GENG4412/5512 Engineering Research Project

EyeSim Version 2: Robotics Simulator Project Final Report
Semester 1, 2026

Author: Aditya Kishor Patil

Student Number: 23367345

School of Engineering, University of Western Australia

Supervisor: Thomas Bräunl

School of Engineering, University of Western Australia

Co-Supervisors:

Travis Povey

Previous UWA student, University of Western Australia

Michael Finn

School of Engineering, University of Western Australia

Word Count: 6351

Submitted: 25th May 2026

Project Summary

EyeSim is a robotics simulator that forms a critical component of education at the University of Western Australia, enabling students to develop and test C programs for physical robots in a simulated environment before hardware deployment. The original EyeSim version 1 (V1) simulator suffered from two compounding problems: a dependency on the XQuartz windowing system that caused frequent installation failures on modern operating systems, and a monolithic, non-modular codebase that had accumulated eight years of inconsistent multi-developer contributions, making the platform increasingly difficult to maintain and extend. V1's codebase had robot classes with code ranging from 500-900 lines per file, resulting in heavily duplicated code across robot variants. This repetition created immense maintenance friction, as introducing a bug fix or new feature carried cascading regression risks, requiring manual synchronisation across all monolithic files. These structural limitations made the platform increasingly fragile and resistant to expansion. This project tested the hypothesis that a complete component-based architectural redesign of an educational robotics simulator could eliminate legacy dependencies and improve maintainability without sacrificing backwards compatibility.

EyeSim version 2 (V2) was initiated to eliminate these issues while adding new features and fixing old ones along the way by rebuilding the simulator entirely within the Unity 6 game engine. The XQuartz display window was ported over purely within Unity using its built-in UI system, and development of features was done with a major reconstruction of the previous architecture with a heavy focus on modularity by using a component-based architecture and ensuring future development updates are taken in consideration. Network handling and robot classes are all moved to a component-based system as previously these were monolithic code files. A new thread separate for parsing network commands was introduced to prevent network packets from stalling the main simulator due to incomplete packets. EyeSim V2 also supports all features of the prior simulator including robot archetypes of differential-drive, submarines and articulated manipulators with full RoBIOS compatibility over Transmission Control Protocol (TCP) sockets.

The resulting simulator successfully executes legacy programs without modification, proving that removing legacy dependencies does not have to sacrifice backwards compatibility. This was verified against a complete test suite of 268 environment files and 95 C programs, which ran as expected across MacOS, Windows, and Ubuntu platforms. The new multi-threaded networking architecture also delivered major performance improvements; benchmark tests under incomplete packet errors showed a 24.7x increase in responsiveness, recovering in ~99ms compared to V1's ~2453ms system stall. Furthermore, the component-based architecture proved its flexibility during the implementation of a hybrid differential-manipulator robot, which required only a single 159-line file and two lines of registry code with zero modifications to existing classes. EyeSim V2 is fully prepared for deployment in UWA's ELEC3020 Embedded Systems labs from semester two 2026, and AUTO4508 Mobile Robots from semester one 2027. This stable, cross-platform release allows students

and teaching staff to finally focus purely on the practical robotics curriculum rather than troubleshooting simulator installation issues.

For future work developers should consider adding a GUI-robot editor to remove the need for manual robi text file editing to create new robots as it is very technical and not user-friendly. JSON Parsing support would also be beneficial alongside the text files for importing, as JSON supports automatic syntax validation which text files do not. Finally, develop an automated regression testing pipeline to guarantee future updates don't break the backwards compatibility and can ensure that a full test can be executed swiftly as opposed to testing against 200+ example files manually.

Ultimately, the success of EyeSim V2 confirms the initial hypothesis: a complete component-based redesign can eliminate fragile legacy dependencies and significantly improve system maintainability without sacrificing backwards compatibility. Beyond the immediate technical fixes, this project highlights the concrete value of upfront software design. Taking the time to properly plan and build a modular, decoupled architecture rather than just rushing to implement features pays off massively in the long run. It results in a highly stable platform where extending the codebase is straightforward, regression risks are kept to a minimum, and the simulator can reliably serve UWA students and staff for years to come.

Acknowledgements

The author would like to thank Thomas Bräunl for his continuing guidance as a project supervisor and for the original vision behind EyeSim V2. The author extends his gratitude towards Travis Povey and Michael Finn for co-supervision and the development on EyeSim V1 features and the start of development of EyeSim V2 alongside continuous support when requested.

Table of Contents

1. Introduction

1.1 Context

Robotics education at the University of Western Australia has historically maintained a sustained dependency on EyeSim, a proprietary simulator that facilitates the learning and development of students in robotics. Users write C programs for simulated robots using the RoBIOS API (Bräunl, 2008) and then run those same programs on physical raspberry Pi-based robots known as Eyebots. RoBIOS is a C library providing a hardware abstraction layer that allows identical code to run on both physical Eyebots and the simulator. The simulator accurately recreates the kinematics and physical behaviour of software executed on these physical Eyebots. Compiling and running a program on the simulator requires substantially less time than deployment onto physical hardware, which requires utilising File Transfer Protocol (FTP) and connecting to the Eyebots via local networking. Additionally, it removes the requirement of having the physical robot with a user to test programs on. Consequently, EyeSim is a critical infrastructure component for student learning and rapid development.

1.2 Issues with EyeSim

The EyeSim simulator, however, contains issues that hinder student learning and development. While modern hardware performance often masks underlying runtime performance, the simulator's underlying architecture lacks modularity and exhibits inconsistent development methods. Developed over eight years ago, EyeSim's codebase has been subjected to numerous feature expansions by independent developers. Without strict design governance, these iterative modifications have caused severe architecture drift, leaving the platform brittle and difficult to update. EyeSim also relies on the XQuartz windowing system, which is an X11 display server that must be separately installed on MacOS devices. Its purpose is to simulate a graphical canvas that displays text and drawings to showcase information during runtime. An issue with X11 is its lack of maintenance by Apple for MacOS and frequently malfunctions with modern MacOS updates. In addition, X11 also has installation issues on many computer devices. Since devices run different versions of operating systems, some features may be incompatible with other devices and results may not be the same on each device. This leads to complications with executing programs and lengthy troubleshooting during student labs. This requires EyeSim to be frequently maintained, yet combined with its architecture drift, has become challenging as it leads to resolving cascading changes due to the architecture issues of EyeSim.

A further deficiency in the legacy EyeSim codebase is its non-modular, monolithic architecture. Rather than decomposing robot behaviour into independent, composable systems, the V1 design concentrates responsibilities across two architectural layers that each suffer from the same problem: a lack of separation of concerns.

At the network layer, all incoming RoBIOS commands are routed through a single 889-line translator class that contains switch statements with over 40 branches. Adding any new

command requires directly modifying this central class, making the system brittle and closed to extension.

At the robot layer, each drive type is implemented as a monolithic class that simultaneously implements nine distinct interfaces. Both the differential and submarine classes follow an identical structure of utilising similar components such as positional sensing devices (PSDs), radio modules and camera feeds. Instead of separating the usage of components for reusability, EyeSim V1s architecture requires a developer to re-write the classes and components for every new drive type, which results in an inefficient structure, lack of modularity, maintenance resolving issues and difficulty in implementing additional features.

The robot configuration loader exhibits the same pattern: a single *process()* method handles file parsing, drive-type instantiation, and all component configuration for six robot variants through deeply nested switch statements, making it increasingly difficult to add new robot types without risking regression of existing ones.

These architectural and dependency issues have caused EyeSim to be difficult to maintain and update with newer features. This motivates the central hypothesis of this project: whether a complete architectural redesign of software (more specifically robotics simulators) and focus on long-term software design can eliminate legacy dependencies while improving maintainability without sacrificing backwards compatibility.

1.3 Introduction of EyeSim V2

EyeSim V2 was introduced to resolve these issues: a complete re-design of the simulator, starting with a blank canvas in Unity 6 to build a solid foundation from the beginning to ensure an increased longevity of V2 over V1 and modular architecture development. The central hypothesis is that component-based architecture, when applied from the ground up, produces a simulator that is both easier to maintain and straight-forward to extend than its monolithic predecessor. To resolve the XQuartz X11 dependency, V2 uses a panel developed within Unity using unity UI that matches the X11 display. This ensures V2 is backwards compatible with previous lab sheets and example programs, while removing a major dependency issue as the display is now simulated within EyeSim V2 itself. The development of the foundation from the ground up was a major decision change as it required a major rework to make it backwards compatible with example robot, environment and simulator files that EyeSim supported.

However, rebuilding from the ground up presents an opportunity to plan a proper modular architecture from the outset, directly addressing the structural shortcomings of V1. EyeSim V2 tackles these problems at both architectural layers where V1 struggled. At the robot layer, rather than implementing all features simultaneously within one large class, it will be replaced by independent components that can be re-used for each robot type to prevent duplication. Modifying a single component therefore propagates consistently to all robots that use it, reducing maintenance effort and improving feature extensibility for developers. This not only accelerates the introduction of new features but also produces a codebase that is

considerably easier navigate: a claim evaluated through a direct architectural comparison with V1.

1.4 Report Outline

This report presents EyeSim V2 as a case study in the application of component-based software engineering principles to educational robotics simulation, demonstrating that a dependency-free, modular architecture can achieve full backwards compatibility with a legacy system while establishing a measurably more extensive foundation for future development.

This report outlines the design and implementation of EyeSim V2, detailing the architectural choices made to address the limitations of its predecessor, the challenges encountered during development, and the outcomes achieved. The scope of future work required to bring EyeSim V2 to completion whilst being backwards compatible with EyeSim V1 files is also discussed, along with recommendations for future work for subsequent developers.

2. Literature Review

2.1 Alternative Simulation Platforms

There are several other robotics simulation platforms that exist for educational and research purposes like EyeSim. Gazebo (Koenig & Howard, 2004) is a popular open-source robotics simulator, integrated with the Robot Operating System (ROS). It provides a powerful physics engine and supports a wide range of sensors and actuators. However, Gazebo is primarily Linux-based, and the ROS dependency introduces substantial installation complexity that makes it unsuitable as a student learning tool on hardware running MacOS and Windows devices. Introduction of ROS would also require a full revamp of learning content of multiple UWA units and entails teaching of ROS to staff and students which is much more complex than RoBIOS which EyeSim V1 and V2 use, hence it is not a suitable replacement.

Webots (Michel, 2004) developed at EPFL and later open-sources under Cyberbotics, provides native cross-platform support and a built in C/C++ robot controller API with a conceptually similar model to EyeSim: controllers are separate executables that communicate with the simulator process. However, a similar problem where the Webots API is entirely incompatible with the RoBIOS interface used in UWA engineering units exists. It would require a complete rewrite of all student-facing content for multiple units yet again, which is not a feasible option due to the time and effort it requires, when there are other options.

The original EyeSim (Bräunl, 2008) preceding version 1 was implemented in C++, using an X11 for its display output. Compared to V2, it lacked a 3D environment and had no physics engine for realistic collision and required all robot types to be built as monolithic source files. V1 addressed a few of these shortcomings, and V2 resolved the shortcomings in V1.

2.2 Component-Based Software Architecture Redesign

Component-based software engineering (CBSE) organises systems as compositions of independent, replaceable modules with well-defined interfaces. In the context of robotics, CBSE moves the development away from building ad-hoc, monolithic architectures to assembling reusable building blocks, which significantly reduces maintenance friction and regression risks (Brugali & Scandurra, 2009). This modular approach allows sensors, actuators, and drive controllers to be developed and tested in isolation before being composed into complete robot configurations. This architectural pattern has become the industry standard and is used extensively in frameworks like the Robot Operating System (ROS), where independent software components communicate over typed topics regardless of the specific hardware in use (Yang & Zhang, 2023).

Unity's architecture is built heavily around component composition using two main elements: *GameObjects* and *MonoBehaviours*. A *'GameObject'* acts as a blank container that represents an entity in the simulation, while a *'MonoBehaviour'* is a specific script attached to that container to give it logic and behaviour. EyeSim V2 uses this exact framework. A robot is simply an empty root *GameObject*, and its individual capabilities (such as PSD distance

sensing or camera tilting) are separate *MonoBehaviour* scripts attached to it. This structural choice is what makes EyeSim V2 inherently modular. It eliminates the old V1 methodology of having to write a massive, custom script file to hardcode the features for every new robot type.

2.3 Unity 6 as a Simulation Platform

The Unity game engine is increasingly utilised for robotics simulation research due to its extensible architecture and real-time physics engine. While EyeSim V1 was developed using Unity 5, the transition to Unity 6 for EyeSim V2 uses substantial advancements in the engine's core physics and networking subsystems. Unity 6 introduces parallelised solver scheduling and optimised multi-threaded physical simulations, which drastically improve the stability of complex physical structures such as robotic joints, rigid-bodies, and articulated manipulators (Unity Technologies, 2026). Furthermore, EyeSim V2's architectural shift, rendering the robot LCD display internally rather than relying on external X11 system calls, requires high-frequency data transmission between the executing C program and the simulator. Unity 6 provides robust, low-latency Transmission Control Protocol (TCP) socket handling that operates securely alongside the main simulation loop. This ensures that the massive volume of bidirectional packet transfers required for real-time sensor polling and display rendering occurs without degrading the real-time physics calculations.

3. Project Objectives

The overarching research hypothesis is that EyeSim V2, rebuilt within the Unity 6 game engine using component-driven architecture and dropping XQuartz dependency will provide UWA students and staff with a more modern, dependency-free robotics simulator that maintains full backwards compatibility with the RoBIOS API and all existing V1 file formats, while also establishing a maintainable, extensible foundation for future development.

The specific objectives are as follows:

1. Maintain full backwards compatibility with existing RoBIOS C programs, robot definition files (.robi files), world files (.maz, .wld) and simulator files (.sim), so that all EyeSim V2 content runs in EyeSim V2 without modification.
2. Eliminate the XQuartz dependency entirely by implementing all display and input functionality using Unity over TCP, such that the simulator runs on MacOS, Windows, and Linux with no external system dependencies.
3. Implement all three supported robot archetypes: differential drive, underwater submarines and articulated manipulators as compositions of independent, reusable components with correct physics, sensor simulation, and RoBIOS command support.
4. Adopt a component-based architecture such that adding new robot types or sensor variants in the future requires composing existing components rather than restructuring the codebase.
5. Produce comprehensive developer documentation covering architecture, file formats and relevant information to lower the onboarding time for future contributors.

Objectives 1 and 2 address the immediate student-facing problems that motivated this project: restoring reliable cross-platform operation and eliminating the installation burden that has disrupted lab sessions. Objectives 3 and 4 define the functional and architectural scope required for deployment in ELEC3020 and AUTO4508, ensuring that all existing robot types are supported while establishing a foundation that future contributors can extend without risk of regression. Objective 5 addresses the organisational issues allowed by V1 to become unmaintainable.

4. System Architecture

4.1 Constraints and Requirements

The architectural decisions in EyeSim V2 were shaped by three non-negotiable constraints inherited from V1, each of which imposed specific requirements on the design. These are:

Two-Layer Architecture: EyeSim is structured around a strict separation between the simulation environment and user-generated C programs to execute. This separation mirrors the real hardware deployment exactly, as a C program that controls a physical Eyebot runs on the robot's Raspberry Pi and communicates with the onboard hardware through its API calls. Thus, the simulator is required to open a TCP connection to receive these incoming messages from the C program and execute them after unpacking the packets.

Backwards Compatibility: The RoBIOS C Library that contains all functions students use to control an Eyebot could not change. This means that the C program should not be able to distinguish between running in a simulator or on the real robots: which ensures backwards compatibility with V1's example files, in line with objective 1. These example files contain 268 world loading files and 95 C programs for testing purposes. Table 4.1.1 shows the breakdown of these files, demonstrating the extent of backwards compatibility examples to test against:

File Type	Count	Purpose
.sim	122	Scene definitions
.wld	47	World files
.robi	35	Robot definition files
.maz	37	Maze files
.esObj	27	Object files
.c	95	C Robot programs

Table 4.1.1: example file breakdown

Platform Portability: EyeSim V1 was available on MacOS, Linux and Windows platforms.

These constraints collectively shaped the architectural decisions described in the following sections: the two-layer TCP model was preserved, the RoBIOS command surface was kept identical, and all existing file formats were parsed without modifications.

4.2 Component-Based Command Handling Network

EyeSim V2 separates receipt of network messages from their execution into two distinct layers. A dedicated background thread listens continuously for incoming commands from the connected C program and places each received message into a thread-safe queue as it arrives (A thread-safe queue is a data structure that allows two concurrent processes to exchange data without interfering with each other). Threading is used to satisfy objective 3: Correct physics simulation as Unity's physics engine requires exclusive main-thread access. A separate process, running in synchrony with the simulation itself, drains that queue once per frame and executes the corresponding robot instructions. This separation exists because the simulation engine requires that any operation affecting the physical world occur within its own update cycle. EyeSim V1 did not enforce this boundary, interleaving network interception and command execution within the same code path and making the system's concurrent behaviour harder to reason about and extend safely.

Beyond threading, command routing in V2 is handled through a registry of independent command factory files, one per command group, covering areas such as drive control, distance sensing, camera output and display. Each factory file is responsible for a single concern: it declares which command types it handles, interprets the incoming message, and produces the appropriate instruction for the robot. At startup, all factories register their supported commands into a shared lookup table held in a central routing file known as *RoBIOSCommandAPI.cs*. When a message arrives from the C program, a single lookup in that table identifies the responsible factory, which processes the message in isolation from all others. Adding support for a new command requires writing one new factory file and adding a single entry to the registry; no existing factory is modified in the process.

EyeSim V1 routed all commands through a single 889-line file containing over 40 command branches. Every modification to that file, regardless of which command it concerned, introduced risk to every other command handled with it, since they shared no isolation from one another. V2's factory registry confines the impact of any change to the file responsible for it. Adding the LIDAR subsystem during development, for example, required no changes to the files handling drive control or sensor reading. This property that allows the system to be extended without modifying existing components, is what made the incremental development of V2's command set tractable, and it stands in direct contrast to the maintenance burden that the monolithic routing structure of V1 imposed. This directly satisfies objective 4, which required that new capabilities be addable without restructuring existing code.

4.3 Component-Based Robot Model

The robot model in EyeSim V2 is structured around separating its identity and behaviour. A single abstract base class, Robot, defines what every robot in the simulator fundamentally is: an entity that holds a queue of pending instructions, processes one instruction per frame, and maintains a reference to a drive controller responsible for its motion. When a C program connects to the simulator, the robot implements a handshake interface that signals readiness to receive commands. Beyond this, the base class is intentionally minimal: it manages the instruction lifecycle and disconnect behaviour, but contains no knowledge of robot movement, sensing or physical structure.

Three concrete robot types extend this base, each requiring minimal code (18, 20 and 9 lines respectively): *DifferentialDriveRobot* (robot driving with wheels), *SubmarineRobot* (robots with thrusters) and *ManipulatorRobot* (robots with arms for movement), as all functional behaviour is delegated to attached components. This reflects objective 4: new robot capabilities can be added by composing components rather than extending robot classes. This architecture satisfies objective 3 by providing a complete implementation of all three robot archetypes.

Components are self-contained modules that attach to any robot object regardless of its drive type. A *PSDComponent* performs distance sensing, a *CameraComponent* renders the scene from the robot's viewpoint and a *ThrusterComponent* applies propulsion forces scaled to fluid density. Each component encapsulates a single concern and can be attached to any robot configuration without modification. A *CameraComponent* on a submarine behaves identically to one on a wheeled robot because it depends only on its own position and orientation in the scene, not on the robot type it is attached to.

Robot definitions are data-driven rather than hard-coded. A text-based definition file (.robi files) describe a robot's physical structure: wheel geometry, sensor placements, camera parameters and drive type for example, is parsed at runtime by an importer that constructs the robot and attaches the appropriate components automatically. Introducing a new robot configuration requires only a new definition file; no code changes are necessary unless the configuration requires a component type that does not yet exist.

The contrast with V1 is direct and measurable. In V1, the differential drive robot class implements nine distinct responsibilities simultaneously across 588 lines, embedding drive logic, sensor handling, camera management and odometry into a single monolithic class. In V2, the equivalent class is 18 lines, with all capabilities delegated to independent component files.

4.4 XQuartz Porting into Unity

A significant architectural change in EyeSim V2 was the elimination of the XQuartz X11 dependency by reimplementing the robot LCD display entirely within Unity. This eliminates platform-specific installation requirements while ensuring visual and functional replication with the legacy display. This feature implementation reduces the installation process complexity for students as XQuartz is no longer a required component for installation of EyeSim.

In EyeSim V1, the robot's LCD Panel was rendered by an external X11 windowing system entirely outside of Unity. The C library drew text and graphics using X11 system calls, and the LCD window appeared as a separate operating system window independent of the Unity application.

EyeSim V2 eliminates this dependency by moving the LCD display into a Unity UI panel. All RoBIOS LCD functions and the menu button system are transmitted as TCP packets from the C program to Unity, where they are handled by the relevant command factories. The LCD display matches the 480x320 resolution of the physical Eyebot display. Each connected robot receives its own independently spawned LCD window, positioned within the simulator interface with a small cascade offset so multiple panels remain distinguishable.

Reimplementing the X11 display within Unity required replicating the exact visual behaviour of the legacy system in an entirely different rendering environment. The physical Eyebot LCD operates with specific font characteristics and colour handling, and any visual deviation would constitute a backwards compatibility failure as existing lab sheets reference specific pixel positions and display layouts. To achieve this, the C library *lcd.c* was rewritten to remove all X11 system calls and window management code, replacing each display operation with a corresponding TCP packet while preserving identical functional signatures.

These changes are implemented without the C program knowing if it is running on a simulator or the physical Eyebots, highlighting the backwards compatibility of EyeSim V2 even with a major dependency removal. This demonstrates that a complete reimplementation of a display subsystem can eliminate a legacy external dependency without compromising backwards compatibility.

5. Results and Discussion

5.1 RoBIOS API Compatibility

Objective 1 required that all existing RoBIOS C programs, robot definition files, world files, maze files simulator files and object files execute in EyeSim V2 with identical behaviour to EyeSim V1. Verification was conducted across all six supported file formats, comprising 268 supporting files and 95 C programs as detailed in Table 4.1.1. All file formats loaded and parsed correctly, with robot definition files producing physically correct robot configurations, maze files generating the expected wall layout and world files instantiating the correct terrain and object layouts.

RoBIOS API compatibility was verified across all command categories defined in the RoBIOS API reference, provided in Appendix A. Rather than testing each of the 268 example files individually, verification prioritised programs that exercised the broadest range of API commands, covering LCD display output, key input, camera capture, image processing, distance sensing, servo and motor control, V-Omega driving interface, and radio communication. Programs that tested a superset of simpler program behaviour - for example, a program executing a full navigation sequence was considered sufficient verification of the individual drive commands it comprised, were used to maximise coverage efficiently. All RoBIOS API functions were exercised through this approach, and all produced results consistent with expected EyeSim V1 behaviour.

Further testing was conducted by running student lab sheets that required EyeSim. These were from ELEC3020 and AUTO4508: The simulator performed with expected results, which ensures that students using EyeSim V2 in their labs next semester onwards will be able to perform the lab activities without failure.

Compatibility was also verified on MacOS Sonoma, MacOS Sequoia, Windows 10, Windows 11 and Ubuntu 26.04 and Ubuntu 24.04, with no platform-specific failures observed across any tested program or file format, confirming that the elimination of the XQuartz dependency produced a genuine cross-platform simulator consistent with objective 2.

5.2 Networking Benchmark

As EyeSim V2 implemented a new thread for network handling, it is important to test the new approach produced measurable improvements over the original V1 implementation. Even though most of the change was for a component-based architecture and reducing monolithic files, it still should demonstrate differences in performance between client connection rates. The purpose of these tests was not only to check whether individual packets were faster, but also to determine whether the redesigned networking model improved the simulator's reliability and responsiveness under less ideal behaviour.

Two networking tests were selected for comparison. The first test measured normal command round-trip performance, where both simulators were expected to behave similarly if command execution remained limited by Unity's frame loop. The second tested robustness under an incomplete packet scenario, where the client sent only part of a network packet and delayed the rest. This was designed to evaluate whether a slow or faulty client could block the simulator.

Round-trip Command Latency: The first test measured the time taken for a normal *PSDGet* command to be sent from an external client and for the simulator to return a response. Each simulator received 1000 *PSDGet* commands, and the client measured the elapsed time between sending each command and receiving the response. The results are outlined in table 5.2.1:

Version	**Samples**	**Mean latency (ms)**	**Median (ms)**	**Min (ms)**	**Max (ms)**	**Std. Dev (ms)**
V1	1000	4.714	4.718	1.939	13.753	0.980
V2	1000	4.587	4.642	2.473	9.400	0.830

Table 5.2.1: Networking benchmark test 1 results

As shown in Table 1, V1 and V2 performed very similarly for normal command-response behaviour. V2 had a slightly lower (better) latency, improving from 4.714ms to 4.587ms, but this is only a difference of approximately 0.127ms. The median values are also very close between the two versions.

This result suggests that the V2 networking rewrite did not produce a major improvement in ordinary packet round-trip speed. This is expected as although V2 reads packets using a background networking thread, commands are ultimately still being executed through Unity's update loop, which executes one command per frame. This is a fixed Unity feature, and it is very difficult to work around, often requiring external dependencies. However, it does not mean that V2 did not produce any improvement. Unity 6 networking capabilities are already robust, resulting in ~4.59ms of latency for a command to be sent back and forth. This represents a response rate sufficient for real-time interactive use.

Incomplete Packet Robustness: Test two evaluated how each simulator behaved when a client sent an incomplete packet. In this test, one client sent a valid packet header but delayed the body by approximately 2500ms. While this incomplete packet was pending, a second problem client attempted to connect and receive a successful connection message. This tested whether one slow or faulty client could block unrelated networking activity. Table 5.2.2 displays test two results:

Version	**Trials**	**Mean response time (ms)**	**Median (ms)**	**Min (ms)**	**Max (ms)**	**Std. Dev (ms)**
V1	10	2453.25	2440.93	2515.14	2515.29	29.97
V2	10	99.38	99.18	99.07	100.33	0.36

Table 5.2.2: Networking benchmark test 2 results

Table 5.2.2 shows a clear difference between the two networking models. In V1, the probe client was delayed by an average of 2453.25ms, which is almost the full duration of the artificial incomplete packet delay. In V2, the probe client became ready in an average of only 99.38ms. This means V2 responded approximately 24.7 times faster than V1 in the incomplete packet scenario.

The incomplete packet result directly validates the threading model described in Section 4.3. Because V2 processes incoming network data on the dedicated background thread rather than within the main simulation loop, a stalled or faulty client cannot block unrelated simulator activity. V1's single-threaded approach tied network reception to command execution, meaning one delayed client held up the entire system. This test demonstrates that V2's architectural separation of network handling from simulation execution produces a measurably more robust simulator under realistic error conditions, beyond the normal operation improvements shown in the round-trip test.

5.3 Robot Implementations

Objective 3 required all supported robot archetypes of differential drive, submarine and manipulators to be working under the new architectural changes. V2 structures each robot as a hierarchy from robot type through command queue, command objects and drive controller interfaces down to physical components. This makes implementation of robots in V2 cleaner and easier to extend. Table 5.3.1 shows the code files for the drive types of each of the three available robots in V1:

V1 Files	**Lines of Code**	**Purpose**
BaseDiffDrive.cs	588	Differential Drive Robot: Robot identity, motors, servos, VWDrive, Sensors, Camera, Radio and laser
BaseThrusterDrive.cs	520	Submarine Drive Robot: Thruster logic, sensors, servos, camera, radio, and laser
ManipulatorDrive.cs	580	Manipulator Drive Robot: DH parsing/state, servo mapping, gripper, movement, callbacks, visual updates.

Table 5.3.1: V1 file sizes for drive types.

Comparing table 5.3.1 to tables 5.3.2, 5.3.3 and 5.3.4, they show the robot drive type implementations in Version 2 for differential drive, submarines and manipulators respectively.

V2 Differential Drive	**Lines**	**Purpose**
DifferentialDriveRobot.cs	18	Inherit robot class
TwoWheelDifferentialDriveController.cs	185	Two-wheel movement
VWDifferentialDriveController.cs	173	VWCommand parser
WheelComponent.cs	121	Individual wheel control
Total	497	-

Table 5.3.2: V2 differential drive file breakdown

V2 Submarine	Lines	Purpose
SubmarineRobot.cs	20	Inherit robot class
SubmarineDriveController.cs	52	Control sensors and thrusters
ThrusterComponent.cs	98	Control individual thruster
Total	170	-

Table 5.3.3: V2 submarine drive file breakdown

V2 Manipulator	Lines	Purpose
ManipulatorRobot.cs	9	Inherit robot class
ManipulatorDriveController.cs	58	Control sensors and arms
RevoluteComponent.cs	58	Rotating component
PrismaticComponent.cs	69	Rotating + extend arm component
GripperComponent.cs	71	Controls the gripper
Total	265	-

Table 5.3.4: V2 Manipulator drive file breakdown

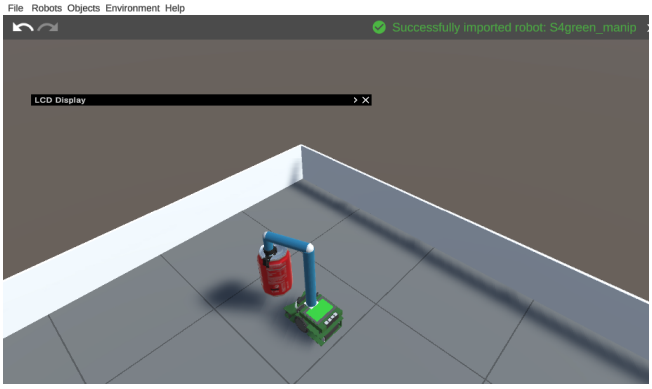
A clear trend is evident as the robot files are separated into distinct components that make up each robot. This ensures easier maintainability, stability and organisation via component-based designing of robots in EyeSim V2 over V1. V2 implements equivalent functionality for implementing EyeSim V1 features which results in fewer lines of code, but more importantly developers need to only work in the files they need to for making their changes instead of one large file for their changes.

Notably, the manipulator implementation demonstrates the consistency advantage of V2s single-cycle development. In V1, the manipulator was added independently by separate contributors working within an existing monolithic codebase. The result was an isolated implementation where servo command handling was embedded directly within ManipulatorDrive.cs alongside arm construction, joint state management, and visual updates: all in a single 580-line class with no separation of concerns. Servo commands for the manipulator were processed through the same 889-line shared routing class used for every other command in the system, meaning a change to manipulator servo behaviour carried regression risks for drive commands, camera commands and sensor commands simultaneously. A developer comment within ManipulatorDrive.cs explicitly acknowledged that the code had become unreadable, providing direct evidence that the architecture had already failed by the assessment of its own author.

In V2, the manipulator follows the same structural pattern as the differential drive and submarine robots. A dedicated servo command factory receives incoming servo commands and resolved them to typed instructions in isolation from all other command types. ManipulatorDriveController dispatches those instructions to the appropriate joint component: RevoluteComponent for rotational joints and PrismaticComponent for linear extension, and GripperComponent for grip control. Each component was implemented as a focused, reusable file. The manipulator is not a special case requiring its own command routing or class hierarchy. It is one application of the same component pattern applied consistently across all robot types in V2, a consistency that was only achievable because the entire simulator was designed and implemented within a single development life cycle with a unified architectural vision from the outset. This reflects established software engineering principles for software design and allows for easier future feature implementations and debugging.

5.4 Extended Features

The component-based architecture of EyeSim V2 enables robot configurations that are structurally impractical in V1. As a concrete demonstration, a hybrid robot combining a differential drive base with a mounted manipulator arm was implemented in V2, controlled by a single C program issuing both drive and servo commands to control either part of the modified robot. This robot type, designated DIFFERENTIAL_MANIPULATOR_DRIVE is shown in Figure 5.4.1.



[Figure 5.4.1: Image showcasing hybrid differential and manipulator robot]

The implementation required one new file of 159 lines and two lines added to the existing drive type registry. No existing robot classes were modified. This is architecturally straightforward in V2 because drive control and arm control are independent components attached to the same robot object: When a drive command arrives, it routes it to the differential drive controller, and when a servo command arrives it routes to the manipulator controller. Neither component is aware of the other, and no shared code path exists between them.

Achieving equivalent behaviour in EyeSim V1 would present a fundamental structural obstacle. V1 represents each robot type as a single class, and C# (programming language in Unity) does not permit a class to inherit from two parents simultaneously. A hybrid robot in

V1 would therefore require either embedding full manipulator joint logic into the existing 588-line differential drive class which carries regression risk for every differential drive robot in the system or creating a new class of approximately 700-900 lines duplicating substantial portions of both existing implementations. Either approach modifies files shared across all robot types, with no isolation between the new feature and existing behaviour. Table 5.4.1 summarises this comparison directly.

This comparison illustrates the central thesis of the project. The value of EyeSim V2 is not confined to feature parity with V1 or cross-platform compatibility. Its primary contribution is an architecture in which new capabilities are additive: composed from existing components without restructuring what already exists. The hybrid robot required no changes to differential drive behaviour, no changes to manipulator behaviour, and no changes to the command routing. In V1 the same feature would have required invasive modifications to foundational shared files. This property, that the system remains open to extension while remaining closed to modification of existing components, is what distinguishes V2 as a maintainable foundation for future development rather than a direct replacement of V1.

5.5 Documentation Updates

EyeSim V2 required corresponding updates to both user-facing and developer-facing documentation to reflect the architectural changes and new features introduced in this project. Rather than producing documentation from scratch, the existing EyeSim V1 documentations served as the foundation. Since the user-facing interface and RoBIOS API remain largely identical between the two versions, most of the existing documentation remained valid with targeted updates to reflect V2-specific behaviour, installation procedures, and new robot configurations. This approach reduced redundancy and preserved the substantial investment already made in V1 documentation.

Developer documentation received more significant attention, as the architectural differences between V1 and V2 are substantial enough that a developer familiar only with V1 would find the V2 codebase unfamiliar without guidance. The documentation was written with the explicit goal of preventing the pattern observed in V1, where independently contributing developers produced inconsistent implementations due to insufficient architectural guidance. Each major subsystem: the component model, the command factory registry, the robot importer pipeline, and the file format specifications are documented with clear explanations of its responsibilities, its boundaries, and the rationale behind its design. Worked examples are provided where the correct approach is non-obvious, ensuring that a developer adding a new robot type or command group can follow a documented pattern rather than inferring one from existing code.

As project manager and sole developer of EyeSim V2 during this project cycle, consistency of documentation was a direct responsibility. Terminology, naming conventions, and structural descriptions are applied uniformly across all subsystems to ensure that documentation of one component provides transferable understanding of others. This consistency directly addresses objective 5 and is intended to lower the onboarding time for

future contributors, reducing the likelihood that subsequent development introduces the architectural drift that accumulated in EyeSim V1 over eight years of multi-developer iteration.

6. Conclusions and Future Work

6.1 Conclusions

EyeSim V2 delivers a functional, cross-platform robotics simulator that meets all five stated project objectives while establishing a demonstrably more maintainable architectural foundation than its predecessor.

Full backwards compatibility was achieved across all six supported file formats and the complete RoBIOS API command surface. Representative programs from the EyeSim V1 example library executed with expected behaviour, and the lab exercises from both ELEC3020 and AUTO4508 were verified across MacOS, Windows and Linux without platform-specific failures. This satisfies objective 1 of full backwards compatibility.

Objective 2 required the elimination of the XQuartz dependency which was achieved by reimplementing the LCD display entirely within Unity, replicating accurate visual and functional behaviour of the X11 display. This removed the necessity of installing XQuartz, a major hurdle in user installation process, and consequently made the UI system consistent across all platforms.

Objectives 3 and 4 stated full robot archetype support within a component-based architecture which highlights the potential of EyeSim V2 use case for future extensions, as V2 achieved these objectives and proved that a component-based architecture allows implementing new and unique robot drive types such as the hybrid differential-manipulator drive example implementation with a streamlined methodology. V2 stripped the monolithic code files in V1 for robot implementations and made it easier for development and modifications. This will aid developers in the future as well due to its flexible system design, making it a powerful simulator for robotics.

Objective 5 was completed with updates to both user-facing and developer-facing materials, written with sufficient clarity and worked examples to guide future contributors without the architectural context that was absent when V1's manipulator feature was added.

Beyond the five objectives, this project contributes a concrete case study in the application of component-based software engineering to an educational robotics simulator. The architectural decisions made in V2 are not specific to EyeSim: the pattern of separating robot identity from robot capability through composable components is a generalisable approach that any simulator, or any program in general facing similar maintainability challenges should adopt.

6.2 Future Work

The following directions are recommended for subsequent contributors:

- Ensure documentation for developers provides sufficient clarity regarding the new EyeSim V2. It should allow even new developers to grasp the mechanics and architecture of EyeSim V2, to prevent the maintainability degradation observer with V1 and become less maintainable in the future. Update the documentation accordingly as necessary.
- Implement a robot-editor feature so users can create robots using pre-existing drive types within EyeSim V2 and use them without requiring a developer to design a *.rob*i text file. This feature will increase flexibility of EyeSim and make it even more robust and user-friendly as text edits of robots will no longer be required.
- JSON implementation for file support: It would allow JSON files to be used alongside *.robi, .maz, .sim* and *.esObj* text files. The benefit would be the automatic checking for syntax errors in JSON files, which does not exist in text versions yet.
- Standard maintenance practices for EyeSim: Routinely resolve errors, bugs or any unexpected behaviours that occurs while using EyeSim V2. Lab facilitators should also remind students to note down any unexpected behaviours so lab facilitators can pass them onto the unit coordinators and developers to resolve them. Students represent a valuable source of feedback as the primary users of the simulator, and as such they will be using EyeSim in ways staff may not and may come across unexpected behaviour.
- Automatic regression testing: Establish a range of tests that run sample programs to test the full capabilities of EyeSim and verifying expected behaviour. This will prevent future changes to EyeSim from silently breaking backwards compatibility.

References

Bräunl, T. (2008). Embedded robotics: Mobile robot design and applications with embedded systems (3rd ed.). Springer, Retrieved May 2026, https://doi.org/10.1007/978-3-540-70534-5

Brugali, D., & Scandurra, P. (2009). Component-based robotic engineering (Part I) IEEE Robotics & Automation Magazine, https://doi.org/10.1109/mra.2009.934837

Koenig, N., & Howard, A. (2004). Design and use paradigms for Gazebo, an open-source multi-robot simulator. Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), https://doi.org/10.1109/IROS.2004.1389727

Michel, O. (2004). Cyberbotics Ltd. Webots: Professional mobile robot simulation. International Journal of Advanced Robotic Systems, https://doi.org/10.5772/5618

Unity Technologies. (2026). Changelog: Unity Physics (6.5.0). Unity Documentation. Retrieved May 26, 2026, https://docs.unity3d.com/Packages/com.unity.physics@6.5/changelog/CHANGELOG.html

University Library. (2026). Library Guides. University of Western Australia. Retrieved May 2026, https://guides.library.uwa.edu.au

Yang, S., & Zhang, Q. (2023). Towards Efficient Robotic Software Development by Reusing Behavior Tree Structures for Task Planning Paradigms. Complex System Modeling and Simulation, https://doi.org/10.23919/csms.2023.0017

Appendices

Appendix A: RoBIOS-7 Library Functions:

LCD Output
LCDPrintf, LCDSetPrintf, LCDClear, LCDSetPos, LCDGetPos, LCDSetColor, LCDSetFont, LCDSetFontSize, LCDSetMode, LCDMenu, LCDMenuI, LCDGetSize, LCDPixel, LCDGetPixel, LCDLine, LCDArea, LCDCircle, LCDImageSize, LCDImageStart, LCDImage, LCDImageGray, LCDImageBinary, LCDRefresh

Key Input
KEYGet, KEYRead, KEYWait, KEYGetXY, KEYReadXY

Camera
CAMInit, CAMRelease, CAMGet, CAMGetGray

Image Processing
IPSetSize, IPReadFile, IPWriteFile, IPWriteFileGray, IPLaplace, IPSobel, IPCol2Gray, IPGray2Col, IPRGB2Col, IPCol2HSI, IPOverlay, IPOverlayGray, IPPRGB2Col, IPPCol2RGB, IPPCol2HSI, IPPRGB2Hue, IPPRGB2HSI

System Functions
OSExecute, OSVersion, OSVersionIO, OSMachineSpeed, OSMachineType, OSMachineName, OSMachineID

Timer
OSWait, OSAttachTimer, OSDetachTimer, OSGetTime, OSGetCount

USB/Serial Communication
SERInit, SERSendChar, SERSend, SERReceiveChar, SERReceive, SERFlush, SERClose

Audio
AUBeep, AUPlay, AUDone, AUMicrophone

Distance Sensors
PSDGet, PSDGetRaw, LIDARGet, LIDARSet

Servos and Motors
SERVOSet, SERVOSetRaw, SERVORange, MOTORDrive, MOTORDriveRaw, MOTORPID, MOTORPIDOff, MOTORSpeed, ENCODERRead, ENCODERReset

V-Omega Driving Interface
VWSetSpeed, VWGetSpeed, VWSetPosition, VWGetPosition, VWStraight, VWTurn, VWCurve, VWDrive, VWRemain, VWDone, VWWait, VWStalled

Digital and Analog I/O
DIGITALSetup, DIGITALRead, DIGITALReadAll, DIGITALWrite, ANALOGRead, ANALOGVoltage, ANALOGRecord, ANALOGTransfer

IR Remote Control
IRTVGet, IRTVRead, IRTVFlush, IRTVGetStatus

Radio Communication
RADIOInit, RADIOGetID, RADIOSend, RADIOReceive, RADIOCheck, RADIOStatus, RADIORelease

Simulation

SIMGetRobot, SIMSetRobot, SIMGetObject, SIMSetObject, SIMGetRobotCount, SIMGetObjectCount