GENG5512 MPE Engineering Research Project Part 2

Virtual Reality for Mobile Robots

Jairus Wong Yong Zhi

21976567

School of Computer Science and Software Engineering,

University of Western Australia

Supervisor: Thomas Bräunl

School of Electrical, Electronic and Computer Engineering,

University of Western Australia

Word Count: 7798

School of Engineering

University of Western Australia

Submitted: October 16, 2022

Abstract

This project's main goal was to extend a virtual reality robot simulation program, EyesimVR. Eyesim itself is a computer-based robot simulation program, producing realistic simulations focusing on mobile robots. As a natural progression of Eyesim, a VR version, EyesimVR, was made for the Oculus family of VR headsets.

The aim of EyesimVR was to mimic the functionality of the desktop program, whilst utilising new features enabled by VR to allow for new modes of interaction with the robots within the simulation.

However, when porting the program to VR, some pre-existing simulation features from the desktop program had to be removed or changed, and many other VR-based functionality have yet to be implemented.

As such, this project aims to provide upgrades to EyesimVR, which can be roughly split into 3 sections (with some overlap). Large system changes, which cover how the simulation program is run/customised by the user, VR specific changes, which cover changes relating to functionality specifically achievable through the use of VR, and small fixes, small changes which lesser impact the program.

A main milestone for the project was completing a large system change, which was the change in the programs simulation file intake and build process. The desktop Eyesim had an important feature, the ability to use custom scripts to control robots within the simulation. EyesimVR did not have this feature, forcing all robots to follow a small selection of pre-made programs. Although implementing the exact functionality from the desktop Eyesim is ideal, it was proving difficult, thus an alternative solution was implemented. This resulted in a few compromises, such as EyesimVR being built as multiple programs (each containing one simulation, rather than one program taking in multiple simulations), scripts/simulations had to be input before building, could not run c binaries, but crucially, allowed users to provide custom scripts.

Additionally, many VR specific features were implemented, a few examples include grabbing and throwing objects, toggle a help mode with popups showing button functionality, enabling robots to run their specified custom script, and enabling the left-hand command window to have much more control on the robots and simulation settings.

With the upgrades, EyesimVR is now able to simulate robots running custom programs, and is better utilising the VR technology. For future work, however, there are still many more possible upgrades that can be made to the program, for example, implementing the ideal system (single program, simulation files and custom c binaries loaded after program is built), or merging it with the newest version of Eyesim.

Table of Contents

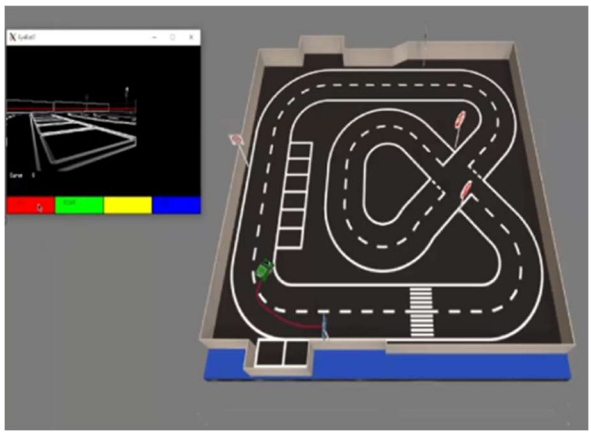
Introduction

Eyesim

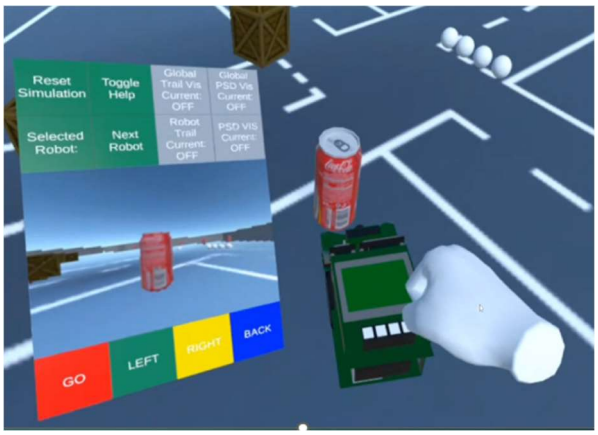
Eyesim is a robot simulation program which was designed for traditional computer systems, with a main focus on mobile robots, providing a sandbox to demonstrate driving/swimming/diving programs on robots in various terrains or environments. One major draw of such a simulation, is allowing users to create and test their own robot systems without the need to obtain and build the physical robots.

A VR version was created for the Oculus family of VR headsets, specifically the Oculus Quest 1 and 2, aiming to mimic all the functionality of the desktop program whilst utilising new features enabled by the use of VR.

Oculus Quest 1



Eyesim Desktop



Eyesim VR

Project Scope

EyesimVR attempts to mimic all the functionality of the desktop program, whilst adding in other features allowed by the use of virtual reality. However, when porting the program to VR, some pre-existing simulation features from the desktop program had to be removed or changed, and many other VR-based functionality have yet to be implemented.

As such, this project aims to re-introduce many of the missing features, provide alternative solutions where the missing features cannot be directly re-introduced, and implement many new features making use of the functionality provided by the VR environment.

Use Cases

Eyesim has generally been used for educational purposes, where students are able to test their scripts on the simulated robots, before using it on an actual robot. Whilst EyesimVR has not yet reached that capability, the features added during this project has pushed it almost to the point where it can be used for such purposes.

Additionally, the features added in EyesimVR has now allowed it to be capable of giving demonstrations of Eyesim to visitors, providing a much more fun and interactive experience compared to just watching Eyesim Desktop run on a screen.

Inputs

To generate a simulation in Eyesim, a variety of different files can be provided to the program, a brief description of each have been provided below.

A simulation file controls the environment of the simulation, controlling the robots that are put in it, the program on each robot, and the area the robot is in. This is the only required file to generate a simulation, but without referencing the other files, the robots will be static, and only pre-made objects/robots/worlds can be generated.

A robot script /program file contains the program to be run on a robot. It is usually a c file, but can also be written in python or c++. For c and c++, they first have to be compiled to produce a binary .x file, before Eyesim can use it.

World and maz files contain information about the world to be loaded. World files state some parameters of the area, including the textures to use, whilst maz files states the shape of the world (location of walls ++) with the use of symbols, such as a line “|” representing a vertical wall.

Robot files represent the robot to be added to the sim, containing links to the actual model, and other parameters of the robot, such as location and direction of sensors, and the colliders, used for physics interactions in unity.

Object files represent different objects that can be placed around the sim to be interacted with, including coke cans and soccer balls.

The Eyesim program can take any sim file as input, which should reference other robots, robot programs and objects etc, all of which will make up the simulation to be generated.

It is important to note that there is a distinct difference between robot scripts, which are compiled binaries/machine-readable instructions or the code to generate them, and all the other files, which are just text-based files which are processed by the program through file system operations. This means that the program has to handle robot scripts differently from all the other files. As such future mentions of simulation/sim files will refer to ALL the possible input to the program, except for the robot scripts, which will be referenced directly.

Literature review

Virtual Reality

Virtual reality refers to a system used for building virtual environments, usually trying to represent some aspects of reality. These systems usually allow some way to interact with the environment, through gloves, hand controllers, joysticks and head-mounted displays[10]. Within the context of this project, it refers specifically to the Oculus Quest Virtual Reality headsets, a more recent set of household entertainment systems comprising of a headset and 2 hand controllers. However, given that the VR term is technically more general, many of the older articles refer to VR systems as any system trying to mimic reality through mixing a virtual environment with other forms of interactions.

Virtual reality systems have often been used to enhance learning experiences, especially to obtain practice for equipment not easily accessible in the real world, and to allow for "safe-fails" where users can gain experience in new situations in which they would otherwise not have had [9], due to the cost of failure being too high.

As the Eyesim program is also being used to enhance students learning experiences in UWA and has proven to improve how well and fast students learn about robots [8], a natural progression of the Eyesim program was the Virtual reality implementation. A VR version of Eyesim had already been partly implemented for the Oculus family of VR headsets. It aims to leverage the VR capabilities to allow for an even wider set of interactions, allowing users to directly interact with the robots, whilst remaining realistic.

Robot Simulators

In terms of robot simulations, there has been many other similar kinds of applications made, many of which are also made for educational purposes. "Webots" is a similar program, containing many of the features present in Eyesim, such as the generation of any kind of mobile robot, the ability to input external custom scripts to control said robots, and the ability to use those custom scripts on actual physical robots.[1] However, given the program was produced almost 20 years ago, VR did not exist and no follow up extensions to the program has been made regarding VR.

Both programs have been used to further extend knowledge of robot programs. For example, Eyesim has been used to test a set of bug navigation algorithms[2], and Webot has been used to create a model for, and study robot platoons[3].

Virtual Reality Simulations

VR, on the other hand, has a range of simulations being made, but they mostly end up focusing on training motor control.

For example, the use of virtual reality was found to improve posture control in older adults, leading to a lower fall rate[4]. Similarly, a much more recent study uses technology similar to the Oculus Quest being used for this project, to determine how students perform with and without VR technology.[5] Students use 2 "pens" acting as their work tools (similar to the oculus hand controllers), to perform tasks, visualised on a computer screen. They were directly compared to students who learnt the same tasks through a normal educational video, and it was found that students with the VR equipment performed marginally better and felt more confident than those who used the video. Most of the students reported that they would rather learn using the VR equipment, showing the advantage VR has over traditional learning tools and validifying the move of eyesim to VR.

Virtual Reality Robot Simulators

However, whilst there are plenty of research going into virtual reality and robot simulations separately, there is a lot less research going into realistic robot simulations IN virtual reality. A VR system has been created to simulate a giant biped robot moving around a virtual world. However, it uses a custom set of hardware and has a bigger focus on the "fun" aspect of VR, turning the system into more of a game than a realistic simulation system.[6]

Finally, another paper was found, capturing some details on how a VR studio ported their game "Creed: Rise to Glory" into the Oculus Quest. Similar to this project, a non-VR program is to be ported to VR (both also to the Oculus Quest). However, this paper's focus is on ways to preserve a visual style, without too much performance loss (since VR systems are generally more graphically intensive) [7]. As such, this paper is not very applicable to this project unless improving the Eyesim VR performance is moved into the scope of this project, which will only happen when the many other systemic changes have been made.

From the Literature reviewed, no VR robot simulations were found, which allowed simulating realistic interactions with humans and programmed robots.

Design Process

Workflow

The work for this project was done with an agile approach, with work being completed and presented in weekly "sprints". There were weekly face-to-face meetings with the client, with each meeting showcasing the new functionality which has been implemented, followed by discussions on problems faced, any changes or alternatives solutions to be made and specific work to be completed by the next week.

Tools

Unity

Eyesim is implemented through the Unity Engine. As such, the main tool used for the project is the Unity Editor, which is an interface that helps to visualise most aspects of the project and easily allow for changes to be made. Below are some key features provided by the Unity Editor.

Being able to drag and drop scripts onto different objects allows for any scripts to be attached to different objects, making implementing some functionality a simple drag-drop operation.

All Objects in Unity exist as part of a hierarchy. The Unity editor allows the hierarchies of these objects to be easily manipulated, allowing for a much simpler implementation of many logical functions within the project.

The Unity editor allows running the project inside the editor without needing to be built. This saves time, and allows for the examining of any spawned objects which helps with debugging. As EyesimVR is meant to be run on an Oculus rather than desktop (and the input devices are also different- hand controllers vs keyboard and mouse), not everything works when running the project in the editor. However, it is able to at least create the world/robots and attach all their required scripts (even if they do not run properly given the different environment/input), which allows for those objects to be examined directly, massively helping with debugging.

The Unity Edito also allows building and running a project directly on the headset. Unity provides a function for projects “build and run”, allowing the project to be built, uploaded to the headset, and opened automatically. This saves a lot of time as the steps would normally have to be done manually, by opening a side-loader to install the APK on the device, then look for the correct application to open, both time consuming and tedious. 1 caveat is that the build and run cannot be used when any simulation files change, but this will be further explained in Future Work – Build Script.

GitHub Projects

Given the large number of changes that need to be made and tracked, a GitHub project page was created, allowing for cards, representing each issue, to be made and displayed on a different columns representing their completion state. New cards can be added, and cards can be dragged to different columns, which make tracking each requested feature/issue discovered easy. A screenshot of the project page has been included in Appendix 1.

Side Quest

After compiling a unity project, an APK file will be created for the program, which can then be installed on the Oculus. Whilst command line arguments can be used to install them, most notably through the “ADB” command, having an external program to do it can be faster and removes the need for any initial learning period of the commands and it’s arguments. As such, an external program, “Side Quest” was installed, and used primarily to install the APKs. Some screenshots to show how to use Side Quest is included in Appendix 5.

Constraints

Eyesim Desktop and Eyesim VR are supposed to share a set of files, “Eyesim library Files”, which are used for producing the actual simulation (as opposed to interacting with it, such as VR functionalities like grabbing). However, Desktop Eyesim and EyesimVR have both been upgraded in different ways, and as such, their set of Eyesim Library Files have diverged significantly. For this project, further changes to the Eyesim Library Files were avoided where possible, to prevent the difference in Eyesim library Files of the 2 version to grow even more, but there were certainly areas in which such changes had to be made.

Success Condition

Given that this project is client based, the main success condition was the client being satisfied with the number and quality of the requested features added. However, to provide a more tangible success condition, a minimum set of features were agreed upon, where the main system change (explained in the “System Changes” section below), and at least 4 VR specific features, had to be implemented. These, along with many more features, were all successfully implemented and will be described below.

Features Implemented

The features which have been implemented can generally be split into 3 different categories, with some overlap. Firstly, “Simulation System Changes”, which refer to changes which affect how the program is built or used. Secondly, “VR Features”, which refer to VR-Specific functionality and “Smaller Changes” which refer to general changes or fixes to smaller/individual parts of the program.

System Changes

Main System Change

Background

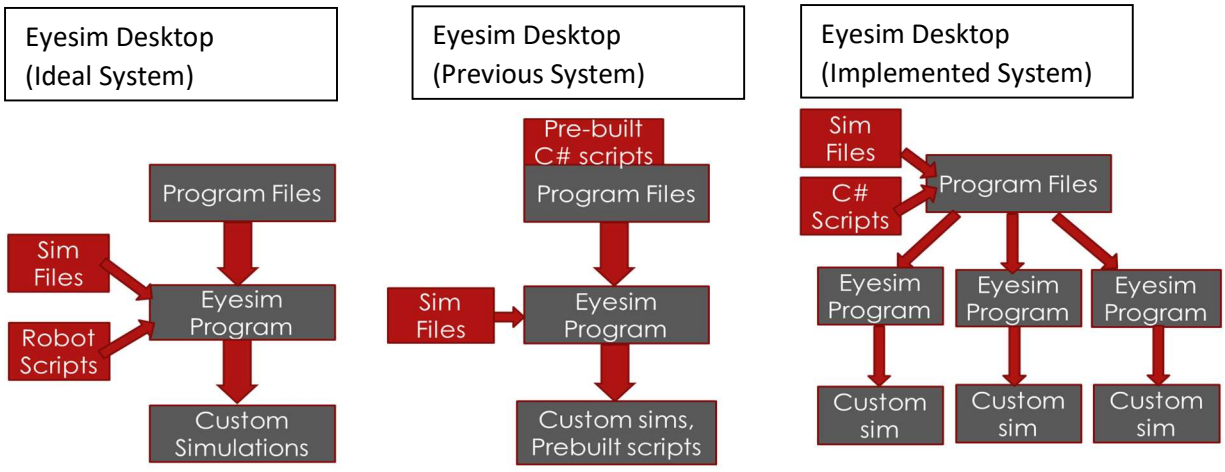
In Desktop Eyesim, the program is first built from the program files into a single executable. Users can run the executable which would open up the Eyesim program, and from that program, a set of simulation files and custom robot scripts can be selected as input into the system, which would generate the desired simulation.

This system had 4 key desirable properties. Firstly, existing as a single program, secondly, being able to take custom scripts in as input, thirdly, being able to run custom scripts of different languages, the most common 2 being c and python, and lastly, the custom scripts and simulation files were input into the system at the same time.

In EyesimVR, however due to the locked down Android environment of the Oculus, instead was implemented with a set of pre-built hard coded C# scripts which could be used to run the robots. The project files were built along with those hard coded C# scripts, to produce a single Eyesim program, but this time, only simulation files could be selected as input, and robots within the simulation could only ever run the set of hard coded C# programs. As such, EyesimVR only retained the first property, but lost the other 3, failing to be able to take in external custom scripts, run custom scripts of different languages, and introducing the simulation files into the program at a different time than the robot scripts.

As being able to input custom scripts into EyesimVR was a priority, the main system change for this project, was to convert EyesimVR into a system that could allow custom scripts as separate files, to be loaded in and control robots. Whilst there was still no solution found that could load C binaries or custom scripts during the program runtime, a solution was found that could allow custom C# scripts to be loaded in, however it had to be done before the program was built. This led to a compromise, where sim files would now be loaded in together with the custom scripts before the program is built, but each new simulation will have to be built separately into a different program, causing the program to now “exist” as multiple programs.

A figure is provided below to provide more clarity on the difference between the 3 systems regarding how the program is built, and how each simulation is generated.



Finally, a summary table showing the 3 different systems and how many desirable properties they have, is provided below to provide an easy comparison between the 3 systems.

	Program State	Run C Binaries	Scripts and Sim files Loaded Together?	Run Custom Scripts
EYESIM DESKTOP (IDEAL)	Single	Yes	Yes	Yes
EYESIMVR (PREVIOUS)	Single	No	No	No
EYESIMVR (Implemented Alternative)	Multiple	No	Yes	Yes

Subtasks

To implement this system change, there were a large set of subtasks which needed to be completed, but the three crucial ones which provided EyesimVR the 2 desirable properties from Eyesim Desktop, have been listed below and will be discussed further.

- Custom scripts -> Separate C# scripts into individual scripts / separate files
- Scripts and Simulation files loaded together -> Method of storing and retrieving simulation files from within the build
- Command line building

Implementation (Custom Scripts)

The most important feature that was added in this system change was the ability to provide custom scripts as separate files into the program. This was done through 3 features provided by C#.

Firstly, the ability to call methods by name, which allowed simulation files to name the desired method to run for each robot (as long as the corresponding robot script is named correctly). On Eyesim Desktop, running a robot involved starting an external program, which would start a server connecting with the robot in the simulation, to control it. The server controlling the robot would use an external robot script (compiled c binaries, python++) and thus the sim file would need to provide the name of that external file for the server to process. For EyesimVR instead, running a robot involved a simple method call which would have been provided as a custom script before the program was compiled. As such, “calling by name” allowed for the program to simply call that method to run a robot, and the code handling the running of robot scripts was then edited to treat the name provided by the sim file as the method name to call, rather than the name of which robot script/binary to find.

Secondly C# allows classes to be split into multiple files by turning each file into a partial class. Each robot is built off a base class called “Eyebot”, which then calls the requested method provided in the

simulation file. As calling methods by name only works on methods provided by its own class, the method names requested for in the sim file has to already exist inside the Eyebot class. In most languages, classes can only ever exist in 1 file, which would make loading in custom scripts impossible, but C# allows splitting a class into multiple partial classes which will automatically be joined together on compilation. As such, the “Eyebot” class was converted into a partial class, so custom robot scripts can be written in a separate file, ensuring that it is declared as a partial class of “Eyebot”.

Lastly, C# allows for methods to be created and used inside another method. Without these, different custom robot scripts which needed helper functions had to ensure that those helper function names were unique. The ability to create methods inside methods allows all these helper functions to be declared inside the primary method (to be called by the robot), ensuring that they do not interfere with identically named helper functions of other custom scripts.

Implementation (Scripts and Sim Files Load Together)

In Eyesim Desktop, the simulation files are first manually placed in a file system, which can then be accessed by the program to generate each simulation.

For EyesimVR instead, since the robot scripts can only be loaded in before the program is built, and the loading in of robot scripts and simulation files together is desired, the simulation files have to be provided before the program is built, so the program will already have the simulation files internally. However, most files/folders within the project are turned into a binary data when built into a program, and as such, changes had to be made on how the program accessed those simulation files.

It was found that Unity allowed a folder specifically named “streaming assets” to be placed into the project assets folder. Unity would try it’s best to retain the file system for any folders/files inside it. If the simulation files are stored there, it should be easy to just change the program to reference those simulation files.

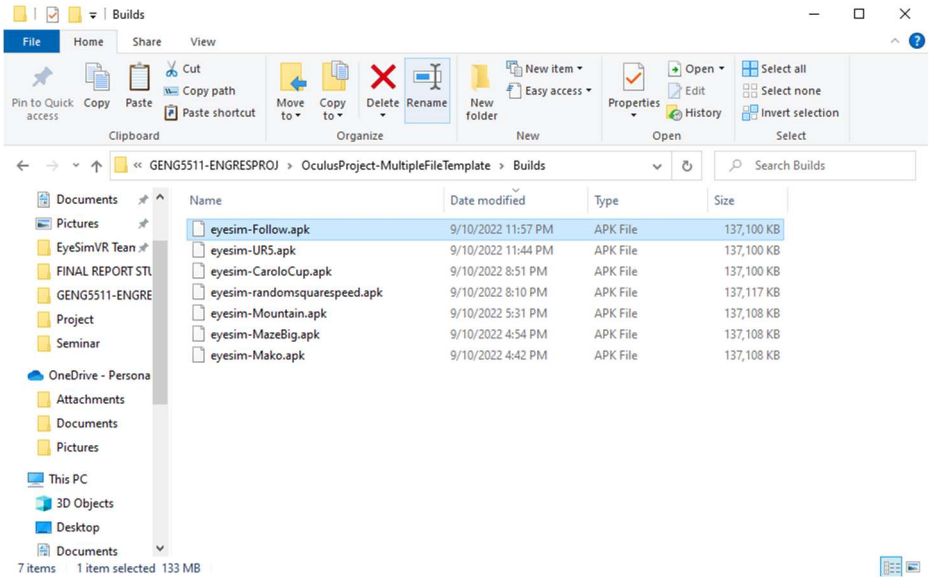
However, whilst this might work when built as a computer program, it does not work when the program is specifically built into an APK file for Android (the Oculus’ Operating System), where contents within the streaming assets have also been turned into data, losing any references to a file system. Although data from specific files can technically still be accessed by name, through using Unity Web Requests, losing reference to the file system is a problem, as the current implementation of processing the simulation files largely requires the use of file system operations, which is impossible given the lack of file system within the APK.

As such, rather than rewrite all the code used for processing simulation files, a solution was implemented which still made use of the streaming assets folder, where all files from the streaming assets folder (stored as data within the APK), were copied out to the Oculus system (where they would have originally been manually placed in), and the program will access those files instead. This left one last issue to solve.

Given the lack of file system operations inside the APK, it was impossible to simply copy “all” the files out into the file system. The solution implemented was to convert all the simulation files into a single zip file before building. This made it easy for the program, as it just needed to target that single zip file, copy it into the Oculus file system, and unzip all the files there, allowing the program to use them as normal.

Build Folder

Given that Eyesim now builds into multiple different APKs, there had to be a better way of storing them. As such, a Builds folder was created where all the unique Eyesim Simulations would be built into, and the name of each APK built would be based off the name of the Simulation file used as input. A screenshot of the build folder has been provided below, showing the naming convention of each generated APKs.



Build Script

Given that many of the steps above had to be manually done, such as copying the required C# scripts into the correct folder, or zipping up the simulation files, a build script was created, which would handle all the steps required to build the program, as long as all the required files were present in a specified folder inside streaming assets. The build script has to be called externally through the command line, and the project cannot be open in the Unity Editor when the build script is run, as the Build Script itself will open the Unity Editor, and the same project cannot be opened twice.

A breakdown of the steps it does is included below.

The program targets specifically, the “eyesimX” folder in streaming assets. Any C# files found in the base folder (non-recursive) are copied into the custom scripts folder in Assets. The Unity builder is force refreshed to recompile those files and only continues if there were no compilation errors with those files. The build process then finds the first simulation file it sees, and configures the program name and other settings, based on that simulation file name. Finally, it zips up the whole Eyesim folder into a zip file.
When the program is run on the Oculus, it starts in an empty scene, which waits for the zip file to be copied out from streaming assets into the file system and unzipped. When fully unzipped, the main scene containing the simulation is automatically started.

More information about building the project has been placed in a manual for deployment steps, the content of which has been provided in appendix 2.

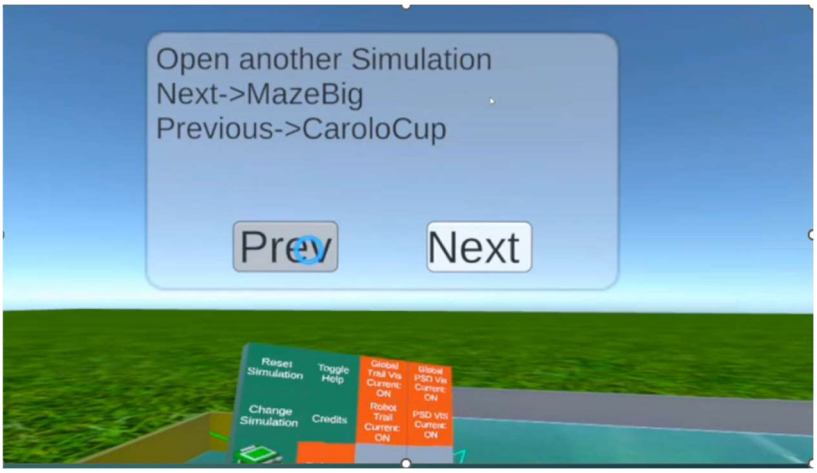
Program Chain

Since Eyesim is now split into multiple programs, having to swop between simulations was a tedious process, where users had to exit the program, find the correct button to open and display all installed programs, find the correct tab to display specifically the EyesimVR programs, and finally, find the actual desired simulation program. This was not a feasible approach especially when EyesimVR has become used a lot more for visitor demonstrations. As such, a system was created which allowed Eyesim programs to chain from one to another, allowing users to select the next, or previous simulation within the Eyesim Program (given that it has been installed). Allowing users to move from one simulation to another from inside the simulation, made providing demonstrations far easier.

This new system involved providing a single text file, named “DemoList.txt” as input to each program before it is built. Each line of this text file should contain the unique program identifier of the desired simulation program, a space separation, and any display name to be shown inside the simulation. The unique program identifier can be found when looking through the “unknown sources” list on the Oculus, whereas the display name can be set to anything.

An example of the contents in a demo list has been provided in appendix 3.

An important limitation of this program chain is that each program will store its own demo list, which means that any change in the demo list will require every program to be rebuilt to have the updated demo list. Having a master list which all programs refer to, would eliminate this problem, and allow the list to be edited as desired. Although placing a master demo list inside the oculus file system was attempted, there were permission errors preventing each Eyesim program to access it. Due to time limitations, it has been left as is, but future work could look into finding ways to feed a master demo list into each program, whether by finding workarounds for the permission errors, or through other means, such as retrieving it from a server.



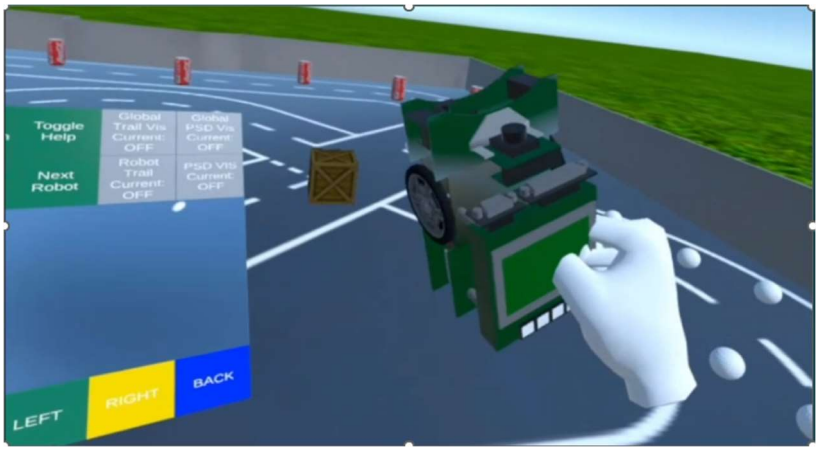
Program chain example

VR Features

Grabbing Objects

Base functionality

The ability to grab both robots and objects, was one of the earliest implemented change, as the base functionality was relatively simple to implement. Oculus has provided some helpful library scripts, one which can be placed onto object to tag the as “grabbable”, and another to place on the hand model, to tag it as a “grabber”. By simply dragging and dropping the corresponding scripts onto the correct object inside the Unity editor, the basic grab functionality was quickly and easily implemented. However, there were a few problems when trying to implement grabbing through scripts, which was needed when loading in custom robots/objects as they would be done in runtime, so the Unity editor could not be used.



Demonstration of grab robot functionality

Buoyancy issues

The provided “grabbable” script always assumes that it will be added by dragging and dropping through the editor, so any variables which need to be adjusted have been set to private, and cannot be edited in runtime. Although the script still works fine for most custom robots, there was a problem when trying to grab robots which have been created with buoyancy, such as submarines.

To know where an object can be grabbed, the “grabbable” script contains a list of Colliders which can tell the hand when it has collided with the object, and thus can be picked up. Inside the Unity editor, the colliders can be manually provided, but if none is given, it will simply initialise with the first collider it finds. This list has also been marked as private, so no outside script is able to edit it once the program has started running. This raises a problem, as to implement buoyancy, a fake collider has to be generated which does not completely match the robot, and more importantly, it is marked as inactive once the buoyancy script has initialised.

As the “grabbable” script cannot be edited in run time, if it’s added to a buoyant robot, could potentially erroneously initialise with that fake collider, and nothing can be done to change it. As the fake collider is marked as inactive, attempts to pick up the robot will cause the program to crash, and changing the fake collider to be marked as active, will cause the robot to collide with objects where it should not.

As such, the “grabbable” script had to be edited to allow the Collider list to be edited at run time and only have the correct colliders added to that list.

Physics issues

Grabbing objects also raises an issue with the physics implemented in Eyesim, where robots would be unintentionally sprung into the air. There were 3 main causes for this effect.

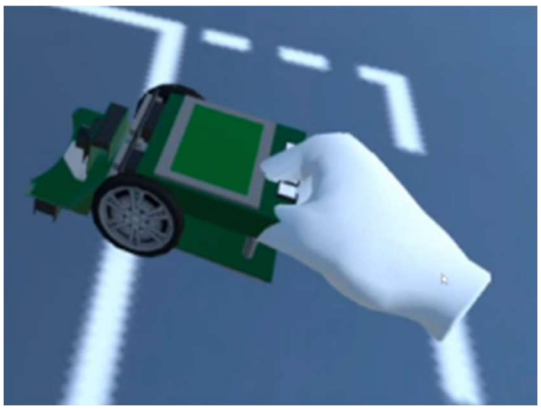
Firstly, Unity uses colliders to calculate the physics of objects knocking into each other, and normally, colliders would not be able to penetrate another collider (i.e., in real life, 2 solid objects are not able to occupy the same space). Given that the position of colliders can be manually edited, users can spawn or teleport a collider to be inside/penetrate another collider. As such, unity has to provide a way to resolve such situations, providing an adjustable “de-penetration” velocity to separate the solid objects. This is useful in VR, as the displayed hand would always track and follow the user’s actual hand movements, and as such, users might grab a solid object, and move it pass a virtual, stationary, solid wall. Unity will simply allow the object to pass through, and if the robot is released inside the wall, the physics engine will apply the pre-set de-penetration velocity to separate them. Steps on how to adjust the de-penetration velocity has been included in appendix 4.

Secondly, the implementation of wheels, in Eyesim, makes use of physics interactions to hold the wheels in place, rather than having a static position relative to the robot. This means that while other components in a robot have a fixed position relative to the rest of the robot, which will NEVER change, the wheels instead can be moved out of position, but will apply a strong force to try and get to its usual position.

As a result, when a robot is grabbed and held through a static wall, all the components of the robot will simply follow, applying the “de-penetration” force when released, except for the wheels, which will instead constantly try to apply a force to push the wheels into the wall.

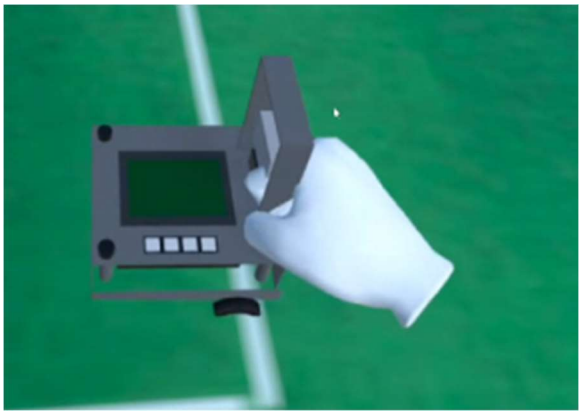
Since the wheels will never succeed in reaching it’s desired position as 2 colliders cannot penetrate each other using force, this causes it to build up a large amount of force. If the robot is released with the wheel positions still inside the walls, the force applied by the wheels to try to get to that position is applied, creating a powerful spring effect launching the robot in the opposite direction, rather than applying the much weaker de-penetration velocity.

Thirdly, the Oculus provided “grabbable “script does try to solve this, by allowing physics interactions to be completely turned off for grabbed objects. However, it does not apply to every component of the robot and crucially, the wheels are also left out. As such, to fix this “spring” problem, the “grabbable” script once again had to be edited to turn physics off for ALL parts of the robot.

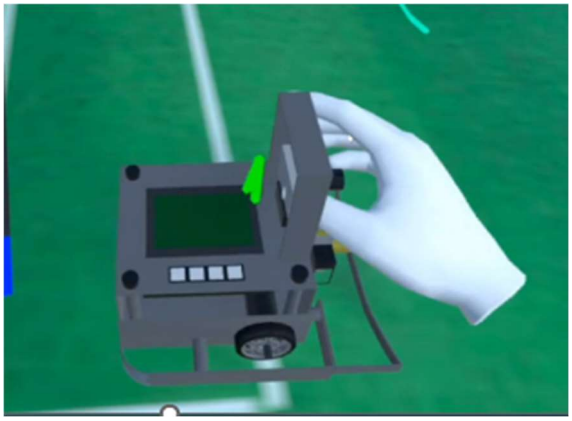


(Previous) Wheels of robot not merging into the ground

(Previous) Force from wheels catapult robot into the air



(Fixed) Wheels of robot correctly merge into the ground



(Fixed) Only small de-penetration velocity applied.

Library File Changes

Both changes stated above were changes to a library script provided by an external source, in this case, the "OVRGrabbable" script provided by Oculus. As such, it is very likely that they will be updated at some point, and if it does, will completely overwrite the current script. Future work or maintenance on EyesimVR should be careful when updating this library script and ensure that if a newer version of the library script is added, the same change is added to the new version, or check if the required functionality is now allowed in the script itself and adjust the current implementation to make use of said functionality.

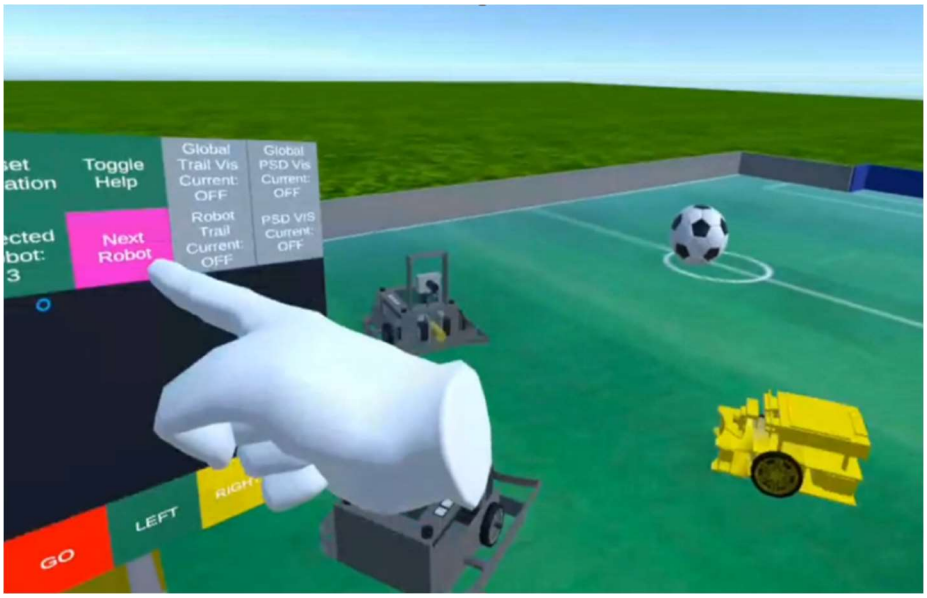
Command window Upgrades

The command window is the panel held on the left hand. It initially did not have much functionality, so one of the biggest set of features added to EyesimVR involved this command window.

Highlight

The ability to highlight robots was a new feature introduced. This allowed for easy identification of the robot that was currently being swopped to. Highlighting of objects involved changing the materials on every component of a robot to a different colour. However, after doing so, there was no easy way to get back the original colour of each robot and it would be stuck on the highlighted colour. As such, a special class "MaterialHolder" had to be created to hold the current colours of the robot so that it could be un-highlighted after a set period of time, or if the user cycles to a different robot.

During initialisation, each robot would create a list of material holders, and search itself and child components recursively for any parts with colours, adding a new "MaterialHolder" into the list and storing the original colour of that part. When the part is to be highlighted, every MaterialHolder changes the colour of it's part to yellow for 1 second, before changing it back the stored colour. As the robot can be highlighted multiple times in quick succession, a counter was also added, which ensure that robot highlights can be immediately turned off if switching to a different robot, and to prevent the robot from reverting back too early, ensuring that only the final call to turn on the highlight will be the one reverting the colour back.



Example of a robot being highlighted when it gets selected

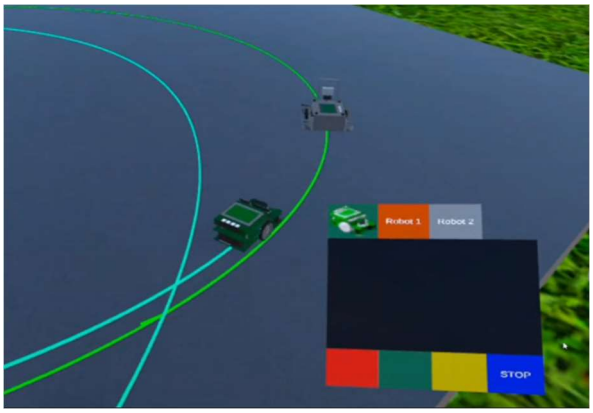
Select/Cycle through Robots

In the previous iteration of Eyesim VR, there was no identification of specific robots. ALL robots would take in the same robot script, all robots would be controlled by the command window buttons at the same time, and all robots would try to output to the screen at the same time.
A large set of features for upgrading the command window was crucial in individualising the robots.

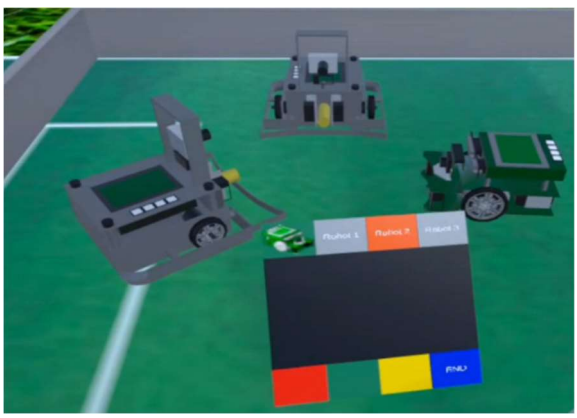
Firstly, the hand controller could now distinguish between each robot, and a button was added which would display which robot was being selected, as well as highlight the robot, if pressed.

Secondly a button was implemented which could cycle between the robots, temporarily highlighting the newly selected robot.

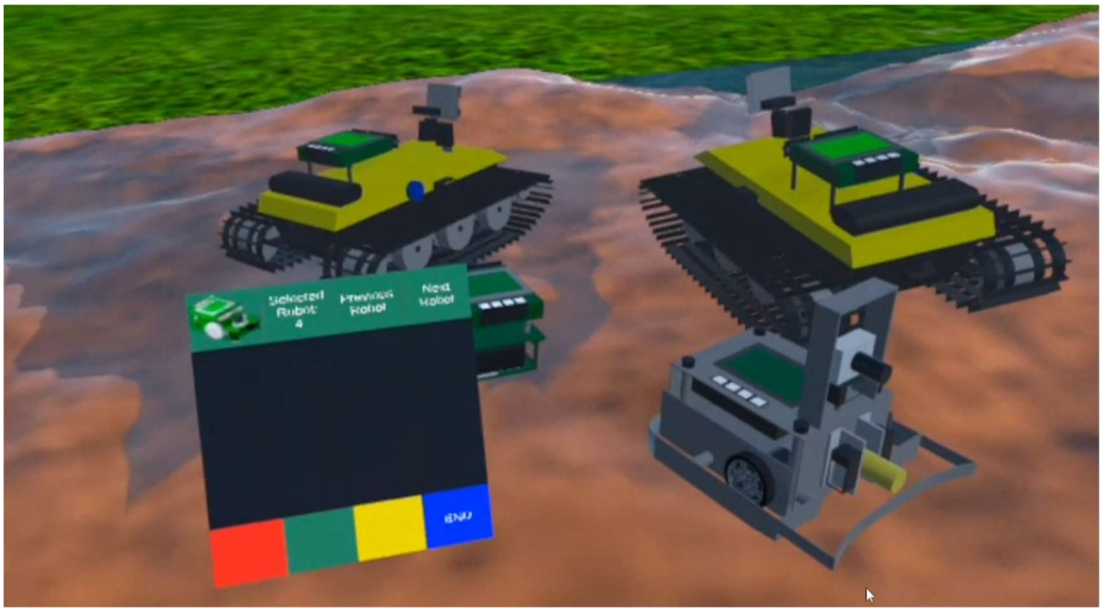
The command window would also morph itself based on the number of robots in the simulation. If there are no more than 3 robots, the command window would display a button for each of the existing robots, which would allow users to select and control each robot directly. If there are more than 3 robots, the command window would instead display buttons to cycle forwards or backwards between the robots, as well as have a button to display and highlight the currently selected robot.



Command window with 2 robots



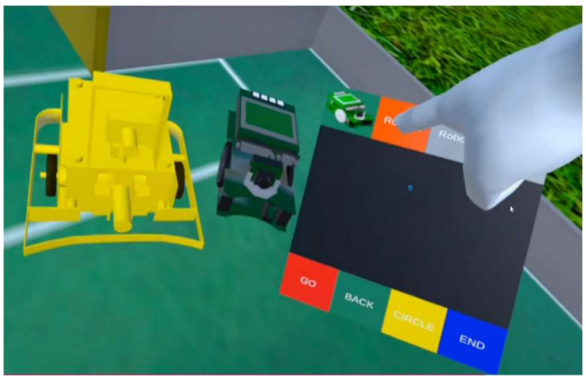
Command window with 3 robots



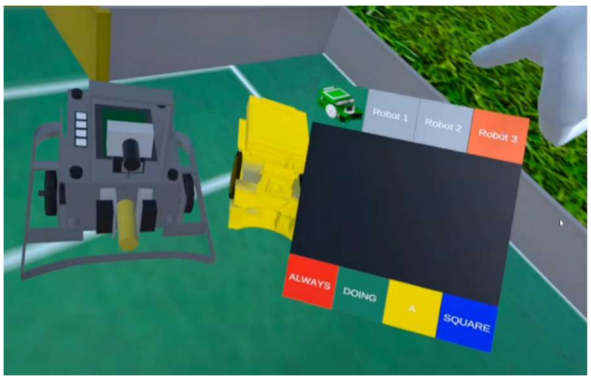
Command window with 4 robots showing "next" and "previous" rather than individual robot numbers

Programmable buttons

On the bottom of the command window are 4 buttons which users can assign text and functionality to. The command window will now show the correct text for each robot, automatically changing it when cycling through/selecting a new robot. Additionally, pressing the buttons will now only target the currently selected robot, and produce the correct functionality for that specific robot.

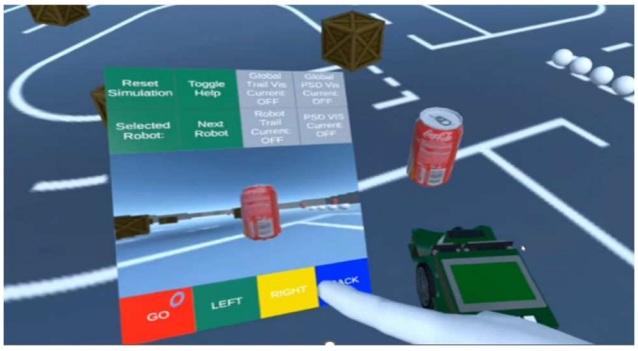


Robot 1 with unique programmable button text



Robot 3 with unique programmable button text

The command window's output screen will now only display output for the currently selected robot, and will automatically clear itself when selecting a different robot. Camera output has also been implemented, so robots can use the camera output function to display images, instead of text, on the screen.



S4 Robot showing camera output

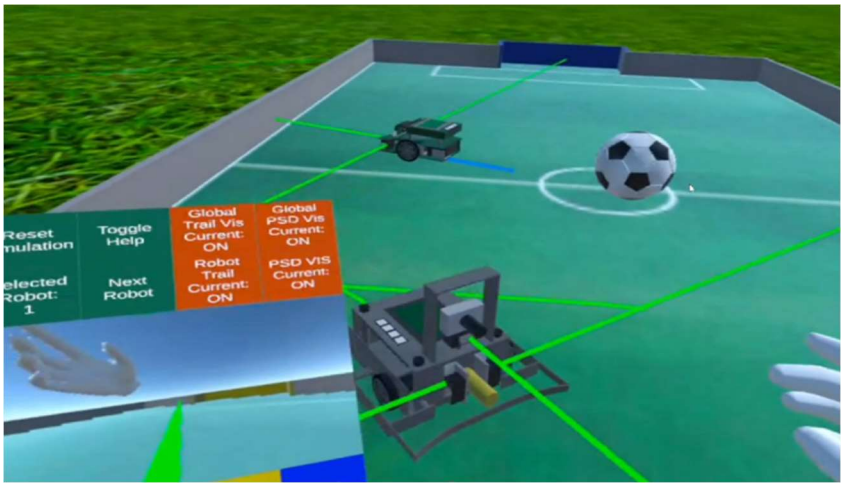
Trail/Sensor Visualisations

Whilst trail/sensor visualisations can be declared in simulation files to enable/disable it, there was no way to manually change it inside the simulation itself. As such, 4 new buttons were implemented which allowed users to enable/disable them. 2 buttons were implemented which could toggle trail or sensor visualisations for a specific robot, and 2 more were implemented which would toggle each of them globally, on all robots.

Button State colours

Previously in EyesimVR, all buttons on the command window were simple push buttons, which activated a function when pressed. As such, it would made sense that the buttons would simply change colour when pressed on, and change back when not pressed on.

However, with many of the new functionalities added to the command window, there were a few buttons which were directly used to represent the state of a certain variable, for example whether the trail visualisations were on or off. With these kinds of buttons, it would be more intuitive to the user if they instead displayed the state of that variable, directly on the button itself. As such, state-based colours were implemented for all buttons which had some state associated to it, with orange chosen to represent “on”, and grey for “off”. Some examples of such buttons include the visualisation options mentioned directly above, or the selection of robots 1-3, which would show orange for the selected robot, and grey for the other 2.



All robots with sensor and trail visualisations on

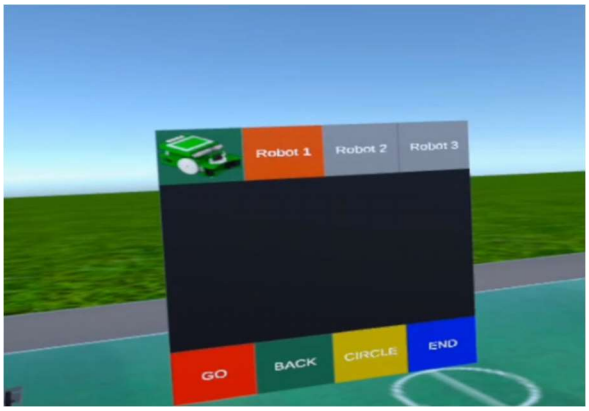
No trails, and only 1 robot with the sensor visualisations on

Show/Hide top row

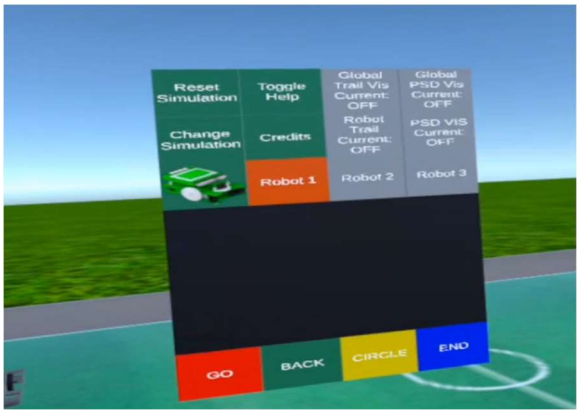
As there were many new functionality being introduced to the command window, new buttons had to be added to it to use said functionality. This resulted in the command window becoming a bit bloated, with too many buttons being displayed at once.

To counter it, all buttons except for those used to select robots, were initially hidden, and a new button was implemented to show/hide all the other buttons.

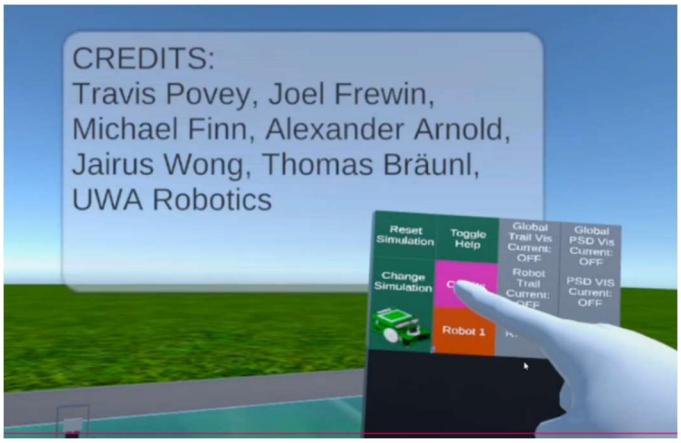
This opened up space for an extra button, so a credits button was added.



Command Window top buttons hidden



Command window top buttons shown

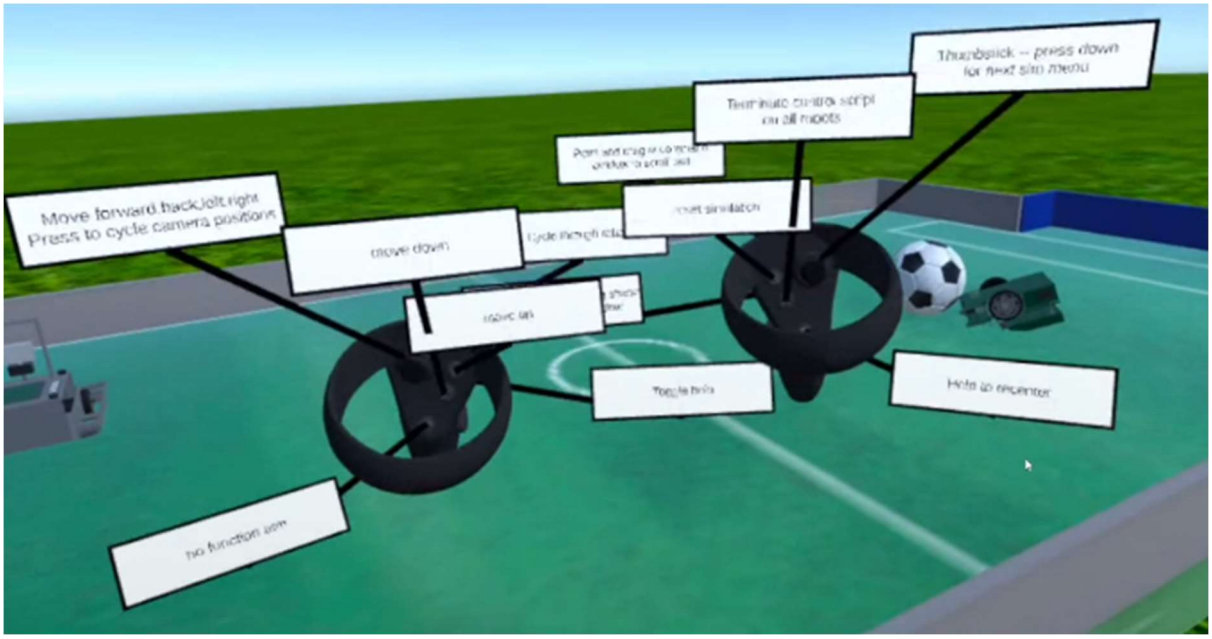


Credits button

Help Mode

Learning the EyesimVR system might be difficult for new users, especially since the buttons on both controllers all have unique functions which will not be immediately obvious to the user. As such, a help mode was enabled, which could allow new users of the system to learn what each button on the controllers do.

A button can be pressed on the command window (and on the left controller too, but it would not be immediately obvious to new users) to toggle help mode, which would turn off all functionality for the buttons, but display a model of the actual hand controllers inside the simulation, and show text-based help popups coming out from each button to describe its functionality. This was implemented through making use of library scripts provided by VRTK (Virtual Reality Tool Kit) found in the Unity Asset Store.



Help mode

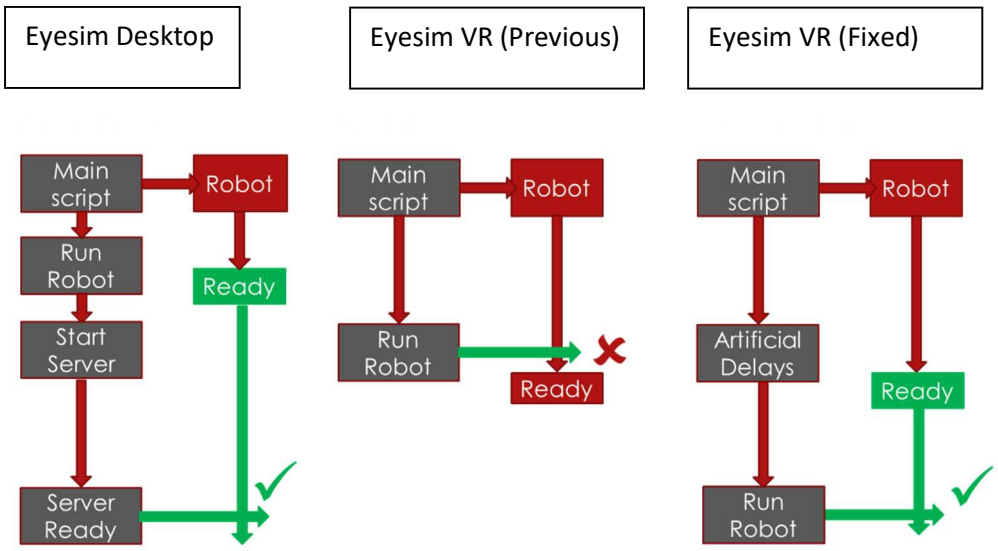
Smaller Changes

Smaller changes generally refer to general changes or bugfixes to the program. Since there were many changes, including some relatively insignificant ones such as changing the display numbers of robots to start from 1 rather than 0, or adjusting wheel positions of some of the pre-made robots, this section will mainly expand on the more significant ones.

Timing Fixes

When loading a simulation, Eyesim would at some point, spawn the desired robots, and after completing some other tasks, would also try to run the robots with their robot scripts. Running robot scripts on robots on EyesimVR, was quite different from the Eyesim Desktop. Whilst the desktop version had a guaranteed long delay between spawning a robot and running said robot as it had to open up an external program to start up a server which would then connect to the robot to control it, EyesimVR did not have much delay as it relied only on a single function call. This caused a problem, as each robot would have scripts attached to it which would need initialising, but not having that long delay, coupled with the fact that the initialising scripts would also take longer to start since the hardware on the Oculus is generally much slower than any desktops, would cause some robots to be run before it was ready, causing the program to crash. As such, an artificial delay was added before running the robots, to prevent it from trying to run too early, almost guaranteeing that the robot would be initialised before it is run.

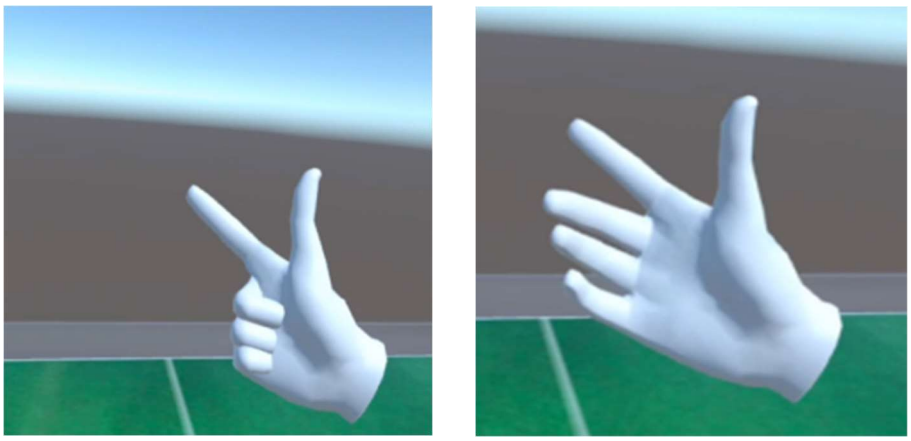
A diagram has been added below, to show the difference in running a robot, and how the timing problem was fixed by simply adding some delay.



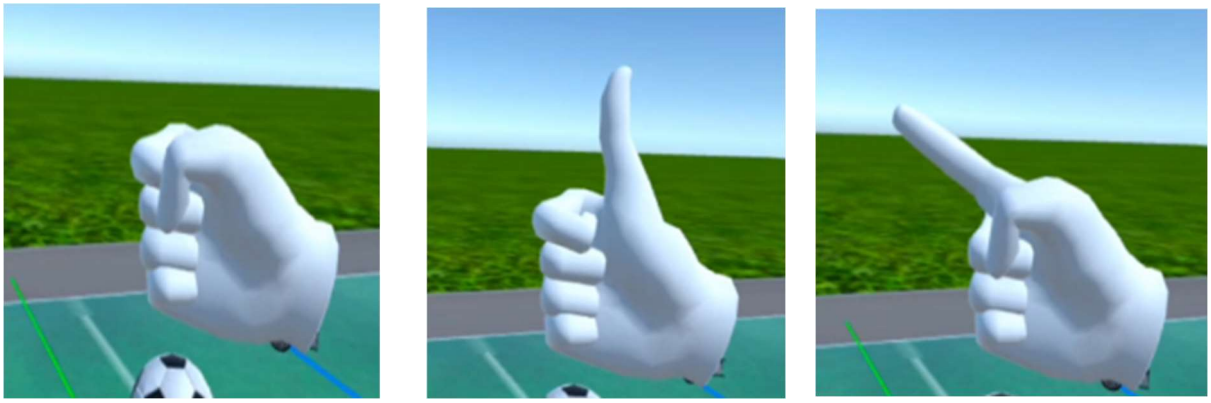
Hand Display

To implement hand display and hand tracking, EyesimVR uses some Oculus provided pre-made hand displays which can simply be added into the project. However, after updating the whole project to the latest unity version, parts of the hand get locked in a set position.

Attempts to update the hand caused it to be de-referenced from many important scripts, and would also cause some scripts to be completely removed from some un-related objects. As such, updating the hand required re-attaching the missing references.



Hand could only display these 2 positions



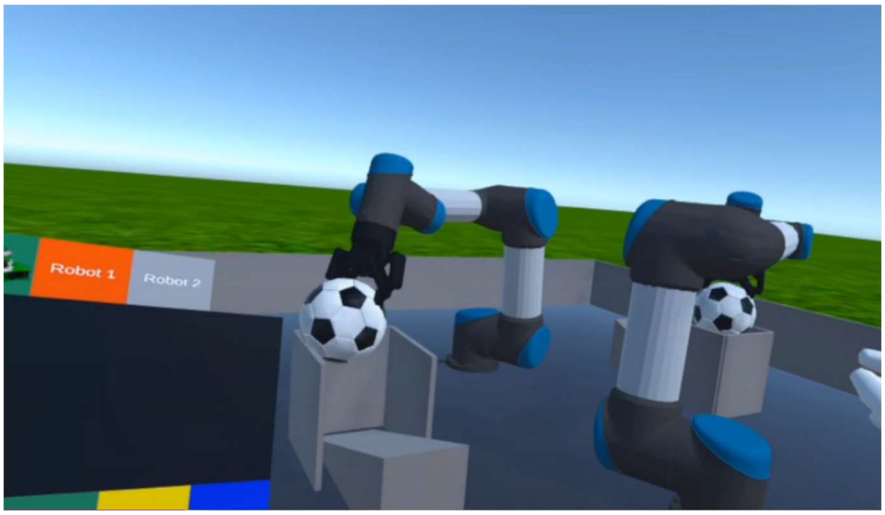
Fixed -> Hand can now display full set of positions

Bounce

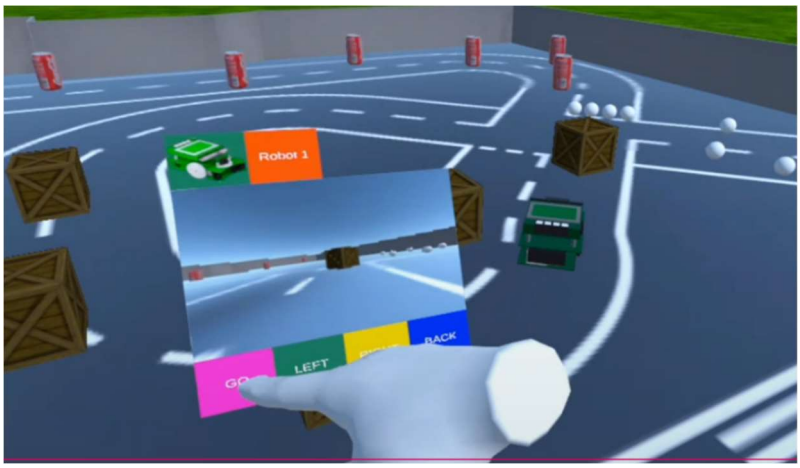
Eyesim allows for custom objects to be added into the simulation. These custom objects have to have an object file, which represents certain important properties which the object is going to have, for example mass, scale, or buoyancy. An additional property has been implemented, “bounce” which can be specified to add “bounciness” to an object. When “bounce” is declared, a value between 0 and 1 has to be provided, which represents the percentage of energy retained when the object hits another solid object, so 0 would represent no bounce, and 1 would represent maximum bounce.

Demo programs

Eyesim Desktop comes with a large set of demonstration simulations, however none of the robot scripts used were written in C#, which is required for EyesimVR. As such, a few demonstration simulations were made using robot scripts which were converted to C#. Shown below, are some demo simulations made which needed a new set of robot scripts to be converted to C#.

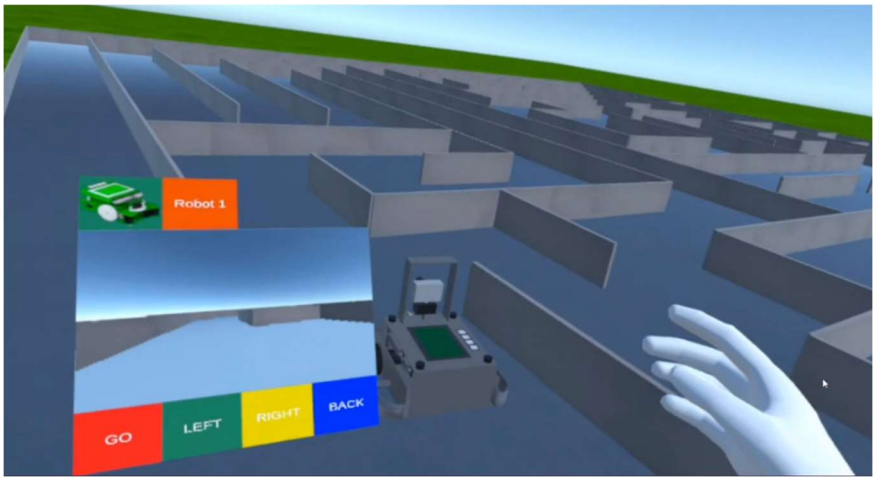


Ur5 -> 2 Manipulator Arms which picks up and drops soccer balls

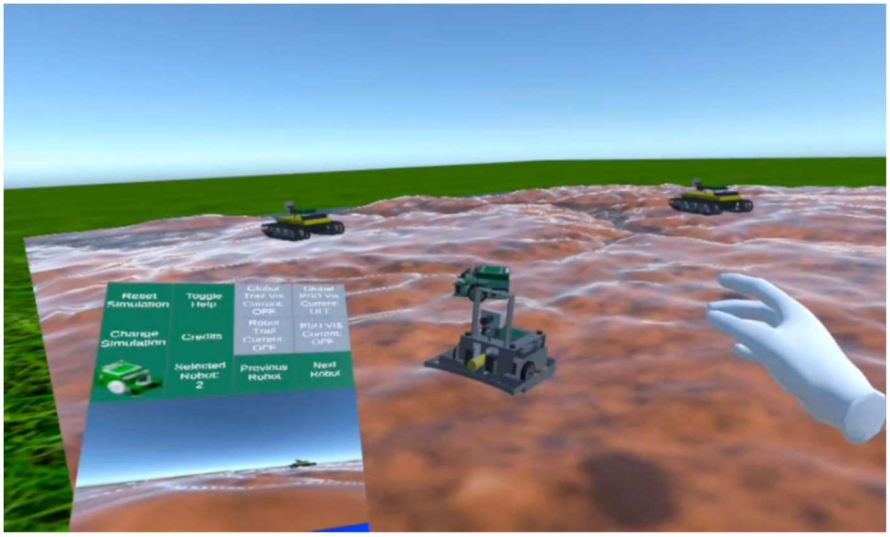


Carolo Cup -> Controllable robot in a racetrack with different obstacles

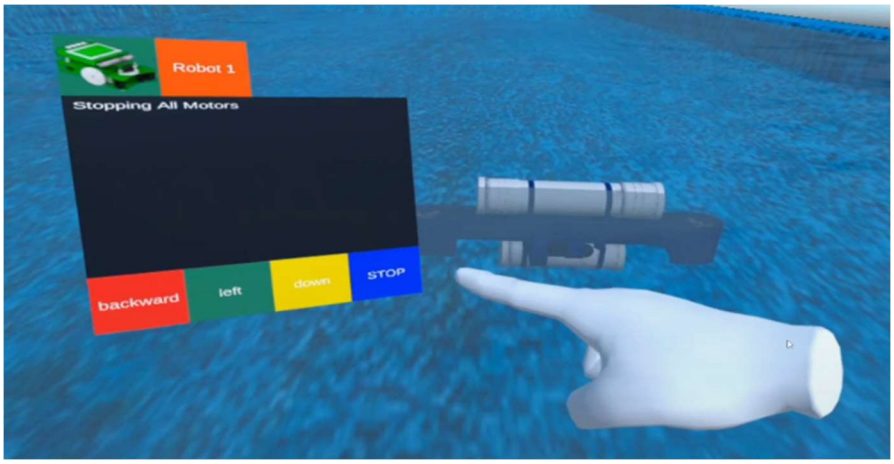
RandomSquareSpeed -> Soccer field where 3 robots are running separate programs: random, square and circle (the script to go in a circle was originally called “speed”, hence the name)



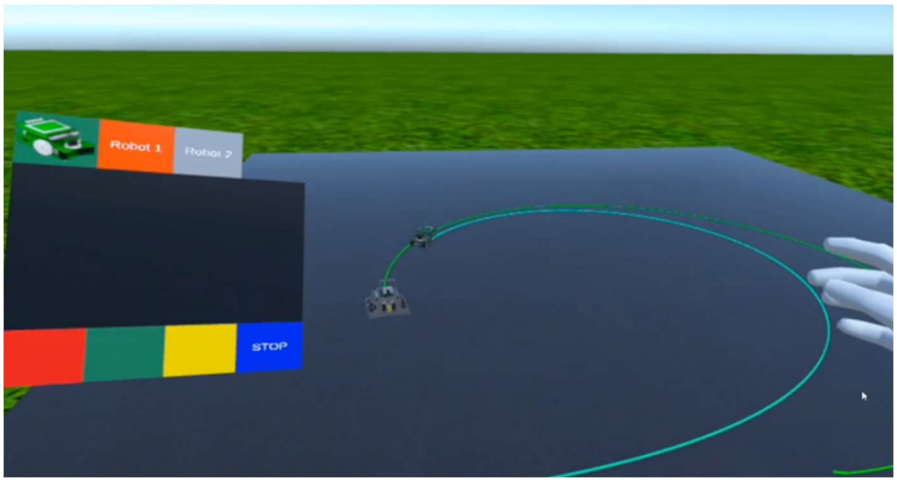
Maze Large -> 1 controllable robot in a large maze



Mountain -> 4 robots moving randomly in difficult terrain



Mako -> 1 controllable submarine



Swarm Follow -> 1 leader robot doing a circle, and a follower robot following it

Conclusion

This project has added many important features into EyesimVR. The system changes introduced has allowed it to capture many desirable properties from Eyesim Desktop whilst working around the limitations of the Oculus' android system.

Additionally, EyesimVR usage as a demonstration tool has been significantly improved, with the implementation of a better system to move between simulations, and the creation of a set of demonstration programs.

The VR features implemented have allowed EyesimVR to better utilise the hand controllers/tracking and enables a more comprehensive set of interactions with robots/objects within each simulation. Finally, many smaller changes have been made which makes EyesimVR improves the presentation/performance of EyesimVR, and fixes many problems arising from the differences between EyesimVR and Eyesim Desktop.

As such, all these new features implemented has allowed EyesimVR to be effectively used for demonstrations, whilst being much closer to being able to be used as an education tool.

Future Work

Eyesim merge

As stated in the constraints for this project, EyesimVR and Eyesim Desktops' "Eyesim Library Files", representing general simulation functionality, are supposed to be exactly the same, but have deviated from each other. As such, a future project could be the merging of both EyesimVR and Eyesim Desktop, possibly using operating system checks to enable/disable features pertaining to VR/desktop, which could ensure that both version will be using the exact same set of Eyesim Library Files when generating a simulation.

Build Script

As mentioned in the main system change, a build script was created to handle all the file operations required, such as zipping, copying, and recompiling, in order to build the desired simulation. However, the build script does not run when directly building from inside the Unity Editor, as Unity provides its own build script. Whilst this is fine for students, who should not be opening up the Unity Editor in the first place, it can get quite tedious for future maintenance on EyesimVR, as every time a change to a simulation is made, the Unity Editor would have to be closed to run the build script externally. As such, future work could look into finding a way to implement the provided build script into Unity's default build script.

Demo list

As mentioned in the Program chain section, the current implementation of the program chain requires each Simulation program to have its own version of a Demo List, where any desired changes to the Demo list would require all programs to be rebuilt with the new version. Having a master list to control all programs is a much better system, and so future work could look to implement this, possibly by fixing the permission errors on the Oculus device, or through downloading the list from a server.

General VR headsets

When EyesimVR was created, different VR companies had vastly different types of headsets and controllers, and as such, it made sense to just focus on one and create EyesimVR specifically for it, in this case, the Oculus Quest 1/2. Given the advancements in VR technology, most headsets and

controllers have become more standardised, so Unity has started to provide libraries which can handle multiple types of headsets/controllers. As such, it is currently a lot more feasible to make a general EyesimVR program which can be run on multiple types of VR headsets, so future work could look to implement that.

Sound

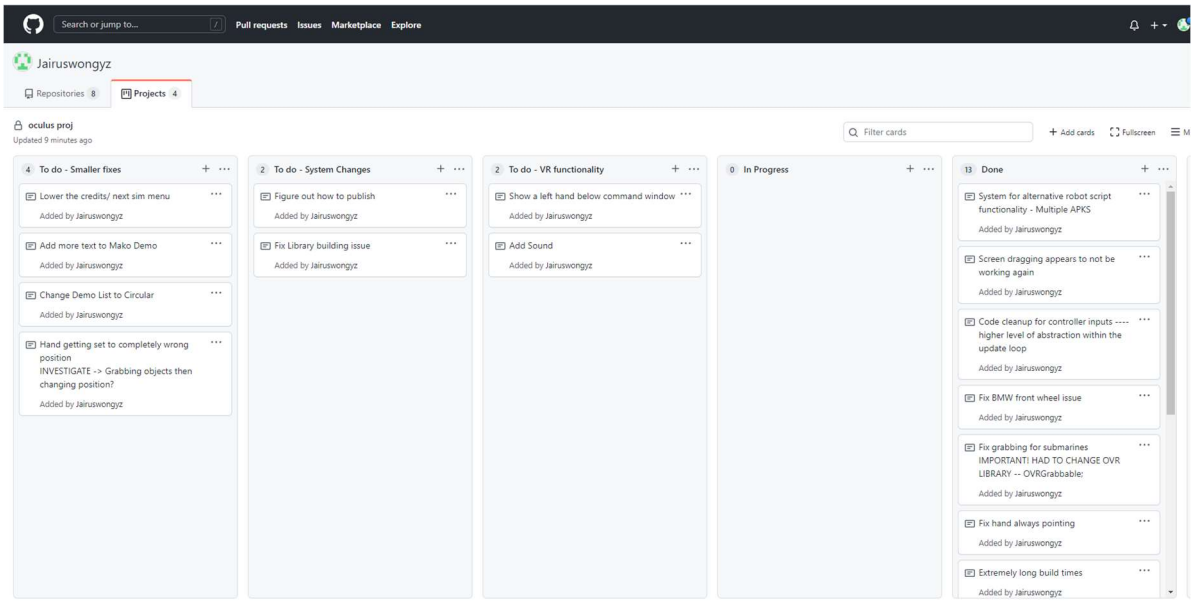
At the moment, EyesimVR is completely silent. A useful feature to improve the general EyesimVR experience, would be the implementation of sound, some examples include robots playing a sound when motors are moving, or when robots collide with each other with enough force.

Publish

The many features implemented in EyesimVR has allowed it to be used a demonstration tool. Whilst this is currently used only for guests visiting the campus, an extension of a demonstration tool would be to allow anyone (with an Oculus) to download and try out some of the Demo programs too. As such, future work could look into ways to publish some of the Demo Programs on the Oculus store, for anyone to download and interact with.

Appendix

1- GitHub Project page



2- Deployment manual

EyesimVR Building APKs

SETUP

Go to OculusProject-MultipleFileTemplate\Assets\StreamingAsset\eyesimX

Place all the required files inside – sim files, C# scripts, world files +++

- Ensure that the sim file and C# scripts are placed DIRECTLY inside the eyesimX folder, not in any subfolders, or they will not be detected.
- The program will build using the first sim file it sees, so make sure to only have 1 sim file inside to avoid confusion.
- A DemoList.txt can be added DIRECTLY inside the eyesimX folder, to create the chain of programs, but each program will need to be built with the same DemoList. – Should look into finding a good place to have a Master List, but for now, each program has their own local copy which will need to be updated manually every time a new list is wanted.
    - Each line should be -> <UniqueProgramIdentifier> <What you want to call it>
    - IE -> com.EyeSimVRTeam.eyesimCaroloCup CaroloCup

Running the build program

‘ <Unity Executable Location>’ -quit -projectPath OculusProject-MultipleFileTemplate/ -executeMethod BuildScript.PerformBuild -logFile –

‘ <Unity Executable Location>’ -> mine was '/d/Program Files/Unity/2020.3.30f1/Editor/Unity.exe'

- Ensure that the Project is NOT open in the unity editor.
- I have placed these commands in deploy.sh -> should be easy to run once the unity executable and project paths have been adjusted for each computer.
- An APK will be built into OculusProject- MultipleFileTemplate\Builds
- Name will be based off the sim file used

What Actually Happens

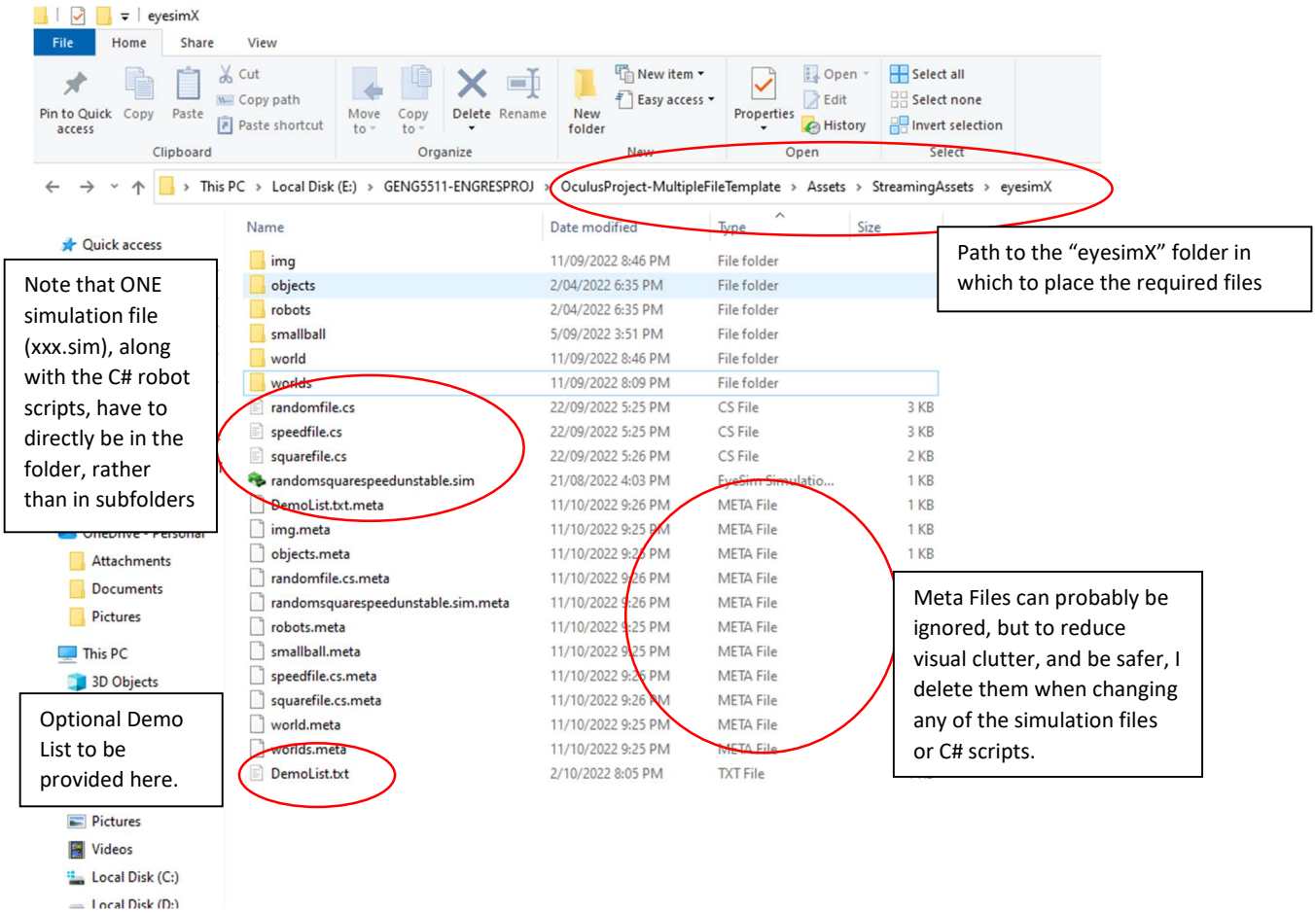
When running with the build script, the build script will do the following things.

1) Copy out every C# script inside …\Assets\StreamingAsset\eyesimX (does NOT look into sub folders), into …\Assets\Scripts\CreateRobot\CustomScripts
    a. (Clears out …\Assets\Scripts\CreateRobot\CustomScripts before copying, so any pre-existing scripts will be deleted)
2) Do a recompile of the project WITH those new scripts inside. Any errors with the scripts will cause the project to NOT compile and will prematurely end the building process.
3) If there were no errors with the scripts, the first .sim file found inside the eyesimX folder will be used for setting the program name/identifer +++
4) …\Assets\StreamingAsset\eyesimX will be built into a zip file …\Assets\StreamingAsset\eyesimX.zip
5) The program should then build the APK into the Builds Folder.

The buildscript is stored in …Assets\Scripts\Editor\BuildScript.cs which should contain the code for the steps above.

^ These steps are **NOT** performed when building directly from the Unity editor itself (still not sure how to integrate the build script into the Unity editors building process) and as such, if a change to any file inside …\Assets\StreamingAsset\eyesimX is required, the above steps have to be done manually, OR you can just close the Unity editor and run the build program, which will take care of all the steps.

Below is a helpful screenshot with important points about building the project.



3- Program chain -> demo list

Below is an example of the contents of a demo list, simply the official identification name of the program, followed by any desired display name. ie -> (program ID) (program display name)

com.EyeSimVRTeam.eyesimFollow Swarm-Follow

com.EyeSimVRTeam.eyesimUR5 UR5

com.EyeSimVRTeam.eyesimCaroloCup CaroloCup

com.EyeSimVRTeam.eyesimrandomsquarespeed randomsquarespeed

com.EyeSimVRTeam.eyesimMazeBig MazeBig

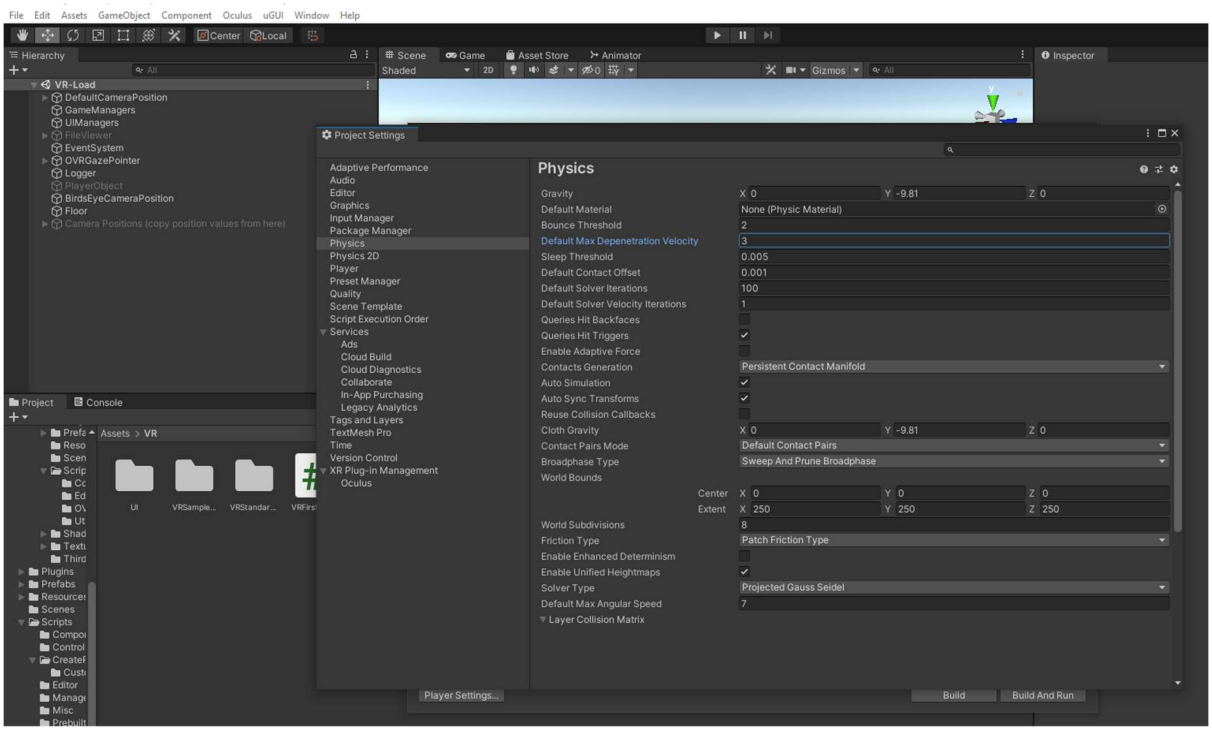
com.EyeSimVRTeam.eyesimMountain Mountain

com.EyeSimVRTeam.eyesimMako Mako

4- De-pentration Velocity

The “de-pentration” velocity value can be set by pressing on the following options

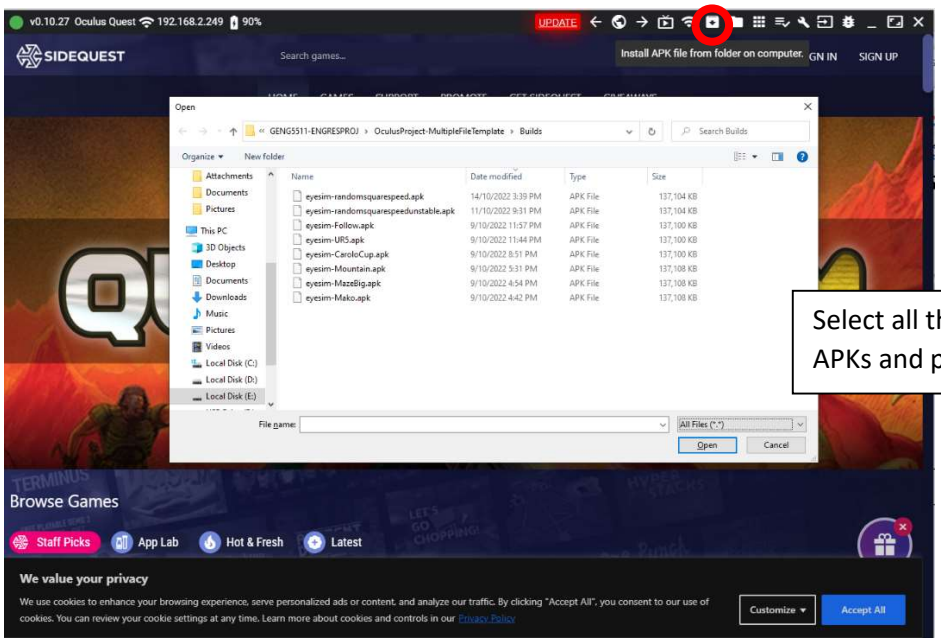
File -> Build Settings -> Player Settings -> Physics. A screen shot has been provided below



5- SideQuest

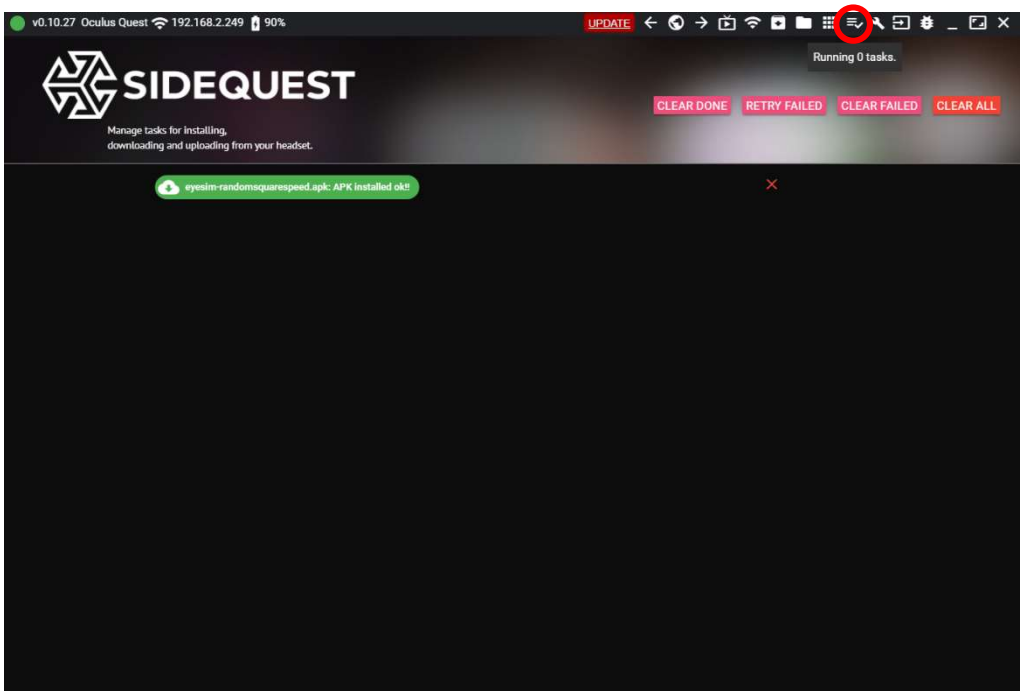
The dot on the top left represents the status of the Oculus. Ensure that the Oculus is plugged in, and developer mode is enabled.

Press on this button to open the file explorer to search for any APKs to be installed



Select all the desired APKs and press Open

Press this button to look at completion status of the APK installations and any potential errors



References

[1] Olivier Michel, “WebotsTM: Professional Mobile Robot Simulation,” International journal of advanced robotic systems, vol. 1, no. 1, pp. 39–42, 2008.

[2] J. Ng and T. Bräunl, “Performance Comparison of Bug Navigation Algorithms,” Journal of intelligent & robotic systems, vol. 50, no. 1, pp. 73–84, 2007, doi: 10.1007/s10846-007-9157-6

[3] O. Karoui et al., “Performance evaluation of vehicular platoons using Webots,” IET intelligent transport systems, vol. 11, no. 8, pp. 441–449, 2017, doi: 10.1049/iet-its.2017.0036.

[4] E. D. de Bruin, D. Schoene, and G. Pichierri, “Use of virtual reality technique for the training of motor control in the elderly: some theoretical considerations,” Zeitschrift für Gerontologie und Geriatrie, vol. 43, no. 4, pp. 229–234, 2010, doi: 10.1007/s00391-010-0124-7.

[5] H. Lesch, E. Johnson, J. Peters, and J. C. Cendán, “VR Simulation Leads to Enhanced Procedural Confidence for Surgical Trainees,” Journal of surgical education, vol. 77, no. 1, pp. 213–218, 2020, doi: 10.1016/j.jsurg.2019.08.008.

[6] S. Hindlekar, V. Zordan, E. Smith, J. Welter, and W. Mckay, “MechVR: interactive VR motion simulation of ‘Mech’ biped robot,” in ACM SIGGRAPH 2016 VR Village, 2016, pp. 1–2. doi: 10.1145/2929490.2932422.

[7] Elkin, Eugene. “Porting Your VR Title to Oculus Quest.” In ACM SIGGRAPH 2019 Talks, 1–2. ACM, 2019. https://doi.org/10.1145/3306307.3328202.

[8] T. Braunl, Robot Adventures in Python and C. Cham: Springer International Publishing AG, 2020. doi: 10.1007/978-3-030-38897-3.

[9] D. King, S. Tee, L. Falconer, C. Angell, D. Holley, and A. Mills, “Virtual health education: Scaling practice to transform student learning,” Nurse education today, vol. 71, pp. 7–9, 2018, doi: 10.1016/j.nedt.2018.08.002.

[10] W. Alhalabi, “Virtual reality systems enhance students’ achievements in engineering education,” Behaviour & information technology, vol. 35, no. 11, pp. 919–925, 2016, doi: 10.1080/0144929X.2016.1212931.