EyeSim-VR Maintenance Manual

Version 1.0

Author: EyeSim-VR Team
Joel FREWIN
Travis POVEY
Ridge SHRUBSALL
Spandana VADDE
Eric ZHANG
Leon ZHIDONG

3 November 2017

Table of Contents

Introduction

The EyeSim program is built using Unity3D, to capitalize on the inbuilt physics engine (NVidia's PhysX) to simulate movement and collisions of the robots and objects. EyeSim allows users to write C or C++ programs using the RoBIOS API. Programs for the real EyeBots can be recompiled, and ran in the simulation without any modification.

EyeSim has been developed using Unity major version 2017, scripted in C# with the Mono backend, .NET 2.0 compatibility. Mono is used for cross-compatibility with Linux, OSX, and Windows. Control code is written in C or C++, and communicates with the simulator over TCP. The EyeSim library replaces the RoBIOS functions with slightly modified versions, which send messages to the simulator. As the control code is a completely independent process to the simulator, they both execute in real-time. Some functionality exists to accelerate the speed of the simulator, allowing faster movement commands.

Software Architecture

Object System

An overview of the object system used in the simulation is presented in Table 1. A class diagram of the inheritance pattern shown in Figure 1. This architecture is designed such that specific managers interact with objects at a specific level of abstraction.

Table 1 Outline of classes in object system

Class	**Type**	**Role**
Monobehaviour	Abstract	Unity's base object. Any script that is attached to a GameObject must inherit from Monobehaviour. Provides integration with Unity's engine (Awake, Start, Update, etc.)
PlaceableObject	Abstract	Base EyeSim class for objects that interact with the mouse. Provides functionality for placing objects in the scene with the mouse, (manages swapping materials to indicate valid/invalid placements)
Robot	Abstract	Provides functionality common to all robots. Maintains a reference to the connection being used to control the robot, a window to view the robot's details
WorldObject	Concrete	Provides functionality common to all world objects. Stores a reference to the inspector window.
ConcreteRobot	Concrete	Concrete implementations of robots. This is currently the RoBIOSDiffDrive, which controls the LabBot and S4. This classes inherit from Robot, and implement interfaces that define what functions they can execute. Responsible for implementing functions such as driving, sensing, cameras etc.

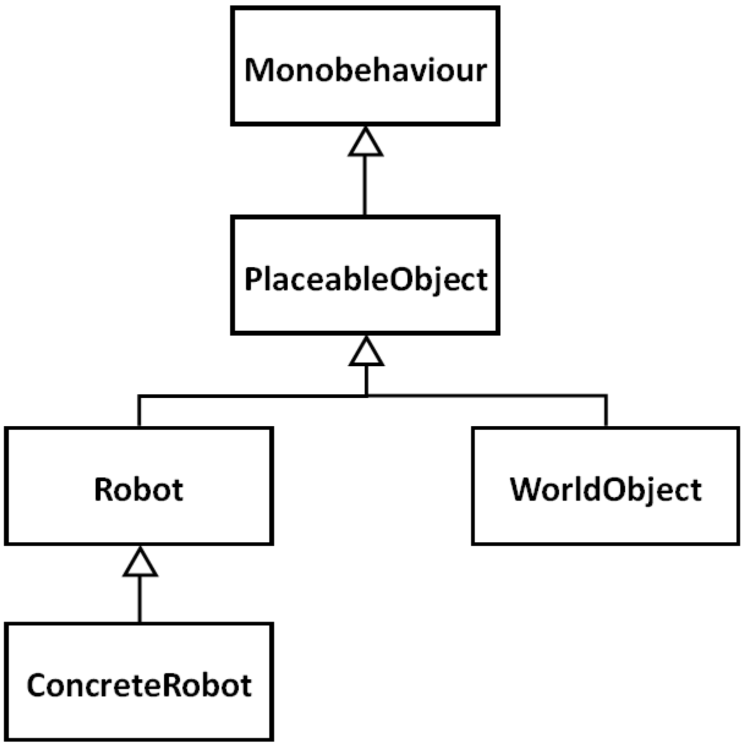


Figure 1 Class diagram showing inheritance of object system

PlaceableObject

The placeable object is the base object that any interactable object inherits from. The main purpose of this class is to provide functionality that handles placement of the object in the scene. This includes:

- Default Vertical Offset: y-value of the object's transform when at rest, in original position
- Vertical Placement Offset: y-value of the objects transform after is has been picked up – value may change depending on position of object (ie can on its side will have lower placement offset than default)
- Placement Booleans: A set of Booleans that define whether the object can be picked up, is placed, and is initialized
- List of Materials: Contains all materials and renderers that are used to render the object. These are modified during placement to indicate valid or invalid placement.
- List of Physical Components: Any component of the object which has a rigidbody or collider. During placement, all rigidbodies are set to kinematic, and colliders are set to triggers. This is to prevent objects that are picked up from interacting physically with the simulation.

This class interacts heavily with the ObjectManager, which handles the movement of objects in the scene with the mouse.

Robot

A robot object is controllable by external code; the base 'Robot' class performs three main functions:

1. Maintain a reference to the connection being used to control it, and begin the disconnect sequence when required.
2. Provide robot-specific functionality for click events (open Robot inspector window when double clicked.
3. Control the visualization features common to all robot types (path trails)

Specific implementations of a robot should implement interfaces that define what it is capable of doing. The provided interfaces are:

Table 2 Interfaces used to define robot capabilities

Interface	**Functionality**
IMotors	Controlling individual motors, and reading encoder values
IPIDUsable	Provide motor control with a PID system
IVWDrive	Provide high-level VW Drive functions (see RoBIOS API)
IServos	Control servos
IPSDSensors	Read values from PSD sensors, and control error
ICameras	Get camera image, and control error
IAudio	Play audio (beeps and small .wav clips)
IRadio	Allow radio communication between robots
ILaser	Read LIDAR scan results

Robots can also be created from .robi files. These robots modify a skeleton robot, which implements the interface ConfigureableRobot. This interface defines functions that modify particular aspects of the robot (wheel diameter, sensor positions, mass etc.) A full list of interface functions can be found in the appendix.

The currently implemented robots (LabBot and S4) use controllers for delegating functionality. All driving functionality is handled by a WheelController, PSD is controlled by the PSDController, and so on. This means modifying a controller will update functionality on both the LabBot and S4, and any other robots that use these classes to control the robots.

World object

World objects are simple objects, such as cans, balls, or boxes, which interact with the simulation physically. WorldObject is a small class that handles the unique on click event for world objects. The shape, size, and mass of the objects is defined in the GameObject itself, rather than in the script.

Objects can be loaded through the ObjectBuilder, which will generate a new world object for use in the simulation. This is covered in-depth in the Loading at Run-Time section.

Implementing an object

Any GameObject (and all its children) should have at most one component that inherits from PlaceableObject, and it should be on the highest level object (ideally at the root level of the GameObject hierarchy). The same GameObject that contains the PlaceableObject, should also contain the main rigidybody and collider components. Children of this GameObject may also contain rigidbodies or colliders. In Awake, a PlaceableObject will automatically generate a list of materials, and physical components used by the GameObject it is attached to, and any children.

In the case of a new object class being created that inherits from PlaceableObject, is important to call the base version of any function that has been overridden; specifically, the Awake function on PlaceableObject must be called for the object to correctly interact with the object manager.

System managers

EyeSim uses a variety of singleton managers to control the simulation. An overview of the managers is shown in Table 3. Each manager is a singleton pattern, which has a public static reference to itself as "instance". These managers are globally accessible by calling Manager.instance.

Table 3 Overview of managers

Manager	**Responsibility**
ApplicationManager	Defines application level functionality – handling setting up on launch, clean up on exit, stores references to the operating system managers
SimManager	Maintains the simulation itself. Stores a list of all robots and world objects in the scene. Adds/Removes robots and world objects, saves and loads sim files, and states. Defines functionality for pausing and resuming simulation.
ServerManager	Handles communication with control programs. Listens for connections, sends and receives packets.
Interpreter	Translates message payloads into robot commands. Receives a packet from ServerManager, along with the connection it is from.
SettingsManager	Stores persistent settings. Contains a dictionary of all settings, loads from disk on start up, and writes them to disk when they are modified.
ObjectManager	Controls the manipulation of objects with the mouse. Handles placing new objects, picking up existing objects, and adding/removing walls.
UIManager	Controls most of the FileBrowsers, and provides callbacks for most of the buttons in the menu system
OSManager	Provides operating system specific functionality, such as opening terminals, and launching additional processes (XMing for Windows)

SimManager

This manager is critical to maintaining the integrity of the simulation. The primary functionality of the SimManager is to store references to the robots and world objects, and to maintain these references. When a new robot is added to the scene, construction of the actual GameObject is done by the ObjectManager. When the object is finished construction, a reference is passed to the SimManager, which then updates its list of objects, and updates any other structures (such as the display to view all robots in the scene).

The SimManager also provides functions to search through the list of objects and find one by its unique ObjectID. Whenever an object is created, it is assigned a unique ObjectID. These are shared between robots and world objects, so that the RoBIOS function SIMSetPosition or SIMGetPosition can be used without reference to the type of object.

The secondary functionality of the SimManager is facilitate saving of the current simulation. There are three save functions:

- SaveSim, which writes the current configuration of objects, robots, and the world, to a .sim file.
- SaveWorld, which writes the current world to a .wld file
- SaveState, which saves the configuration of all objects in local memory, which can then be restored to via RestoreState. Any modifications to the world (such as adding walls, or loading a new world) will invalidate the current state.

The final responsibility of the SimManager is to provide the implementation for pausing and speeding up the simulation. To pause the simulation, all physical objects are set to be kinematic. These means they cannot move, but they retain their current physical properties (inertia). Upon resuming, all objects are changed back to non-kinematic. There are also two delegates provide in SimManager:

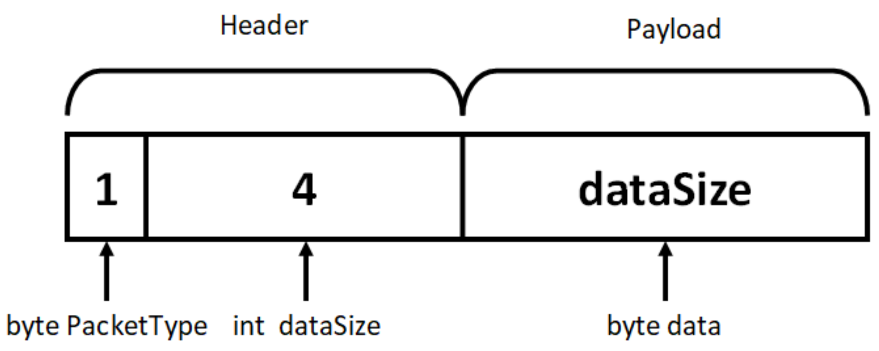
OnPause and OnResume. Any class that requires specific functionality on pause or resume should provide a function to be called, and register it with these delegates.

ServerManager and Interpreter

The ServerManager handles communication with the control code. A TCP listener is created on start-up, which listens for connections on the specified port (default 34721). Incoming connections first undergo a handshake, where the server ensures that it is communicating with a RoBIOS program, then it indicates it is ready once the connection has been assigned to a robot. If there is no robot in the scene, the server manager will reject the connection.

Connections are maintained using the RobotConnection object, which contains a reference to a TcpClient (an open connection to a control program), and a Robot (the corresponding robot executing the control). The ServerManager maintains a list of RobotConnections, and checks each one periodically to see if the TcpClient has a message waiting.

Communications between the server and the clients are done using packets, which have the following structure:



The header is read first, to determine the data size, and then dataSize bytes are read into a byte array. The packet type is used to determine what operation to perform with the incoming packet. The packet types are:

Table 4 Packet types

Origin	PacketType	Purpose
Control program	CLIENT_HANDSHAKE	Initiate the handshake process
	CLIENT_MESSAGE	Send a command to a robot
	CLIENT_DISCONNECT	Disconnect from the simulation
Simulator	SERVER_HANDSHAKE	Respond to the handshake
	SERVER_READY	Inform client ready for execution
	SERVER_MESSAGE	Send a message to client (eg PSD Sensor value)
	SERVER_DISCONNECT	Disconnect client

Most communication is done via messages. Messages from the client as used to control the robots. These are a single character, followed by a set of values. These messages are deserialized by the interpreter, which calls the corresponding function on the robot. These messages are binary, and use different data types for different commands.

An example of a command is VWStraight(500, 200), in plain text would be “y 500 200”, which is serialized as:

```
0x0C01F400C8
```

This is 1 byte for y in asci, 4 bytes for 500, and 4 bytes for 200. A full list of the commands can be found in the appendix. Values received in messages are assumed to be big-endian, and as such must be converted to the endianness of the simulator (likely little-endian).

It is possible to bypass the interpreter; if a new robot that uses a different control scheme is implemented, it would be possible to utilize the existing server architecture, with a different interpreter implementation, and forward messages to the appropriate one via some check (interfaces or type inference).

SettingsManager

The settings manager stores all the persistent settings. These are read from a file when the simulator first starts, and writes to this same file when the settings are modified. This process is handled by Unity's PlayerPrefs class, which handles writing persistent settings for all operating systems.

Settings are stored in a dictionary based on their type. Each setting is a key value pair, which is a short string describing what the setting is as the key, along with the actual value. Accessing the settings is done through the functions ChangeSettingsValue and GetSettings. These functions are overloaded for each data type of setting. Accessing the dictionary is done through the following lambda function:

```
(x, y) => y ? settingsVariable = x : settingsVariable
```

The y variable defines whether the setting should be updated or retrieved; if true, the settingsVariable is set to the value x, if false, the settingsVariable is returned. Modifying a settings value can then be done via:

```
settingsDictionary[“settingName”](newValue, true)
```

The value of a particular setting can be retrieved with:

```
toSet = settingsDictionary[“settingName”](0, false)
```

The purpose of using this dictionary is that settings can be quickly added by creating a new settings name (string descriptor), assigning it a variable in memory, and then adding this to the correct dictionary. This provides a centralized access point for all persistent settings.

ObjectManager

The object manager handles interactions of the mouse with the scene, and the creation of prefabricated objects. Adding objects to the scene is done with the function

```
AddPredefinedObjectToScene(string type, string args)
```

This takes a known type of object, instantiates a copy of it from a prefabrication, and if nothing is passed to the `args` parameter, it will attach to the mouse, else it will use `args` to place the object in the scene at a specific location. Several single input functions are used as callbacks to this with specific types, for integration with Unity's UI system (which can only call functions with a single input parameter). The predefined objects currently are:

- LabBot (Robot)
- S4 (Robot)
- Can (WorldObject)
- Soccer (WorldObject)
- Crate (WorldObjecT)

The object manager is also capable of creating custom objects loaded from .esObj files. More information about the loading process can be found in the Loading at Run-Time section. When a new custom object is loaded, a copy is created and held in “limbo”, so that it can be cloned. A new entry is created in the Add Objects menu, which allows the custom object to be created and placed in the scene the same as any predefined object.

The second function of the object manager is to handle interactions with the mouse. There are currently four states the mouse can be in:

- Free
- Holding Object
- Placing Wall
- Removing Wall

The mouse can only be moved from the state Free to a non-Free state, or vice versa.

The state Holding Object refers to any time an object is on the mouse, either after being initially created, or an existing object being placed. A raycast is done from through the mouse to the ground, to determine where the object should be. If the object's collider is in contact with another object, it will be highlighted red, indicating an invalid placement, else it will be green, indicating valid placement. If the user clicks whilst the object is green, the object will be placed in the scene.

Placing and Removing walls allows users to modify the environment in the simulation. Walls can only be placed along the ground plane. The placing wall process is begun when the user selects "Add Wall" from the menu, after which the first click will set the start position of the wall, and the second click will set the end position and construct the wall.

UIManager

The UI manager stores a reference to all the persistent windows (non inspector), and has callbacks for most of the menu buttons. It also contains references to most of the FileBrowsers, and handles blocking interaction with the simulator when a file browser is open. Due to the limitations of Unity's event system, which drives the user interface (can only call functions with a single input parameter, of a primitive type), several small functions are required in the UI manager to translate between UI interactions, and the corresponding action to take.

OSManager

The operating system manager is an abstract class that is implemented for Windows and OSX. It provides functionality specific to the operating systems. This includes opening a terminal (cygwin on Windows, default terminal on OSX), and launching extra processes (on Windows, it manually launches the XMing XWindows client, to allow control programs to display their LCD screens). Any functionality that is operating system specific should be added to implementations of this class.

User Interface

The user interface has two components: the menu bar system (located at the top of the screen), and a window system.

Menu

The menu is broken down into 5 categories:

- File - contains options for opening the terminal, interacting with the local file system (saving/loading), manipulating the world (create/reset), changing the settings, and exiting.
- Simulator - contains buttons for adding objects, viewing existing objects, saving/loading the simulator state, and pausing/resuming the simulation.
- Camera – contains buttons for changing between bird's eye view (Orthographic), and regular perspective camera
- Environment – contains buttons to add / remove walls
- Help – contains links to RoBIOS API, user manual, and opening the About window.

The menu bar also contains quick buttons for pausing, resuming, and speeding the simulation up to double speed. These execute the same functions that are called by the menu items Pause/Resume under Simlulation.

There are 3 classes that are used in the menu system:

- MenuBarManager: Contains a reference to the currently active submenu, and the colours used. Also is responsible for adding custom objects to the Add Objects submenu.
- MenuBarButton: A top level menu button (File, Simulator etc.). Only a single menu item can be active a time. Uses a static reference to the current open menu item to enforce this.
- MenuBarItem: Actual items in the menus. Can either be a clickable button that triggers some function, or can open another submenu. Visually, a caret pointing to the right indicates a submenu button.

When a MenuBarItem is being used as a simple button, it can trigger a UnityEvent callback. These can be assigned in the editor (the same callback used by Unity's default UI Button). If an item opens a submenu, the Boolean hasSubmenu should be set to true, and the submenu GameObject must be set to the actual submenu object (container for other menu items). Each item inside the submenu must have the Boolean isSubmenuItem set to true, and must contain a reference to the MenuBarItem that opens the submenu. This is required to correctly manage allowing only a single submenu to be open at a time, and to properly close the menu when a submenu item is pressed.

Adding a new menu can be done by creating a new GameObject under the MenuBar object in the Canvas. This new object must have a MenuBarButton component attached. Items in the menu are placed into a DropDownMenu object, which is used to control the visualization of the items, using a Vertical Layout Group and a Content Size Fitter. MenuBarItems are placed as children of the DropDownMenu object, and will automatically be sized and placed in the appropriate location.

Windows

The windowing system in EyeSim uses a class Window, which has specific implementations for all the different windows. Window is not an abstract class, so in the case of very simple windows, it is sufficient to simply place it on the window GameObject as a component.

The Window class defines the Open and Close function, which very simply sets the active state of the GameObject the window is attached to, and in the case of Open, also brings the window to the front. This class also prevents camera zoom when the mouse is over an open window, which prevents the mouse wheel from zooming the camera when using it to scroll up and down on a scrollable window.

The TabWindow is an abstract class that inherits from Window, and provides the skeleton code for a window that can several layers of content, which can be toggled between using tabs. This is useful for windows where there is too much information or content for a single layer.

A window GameObject consists of two parts: the content portion, and a header bar. The header bar has a DragPanel object attached, which allows the user to move the window around the interface, and also contains the button to close the window.

The currently implemented windows are:

Type	Window Name	Purpose
Persistent	View Robots	View a list of all robots in the scene. Clickable buttons to open the inspector window of individual robots
	View Objects	View a list of all objects in the scene. Clickable buttons to open the inspector window of individual objects
	Create World	Create an empty box world, user inputs width and height.
	Colour Picker	Select the colour of a marker. Triggered when the "Select" option for the colour of a marker is pressed
	Settings	View and modify the current persistent settings, such as camera sensitivity, home directory, and error
	Log	View the simulator log.
	About	Display some information about EyeSim
Temporary	Inspect Robot	View and edit information about a particular robot. Is first created when first opened. Destroyed when the corresponding robot is destroyed
	Inspect Object	View and edit information about a particular object.
	Inspect Marker	View and edit a marker.

New windows can be created by implementing a new class that inherits from Window or TabWindow. The top level GameObject of the window consists of only the Rect Transform, which defines the total size of the window, and the implemented Window class. The children of the main window game object consist of the header bar, with a DragPanel attached, and the main content, which is fully customizable.

There are two containers for objects – the GameWindowContainer, and the FrontWindowContainer. GameWindowContainer holds all window objects that should interactable along side the simulator, such as inspector windows, whilst the FrontWindowContainer prevents any interaction with the simulator whilst it a window inside it is open, such as the file browser windows.

Keyboard Shortcuts

Keyboard shortcuts fall under two categories: Input Axis (Unity's default input management), and hardcoded KeyCode input. All camera movement is handled with the Input Axis type of keybinding. These provide a single axis, identified by a string, and two associated buttons (positive and negative). Due to limitations in Unity, the associated buttons cannot be changed at run time (via script), instead, they can only be changed by the user in the prelaunch dialogue.

Some functions are triggered by hard-coded KeyCode values. These are the deletion of an object on the mouse (via Escape or Delete), and resetting the simulation to the last saved state (Control + Shift + R). The advantage of using KeyCode based inputs, is that the particular key that is checked can be changed at run-time with a script (through the use of some keybindings settings), although this is not currently implemented.

Logging

Messages from the simulator are logged via the EyesimLogger, which can write to a window (LogWindow), and to a text file. Log messages are sent to the EyesimLogger with the command

```
EyesimLogger.instance.Log(string message);
```

This will add an entry to the internal log, and if a log file has been previously opened with CreateNewLogFile, it will also write to this file. Log messages are prefixed by the time since the application was launched in [hh:mm:ss] format.

Run-time functionality

EyeSim facilitates loading objects, robots, and worlds at run-time, to allow users to create their own custom simulations without needing to edit and recompile the source. Files loaded from disk are plain text, with an extension that indicates the type of object or environment to load in. The loadable files are outlined below in Table 5.

Table 5 Files loadable from disk

Extension	Type	Loader
.robi	Robot	RobotLoader
.esObj	Object	ObjectLoader
.wld	World	WorldBuilder
.maz	Maze	WorldBuilder
.sim	Simulation	SimReader

Each object type has a specific class dedicated to reading and interpreting files of that type. All these classes utilize a common underlying object for disk IO. Performance of these classes is not thoroughly optimized, as this is not a performance critical operation.

RobotLoader

The RobotLoader constructs a new robot object from a .robi file. The current implementation uses a skeleton bot of a specific type to begin this process. The first non-comment line a .robi file should be the type of robot (differential drive, Ackermann drive, or Omnidirectional drive). A prefab of each of these types of robot exists, which contains the building blocks required for the rest of the process. Subsequent lines in the .robi file will call configuration functions on the new created object, which are defined in the skeleton class.

A customizable robot implements the interface ConfigurableRobot, which declares several configuration functions, such as ConfigureWheels, ConfigureSize, and so on (see interface definition for full list). The RobotLoader will read each line in the .robi file, verify that the input arguments are valid, and then call the corresponding configuration function. If there is an error in the input file, the RobotLoader will report that it has failed, and abort the process (deleting the associated GameObject). Upon successfully loading a robot, a new entry is added to the Add Robot menu, which allows the user to instantiate a copy of the custom robot.

For a full list of the configurable properties of a custom robot, see the user manual.

ObjectLoader

The ObjectLoader functions very similarly to the RobotLoader, but with a significantly reduced set of configurable properties. There are no skeleton objects, as a WorldObject does not have any significant functionality. Instead, only the model, colliders, and mass can be specified in the .esObj file. The colliders that can be used are limited to sphere, capsule, and box, to allow for optimization by the physics engine. There is currently no support for mesh colliders, as these have a significant performance cost associated. An outline of how a .esObj file is structured can be found in the user manual.

WorldBuilder

The WorldBuilder is responsible for reading .wld and .maz files, as well as constructing any other world required. When processing world or maze files, the WorldBuilder will attempt to construct the entire world, and if successful, will delete the existing environment and supply the SimManager with the newly created one. If it fails, it will report the failure and abort the process. A .wld file contains the dimensions of the floor, and walls, in x – y coordinates, and an optional path to a PNG file to use as a floor texture. A .maz file is an ASCII representation of what the maze looks like (using | and _ to represent walls).

SimReader

The SimReader is used to reload saved simulation files. A .sim file is a configuration file that specifies the environment, and the position of the objects and robots in the scene. These are an easy and fast way to start a specific simulation. If the SimReader encounters an error whilst loading a .sim file, it will delete the entire scene and create an empty box.

Loading files

When an external file is loaded, it is opened in an IO object, which wraps the standard C# stream reader, and provides some extra functionality. This includes extracting the next set of arguments from the file (next non-comment line, with arguments separated by white space, and possibly encapsulated by quotes), and searching for a file given a relative or absolute path.

Robi, esObj, and sim files can all contain paths to other files that need to be loaded, such as object models, or world files, and the search path is relative to the original loaded file. The IO class provides a function to locate a file provided as an argument. The priority of search paths are:

1. Absolute path
2. Relative path from location of directory file loaded into IO
3. Relative path from a supplied root directory, provided in code
4. Relative path from the home directory

Errors that occur during the file loading process (ie unable to find a specified file) are reported via the Logger. The default behaviour for what to do given a failure is coded into the individual loading classes, the IO class itself doesn’t directly report any errors.

Virtual Reality

EyeSim supports virtual reality through the Oculus Rift, allowing the user to move around the scene as a virtual reality environment, or to take the perspective of one of the robot's cameras. This is implemented in the VRControl class, which handles enabling and disabling the virtual reality components, and moving the virtual reality camera.

When in virtual reality mode, the user is placed inside the scene attached to a CharacterController (a Unity component). Movement of the CharacterController is done through the Vertical and Horizontal input axes, by default mapped to the arrow keys. If a controller is attached, the joystick can also be used to control movement. Rotating and looking around the scene is done through movement and rotation of the virtual reality headset itself.

Switching between normal and virtual reality cameras is done through a key press (currently B to go to normal camera, N to go to virtual reality), and taking the perspective of a robot is done through the robot's inspector window, under the Control tab. The implementation is very basic, there is no interaction with the user interface whilst in VR mode (no placing or moving objects, or interacting with the menu system).

RoBIOS Compatibility Library

RoBIOS functionality is implemented in a library that is based on the original API used by the real Eyebots. The library serializes commands into a message, which is sent through a TCP connection to the simulation server.

Connecting to simulator

Any call to a RoBIOS function will first check if the connection has been established to the simulation server. Every function begins with a call to SIMInit(), which checks the variable SIMInitialized to determine whether or not the connection is made. If not, the program is connected to the simulator.

The conn.c file in the library code contains all the functions required to maintain, and utilize the connection. Each RoBIOS function that communicates with the simulation server creates a message, consisting of a single character identifier encoded in ASCII, and numerical parameters, serialised to an array of bytes in big endian form. This message is passed to the function SERSend, which calls write_packet to send the final packet. The appendix contains the full list of functions that use this method.

Several functions in the RoBIOS API do not communicate with the simulation server. These are the LCD output, key reading, image processing, and two camera functions. The execution of these functions are purely client side.

LCD Display

The LCD display system utilizes the same code as the original Eyebot system. Since this system uses the X11 library to render the display, it requires an X Windows server to be running on the client. Most linux distributions ship with X11 functionality inbuilt, so no extra software is required. For OSX, XQuartz is required for both the display, and for Microsoft Windows, XMing is used. The X11 developer libraries are required by users to compile their RoBIOS programs with the Eyesim library.

User Programs

Users of the simulation program develop their own control code to execute on the robots. The library is built on POSIX compliant framework, and as such will compile on Linux and OSX without issue. For Microsoft Windows, cygwin is required. Cygwin provides the compilers for POSIX compliant code (gcc), and an execution platform for the programs.

Extending to other languages

As the communication between the simulator and client programs is done via standard TCP connections, it is possible to write RoBIOS programs in any language that can interface with a C library.

Appendix

RoBIOS API Functions implemented in simulator

Function Name	Char	Arguments	Return
Camera Functions			
CAMInit	F	Width | Height	None
CAMGet	f	None	Cam Image
Audio Functions			
AUBeep	b	None	None
Sensor Functions			
PSDGet	P	PSD ID	PSD Value
LIDARGet	l	None	Laser scan results
Servo Functions			
SERVOSet	s	Servo ID	None
SERVORange	S	Servo ID | Min | Max	None
Motor Functions			
MOTORDrive	m	Motor ID | Speed	None
MOTORDriveRaw	m	Calls MOTORDrive	None
MOTORSpeed	M	Motor ID | Speed	None
ENCODERRead	e	Uint8: encoder ID	Encoder value
ENCODERReset	e	Calls ENCODERRead	None
VW Functions			
VWSetSpeed	x	Linear Speed | Angular Speed	None
VWGetSpeed	X	None	Linear Speed | Angular Speed
VWSetPosition	Q	x | y | phi	None
VWGetPosition	q	None	x | y | phi
VWStraight	y	Distance | Linear Speed	None
VWTurn	Y	Angle | Angular Speed	None
VWCurve	C	Distance | Angle | Linear Speed	None
VWRemain	z	None	Distance Remaining
VWDone	Z	None	Is Driving
VWWait	L	None	True on drive complete
VWStalled	Z	None	True on drive stalled
Radio Functions			
RADIOGetID	i	None	Robot ID
RADIOSend	R	Robot ID | Message	None
RADIOReceive	r	None	Robot ID | Message
RADIOCheck	c	None	True if message waiting
RADIOStatus	I	None	All Robot IDs
Sim Functions			
SIMGetPose	1	None	x | y | phi
SIMSetPose	2	x | y | phi	None
SIMGetObject	3	Object ID	x | y | phi
SIMSetObject	4	Object ID | x | y | phi	None

LCD Output, Keys, Image Processing functions are implemented on the EyeSim library side (client code). See http://robotics.ee.uwa.edu.au/eyebot7/Robios7.html for full list of functions.

```
int CAMInit(int resolution)
int CAMRelease(void)
int CAMGet(BYTE *buf)
int CAMGetGray(BYTE *buf)

int AUBeep(void)

int PSDGet(int psd)
int LIDARGet(int distance[])

int SERVOSet(int servo, int angle)
int SERVORange(int servo, int low, int hight)

int MOTORDrive(int motor, int speed)
int MOTORDriveRaw(int motor, int speed)
int MOTORSpeed(int motor, int ticks)
int ENCODERRead(int quad)
int ENCODERReset(int quad)

int VWSetSpeeD(int linSpeed, int angSpeed)
int VWGetSpeed(int *linSpeed, int *angSpeed)
int VWSetPosition(int x, int y, int phi)
int VWGetPosition(int *x, int *y, int *phi)
int VWStraight(int dist, int lin_speed)
int VWTurn(int angle, int ang_speed)
int VWCurve(int dist, int angle, int lin_speed)
int VWRemain(void)
int VWDone(void)
int VWWait(void)
int VWStalled(void)

int RADIOInit(void)
int RADIOGetID(void)

int RADIOSend(int id, char* buf)

int RADIOReceive(int *id_no, char* buf, int size)

int RADIOCheck(void)

int RADIOStatus(int IDlist[])

int RADIORelease(void)


void SIMGetPose(int *x, int *y, int *phi)

void SIMSetPose(int x, int y, int phi)

void SIMGetObject(int id, int *x, int *y, int *phi)

void SIMSetObject(int id, int x, int y, int phi)
```

Robot Control Interfaces

```
// Control invidiual motors at low level
public interface IMotors
{
    void DriveMotor(int motor, int speed);
    int GetEncoder(int quad);
}

// Set PID Controller values for a motor, and drive using
public interface IPIDUsable
{
    void DriveMotorControlled(int motor, int ticks);
    void SetPID(int motor, int p, int i, int d);
}

// VW Drive interface for RoBIOS commands
public interface IVWDrive
{
    // Initalize VW Parameters (mostly unused)
    void InitalizeVW(int[] args);
    // Get robots internal position
    Int16[] GetPose();
    // Set robots internal position
    void SetPose(int x, int y, int phi);
    // Set vehicle speed manually
    void VWSetVehicleSpeed(int linear, int angular);
    // Get current speed
    Speed VWGetVehicleSpeed();
    // Drive a straight line
    void VWDriveStraight(int distance, int speed);
    // Turn on the spot
    void VWDriveTurn(int rotation, int velocity);
    // Drive an arc of a circle
    void VWDriveCurve(int distance, int rotation , int velocity);
    // Return remaining distance to drive
    int VWDriveRemaining();
    // Return whether or not a controlled drive is being executed
    bool VWDriveDone();
    // Return whether or not a motor has stalled
    int VWDriveStalled();
    // Send a reply when the current drive has finished
    void VWDriveWait(Action<RobotConnection> doneCallback);
    // Clear any current VWWait command (used when control is terminated whilst a
VWWait is pending)
    void ClearVWWait();
    // Use accurate positioning
    bool VWAccurate { get; set; }
}

// Controling mechanical servos
public interface IServos
{
    void SetServo(int servo, int angle);
}
```

```
// Using position sensitive devices
public interface IPSDSensors
{
    UInt16 GetPSD(int psd);
    float MeanError { get; set; }
    float StdDevError { get; set; }
    bool UseError { get; set; }
    bool UseGlobalError { get; set; }
    void SetVisualize(bool val);
}

// Using cameras
public interface ICameras
{
    byte[] GetCameraOutput(int camera);
    void SetCameraResolution(int camera, int width, int height);
    string GetCameraResolution(int camera);
    EyeCamera GetCameraComponent(int camera);
    // Salt and Pepper noise parameters
    bool SaltPepperNoise { get; set; }
    // Salt and Pepper noise % of pixels to modify (average)
    float SPPixelPercent { get; set; }
    // Salt and Pepper ratio of black to white pixels (average)
    float SPBWRatio { get; set; }
    // Gaussian noise parameters
    bool GaussianNoise { get; set; }
    float GaussMean { get; set; }
    float GaussStdDev { get; set; }

}

// Playing audio (note AUClip is handled entirely in the executing control program)
public interface IAudio
{
    void PlayBeep();
}

// Sending/Receiving radio messages
public interface IRadio
{
    void AddMessageToBuffer(byte[] msg);
    byte[] RetrieveMessageFromBuffer();
    void WaitForRadioMessage(Action<RobotConnection, byte[]> radioDelegate);
    int GetNumberOfMessages();
}

// using LIDAR Scanner
public interface ILaser
{
    int[] LaserScan();
    void SetVisualize(bool val);
}
```