This detects when a Player scores (or fails to score) and updates the Display and Notification objects under Scoreboard before resetting the playing field. Additionally, the Keeper object is translated around the goal region in this script, and the end-game conditions are determined here (currently, the first to 5 goals/saves). The Player object is once again a Labbot, but with an additional script (VRRemoteControl) allowing it to be controlled by the buttons on the Oculus Go controller.

2.3 Development Environment

To develop for the Oculus Go, developers will require:

- A Windows/Mac computer (recommended Windows 10 or later, macOS High Sierra or later)
- Android Debug Bridge (ADB) installed (either by Android Studio or standalone)
- 1st generation Oculus Go headset (two available in the EECE 3.13 lab)
- Access to *braunl/RobotVR* repository (on GitHub)
- A recent version of Unity (tested on Unity 2018.2.14f)

The latest code for Oculus Go for EyeSim is located on the *vr-oculus-ui* branch of the repo to ensure no instabilities arise from the interactions of the two VR systems. Build scripts are located under *InstallerScripts/OculusGo* for both Windows and Mac, producing an Android .apk and installing it to an Oculus Go (if connected). If building manually, make sure to include all scenes mentioned above in the build (with *VR-MainMenu* ordered first).

2.4 Extension

A third minigame was partially created for the Oculus Go called Ocean Explorer, where you would navigate an underwater scene avoiding obstacles and sea creatures to find a treasure chest. However, due to the additional assets required to complete that scene, the limited time for the initial build, and the constraints on resources in the weeks following the initial build, efforts were instead focused on improving the other two minigames. Future teams may be able to get this minigame to a releasable state if requested by the client.

3. HTC Vive

The HTC Vive is a full 6-degrees-of-freedom virtual reality headset with room-scale tracking. The VR environment of the Vive is computed on a PC and sent to the headset (via cables or an upcoming wireless adapter). With these features, a more in-depth virtual reality experience can be created, with features far closer to the desktop version of EyeSim, when compared to the Oculus Go.

3.1 Overview

The HTC Vive version of EyeSim is bundled as part of the Windows build, as currently the client only required that it work on the dual-GPU lab machine running Windows 10. When EyeSim is opened on Windows, it determines whether an HTC Vive headset is connected. If so, the VR version of EyeSim will be run. This has the added benefit of reducing the number of builds between VR and desktop versions and allows users to write code in different versions without having to manage multiple environments/executables. To switch between versions, EyeSim must be restarted.

To implement the VR environment, the team used the SteamVR Unity asset from the Unity Asset Store to implement basic features like teleporting, the player model, and visualisation of the HTC Vive controllers. This was chosen over alternatives such as VRTK (which provides much greater out-of-the-box support) due to the high-quality visuals and greater community and industry support.

3.2 Architecture

The VR version of EyeSim is contained within a separate Unity scene (called 'vr') and is comprised of several new and existing components, as shown in Figure 4 below:

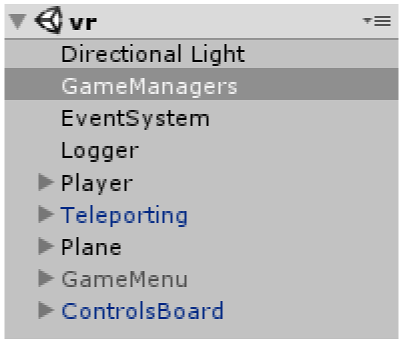


Figure 4: GameObjects in the 'vr' Unity scene for HTC Vive

3.2.1 GameManagers and VRManager

The GameManagers component mirrors the one in the desktop version of EyeSim (see: Desktop Maintenance Manual), however it is extended here to include a new manager: VRManager. The VRManager script handles the selection between VR and Desktop modes when the application starts, as well as referencing the other components in the system such as the VR-specific file paths for environment files and scripts.

The functionality for resizing the Player (affectionately known as "Antman Mode" within the team) also resides within the VRManager script. The reference to the Leg model is also located here.

3.2.2 Player and Teleporting

The Player is an existing prefab from the SteamVR asset/library, providing all the logic related to visualising the controllers, providing body and head colliders for the player, and displaying the boundaries of the play area mapped out within SteamVR before loading EyeSim.

The Teleporting prefab is also from the SteamVR asset/library and provides the logic, visualisations and sounds for teleportation within EyeSim. The sounds have not been enabled in the EyeSim version as of the creation of this document, however they remain in code for future additions to the system. This prefab was modified and extended so the teleportation feature to be used to teleport onto robots, allowing robots to be ridden.

3.2.3 GameMenu

The GameMenu is the reimplementation of the UI Menus from the desktop version so that they may be used as standalone, floating menus in VR. The GameMenu object holds many menu windows, including a template object, as shown below in Figure 5:

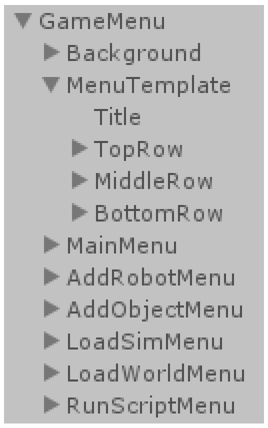


Figure 5: The GameMenu object expanded to show the child objects. The MenuTemplate object has been expanded to show the menu components.

The LoadSimMenu, LoadWorldMenu and RunScriptMenu objects differ from the other objects in that they have an additional VRFileFinder script attached to them, which allows them to parse .sim, .world or compiled C/Python scripts. The VRFileFinder script contains code similar to the file parsing code in the desktop version of EyeSim, and controls how the application interacts with simulation files while in VR mode.

3.2.4 ControlsBoard

The ControlsBoard object is a simple model that displays the Vive controls of the simulation to users. The back of the ControlsBoard object contains a small credits list, so future developers may wish to extend this.

3.3 Development Environment

To develop for the HTC Vive, developers will require:

- A quad-core computer with a Nvidia GTX 1060 GPU (or equivalent)
- A HTC Vive VR headset (including sensors)
- A recent version of Unity (last tested on Unity 2018.2.14f for Windows)
- Access to the *braunl/RobotVR* repository (on GitHub)

The build process for the HTC Vive version of EyeSim is the same as the desktop version – see the Desktop Maintenance Manual for more details. The system currently only works for Windows, and both the *main* and *vr* scenes need to be included in the build.