NETWORK-BASED OPERATOR INTERFACE FOR AN UNDERWATER ROBOT

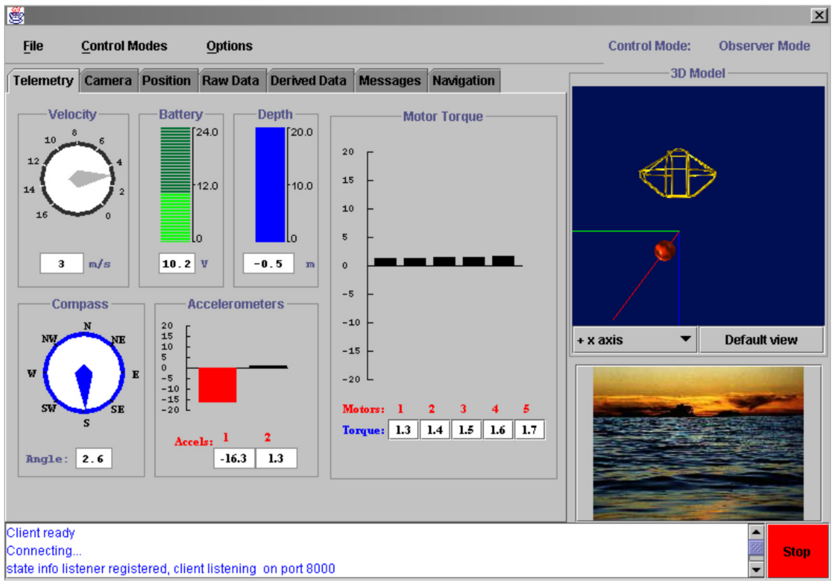
by
Chris McPherson (3028065)

A thesis submitted in partial fulfilment of
the requirements for the degree of

Bachelor of Engineering

Australian National University

1999



Control Mode: Observer Mode

```
Client ready
Connecting...
state info listener registered, client listening  on port 8000
```

SUPERVISORS: SAMER ABDALLAH

DAVID WETTERGREEN

ALEX ZELINSKY

ACKNOWLEDGEMENTS

The author wishes to thank Samer Abdallah and David Wettergreen for their continuous assistance throughout this project. Thanks also to Chris Gaskett for his help with my understanding of Java and other aspects of the interface, and to Harley Truong for developing the 3D model for my use. Finally, I'd like to thank all those involved with the Kambara robotics group at RSISE.

ABSTRACT

The aim of this project is to design advanced operator interfaces for underwater robots and to develop a specific instance for ANU's vehicle, Kambara. The interfaces receive state information from the robots and present it to the operator in intuitive graphical representations, while receiving commands from the operator.

The motivation for this project stems from the fact that underwater robots require adequate guidance and control to perform useful tasks. The eventual goal is to enable a user to command an underwater robot to hold station on a reef or swim along a pipe, and to have that user observe the results via real-time updates of a GUI.

This project has developed a system which uses Java 2 and it's communication method RMI to provide cross-network capabilities. This allows multiple users to view the state of the robot, while having only one user control it. Specifically, the cross-network capabilities allow updates of the robot's state information at a rate of 10 Hz if the client receives packets directly, but only 4 Hz if a server is present in the system. State information is transmitted using UDP packets, whereas control requests make use of RMI methods. A basic control system is implemented, allowing a point-and-click method of operation.

Additionally, a three dimensional model of the robot was created using Java 3D, giving the user an intuitive sense of position.

TABLE OF CONTENTS

GLOSSARY i

LIST OF FIGURES ii

1. INTRODUCTION 1

1.1. Aim 2

1.2. Motivation 2

1.3. Scope 2
1.3.1. Software Goals 3
1.3.2. Java and the use of Java 3D 3

1.4 Thesis Outline 4

2. RELATED WORK AND HUMAN-COMPUTER INTERACTION 6

2.1 Related Work 6
2.1.1. Comparison to AUV interfaces 6
2.1.2. Comparison to general interfaces 6
2.1.2.1. Other Robotic or Simulator Interfaces 6
2.1.2.2. Game and Dashboard Interfaces 8

2.2 Human-Computer Interaction 9
2.2.1. Reducing the response time 9
2.2.2. Making the interface easy to read and follow 9
2.2.3. Resulting considerations 10
2.2.4. Basic design principles 11
2.2.4.1. Space 11
2.2.4.2. Alignment 11
2.2.4.3. Proximity 11
2.2.4.4. Contrast 12

3. PROGRAMMING LANGUAGE SELECTION AND THE SOFTWARE DEVELOPMENT LIFE CYCLE 13

3.1 Programming Language Selection 13
3.1.1. Comparison of Programming Languages 13
3.1.2. Java's Advantages 14
3.1.3. Java's Primary Disadvantage 14
3.1.4. Summary of Java's Selection 15

3.2 Software Development Life Cycle (SDLC) 15

4. ANALYSIS AND DESIGN 17

4.1 Analysis 17
4.1.1. Definition of State Information 17
4.1.2. Planning for Control Modes 18
4.1.2.1. Observer Mode 19
4.1.2.2. Individual Control Mode 19
4.1.2.3. Teleoperation Control Mode 19
4.1.2.4. Supervisory Control Mode 19
4.1.3. Analysis Summary 19

4.2 Design 20
4.2.1. Network Architecture 20
4.2.1.1. Long-term Goal for Network Design 20
4.2.1.2. Short-term Goal for Network Design 21
Server 22
Client 24
Flow of Data Within The System 26
4.2.2. Software Architecture 27
4.2.2.1. Software Architecture of the Kambara Client 27
Overview 27
Network details (KambaraClient) 27
GUI components (KambaraMain) 28
Thread component (OutputKambaraData) 30

4.3. Conclusion 30

5. IMPLEMENTATION 31

5.1. Presenting State Information 31
5.1.1. Division in to Tabbed Panes 31
5.1.2. GUI Layout 32

5.2. Cross-network capabilities 32
5.2.1. Socket Connections in Java 2 32
5.2.2. Using RMI on Windows NT 33

5.3. Implementation of Java 3D Model 34
5.3.1. The Evolution of the 3D Model 34
5.3.2. Applying the 3D Model 35
5.3.3. Co-ordinate System Problems 37

5.4. Control Modes 38

5.5. Video Feed 39

6. VALIDATION AND VERIFICATION 40

6.1. Testing procedure 40

6.2. **Basic design principles** 41

6.3. **State information updating** 41

6.4. **Network testing** 42

6.5. **Control mode testing** 42

7. CONCLUSION AND FURTHER WORK 44

7.1. **Conclusion** 44

7.2. **Further Work** 45
7.2.1. Extending the Control Modes 45
7.2.2. Implementing long-term network architecture 46
7.2.3. Logging information 46
7.2.4. GUI improvements 47
7.2.5. 3D model improvements 47
7.2.6. Video feed implementation 47
7.2.7. Client implementation 48
7.2.8. Server implementation 48

BIBLIOGRAPHY 49

Appendix A: Remote Method Invocation (RMI) 51

Appendix B: Vector of data transmitted by simulator 55

Appendix C1: Screenshot of telemetry paneL 56

Appendix C2: Screenshot of CAMERA paneL 57

Appendix C3: Screenshot of POSITION paneL 58

Appendix C4: Screenshot of RAW DATA paneL 59

Appendix C5: Screenshot of DERIVED DATA paneL 60

Appendix C6: Screenshot of MESSAGES paneL 61

Appendix C7: Screenshot of NAVIGATION paneL 62

GLOSSARY

ANU Australian National University

API Application Programming Interface

AUV Autonomous Underwater Vehicle

GUI Graphical User Interface

JDK Java Developer's Kit

JRE Java Runtime Environment

Kambara ANU's Underwater vehicle, named after the Aboriginal word for crocodile.

Object An object is a self-contained element of a computer program that represents a related group of features and is designed to accomplish specific tasks.

RMI Remote Method Invocation

RMIC Remote Method Invocation Compiler

TCP/IP Transmission Control Protocol/ Internet Protocol. A connection-oriented approach to communicating between two entities.

UDP User Datagram Protocol. A connectionless approach to transporting packets between entities.

URL Universal Resource Locater

VM Virtual Machine

LIST OF FIGURES

Figure 1: Kambara. 1
Figure 2: Screenshot of the Kambara interface. 2
Figure 3: Host software for the ROV named REMUS. 7
Figure 4: State Information for the robot Nomad. 8
Figure 5: Interface for the submarine simulator game 688 Hunter/Killer. 8
Figure 6: Overview of Software Development Life Cycle. 16
Figure 7: Long-term Network Architecture for the Kambara System. 20
Figure 8: Short-term Network Architecture for Kambara System. 21
Figure 9: State Information Distributor. 22
Figure 10: Typical Flow of Data in Kambara's system. 26
Figure 11: Software Architecture for the client side Kambara GUI. 29
Figure 12: Early prototype of 3D model. 34
Figure 13: Intermediate 3D model of Kambara. 35
Figure 14: Scene Graph hierarchy for Kambara's Java 3D System. 36
Figure 15: Default 3D orientation. 37
Figure 16: Desired 3D orientation. 37
Figure 17: RMI layers. 52
Figure 18: The flow of control from the client to the server and back. 53

Chapter 1

1. INTRODUCTION

At the Australian National University, an autonomous underwater vehicle, Kambara, is being developed for tasks in exploration and inspection. The objectives are to enable submersible robots to autonomously search in regular patterns, to follow along fixed natural and artificial features, and ultimately to swim after dynamic targets. These capabilities are essential to tasks like cataloguing reefs, exploring geologic features, and studying marine creatures, as well as inspecting pipes and cables, and assisting divers.

The underwater vehicle is named Kambara, an Australian Aboriginal word for crocodile. Kambara's mechanical structure (shown in Figure 1) was designed and fabricated by the University of Sydney. It is a simple, low-cost underwater vehicle suitable as a test-bed for research in underwater robot autonomy. At the Australian National University, the task has been undertaken to equip Kambara with power, electronics, computing and sensing [1]

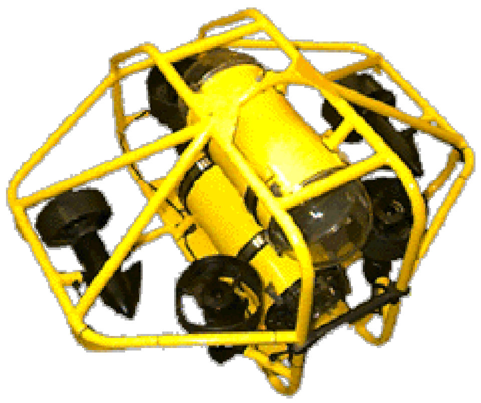


Figure 1: Kambara.

During the initial testing stages, an optical fibre tether is in place, running from the vehicle to the system controlling Kambara. It is desired that in the long-term, this tether is removed to allow for remote, autonomous activity.

1.1. Aim

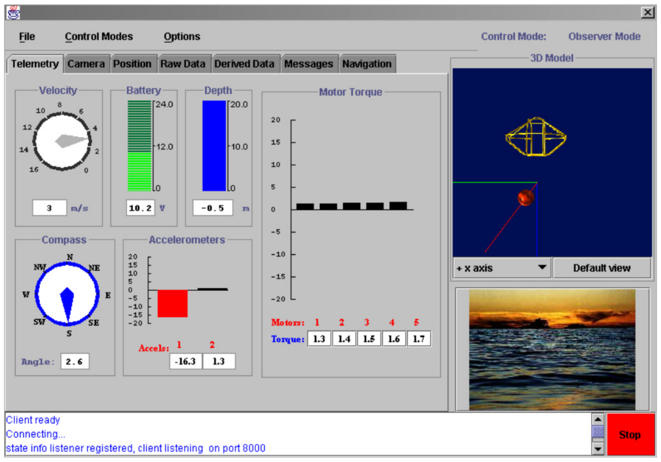
The aim of this project is to design advanced operator interfaces for underwater robots and to develop a specific instance for ANU's vehicle, Kambara. The interfaces will receive state information from the robots and present it to the operator in intuitive graphical representations. The interfaces then receive commands from the operator via mouse, keyboard or a combination to indicate the desired actions for the robot.

1.2. Motivation

Underwater robots require adequate guidance and control to perform useful tasks. The operator interface interprets telemetry and presents a numerical expression of vehicle state. It will provide a method for generating commands to the vehicle interface for direct teleoperation of vehicle motion and for supervisory control of the on-board modules. The eventual goal is to enable a user to command an underwater robot to hold station on a reef or swim along a pipe, and to have that user observe the results via real-time updates of a graphical user interface (GUI).

1.3. Scope

Specifically, the GUI being developed should provide a more intuitive representation of information than similar Autonomous Underwater Vehicle (AUV) interfaces presently being developed in the world. The exact form of this representation (see Figure 2) is directly related to some of the specific software objectives.



Control Mode: Observer Mode

Velocity: 3 m/s
Battery: 10.2 V
Depth: -0.5 m
Angle: 2.6

Accels:	1	2
	-16.3	1.3

Motors:	1	2	3	4	5
Torque:	1.3	1.4	1.5	1.6	1.7

```
Client ready
Connecting...
state info listener registered, client listening  on port 8000
```

Figure 2: Screenshot of the Kambara interface.

1.3.1. Software Goals

The software requirements of the project are to:

- present the robot telemetry information to the user in an intuitive graphical representation,
- develop the interface to have cross-network capabilities, allowing multiple users to view the state of the robot, while having only one user control it. Specifically, the cross-network capabilities will allow updates of the robot's state information at a rate of at least 10 Hz.
- provide a three dimensional model of the robot, to give the user an intuitive sense of position,
- develop a control system for the robot which allows simple point-and-click control from the interface,
- provide the ability to view footage from the robot's onboard cameras,

The cross-network portability is an important step that will make the interface more easily accessible for research purposes, as well as gaining outside interest in the project. All other robotics groups examined develop their interfaces with one viewer in mind; whereas the Kambara interface will allow multiple users to view the state of the robot, while allowing one user to take control.

The control system for Kambara allows for simple point-and-click control from the interface. By using Java 3D, consideration was made for long-term development, when alternative devices may be used for control.

Video is likely to be displayed to the user by using a digitiser to continually refresh an image which will be presented in the GUI. However, this relies on the submarine development having reached the point, for this to happen. Otherwise, an image placeholder will be used in anticipation of a video stream.

1.3.2. Java and the use of Java 3D

The use of the Java programming language is one important aspect where the Kambara interface stands out from others. No other AUV interfaces that have been examined so far make use of Java. In addition to using the Java programming language, the Kambara interface utilises the very latest developments of this language. The most recent version of Java (version 2), contains Swing libraries which enable exciting new developments with a GUI.

Finally, very few interfaces have 3D models to assist the user. The Kambara interface will make use of the emerging Application Programming Interface (API) known as Java 3D to provide the user with an intuitive representation of the submarine's position.

Inevitably, when developing an interface for any kind of vehicle it is necessary to provide a visual representation of it in order to assist an operator Many ground based robots make use of two dimensional views of the vehicle, often from two different angles (perhaps a top view and a side view). However, it seemed more appropriate to use a 3D model for Kambara, due to the fact that it has five independent degrees of freedom. This comes about from the fact that Kambara's five thrusters enable roll, pitch, yaw, heave and surge manoeuvres.

Java 3D also provides the capability to receive inputs from some of the latest input devices, such as six-degrees-of-freedom tracker information. This allows for the use of cutting edge technology to control the robot.

1.4 Thesis Outline

Bearing in mind the aims of this project, the details to come have been based on firstly examining aspects relating to GUI's in general. Following this is an examination of the various stages of the Software Development Life Cycle needed to complete such a project.

Chapter 2 investigates related projects at other research labs and organisations. This includes the examination of interfaces from not only AUVs but many other sources also. Following this, the human-computer interaction considerations for this project are investigated.

In Chapter 3, the reasons for selecting the Java programming language are discussed, prior to giving an introduction to the Software Development Life Cycle (SDLC) upon which this project is based.

The initial stages of the SDLC, analysis and design, are detailed in Chapter 4, discussing the specifications determined for the project, and the method for completing the determined tasks

This is then continued in Chapter 5 with the implementation section. Some of the problems encountered along the way are discussed, along with the application of the evaluated design. Following this, Chapter 6 compares the results of implementation, with the software objectives of the project.

Conclusions are drawn in Chapter 7, along with a discussion of the work which could continue using this project as a basis.

Clearly, when undertaking any project, it is important to be aware of the context in which the work is being done. The next chapter considers other work relating to the development of GUIs as well as considering the impact that the use of the GUI will have on a user.

Chapter 2

2. RELATED WORK AND HUMAN-COMPUTER INTERACTION

Examination of other existing AUV projects and robotics projects in general have been important in designing the GUI for Kambara. While the information on other AUV interfaces is scarce, there are many other sources of information which can be used to place the Kambara GUI in an appropriate context. It has also been beneficial to consider human-machine interaction issues when developing the Kambara interface.

2.1 Related Work

2.1.1. Comparison to AUV interfaces

Very few other AUV developers provided information about the specifics of their vehicle's interfaces. Despite this, information was gathered about the interface for the AUV upon which Kambara was based. This vehicle, called Oberon, is based at the University of Sydney. The code for this project was written in Visual C++ and made extensive use of TCP/IP (see Glossary) communications to allow for an event-driven paradigm to be implemented. The communications for Kambara was based along similar lines, however an RMI connection was used (see Appendix A).

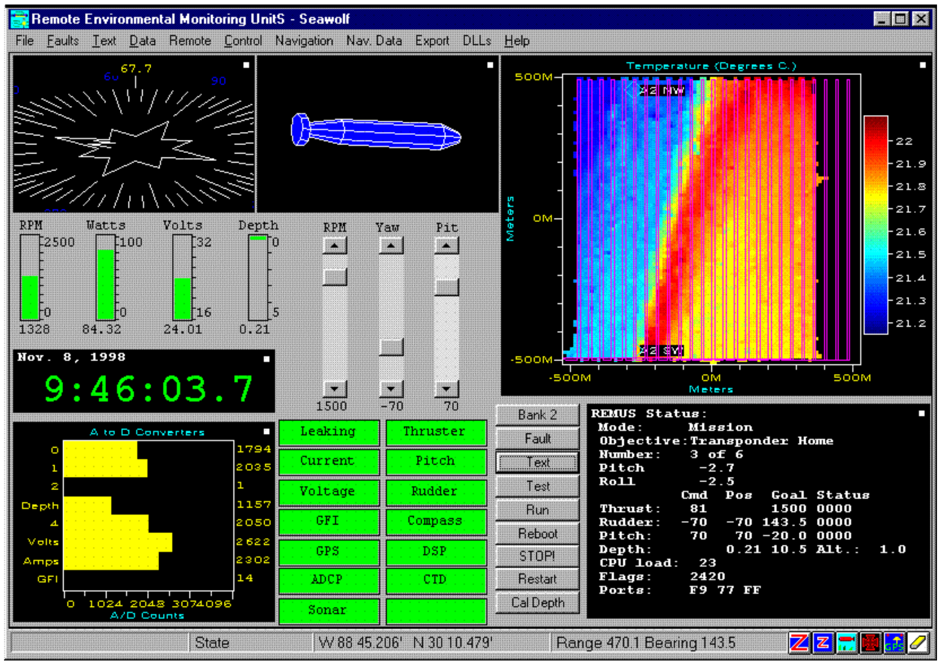
2.1.2. Comparison to general interfaces

In addition to studying the interfaces of underwater vehicles around the world, additional methods have been used to examine the types of graphical representations that might be necessary on Kambara.

2.1.2.1. Other Robotic or Simulator Interfaces

The interfaces of various telerobots at NASA were examined, and provided some ideas for the GUI layout as well as methods of control for Kambara. In particular, a software engineer at Jet Propulsion Lab at NASA (Paul Backes) is developing a similar system for Internet-based visualization and command sequence generation to be sent to Mars landers and rovers [19]. This is done using Java 2 and Java3D.

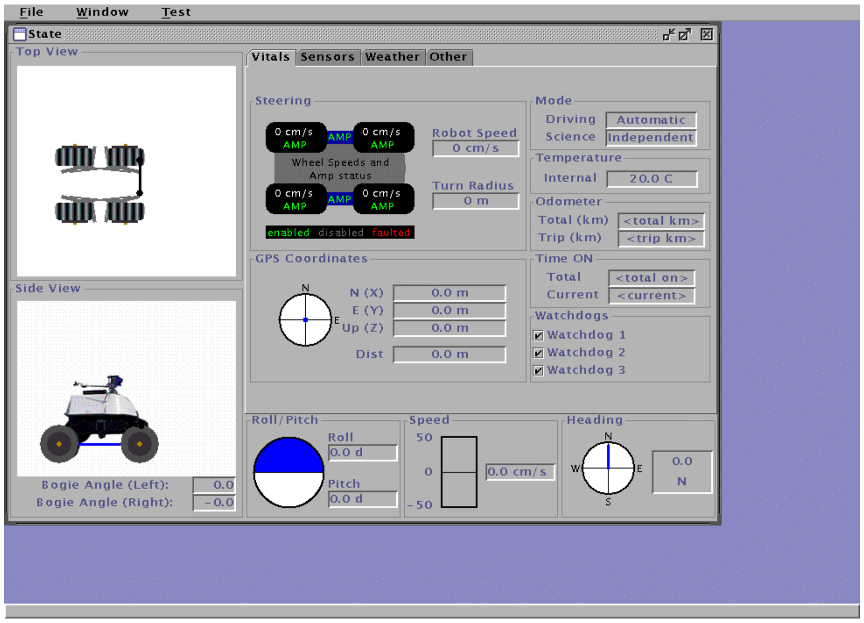
Remus (Remote Environmental Monitoring Units) is a developmental tool of the Oceanographic Systems Lab of the Woods Hole Oceanographic Institution, to perform underwater environmental research. [11]. The interface for Remus is shown in Figure 3, and while not written in Java, it does demonstrate ways to represent a lot of telemetry data coming from an underwater submarine. Of particular note, it is apparent that a 3D model has also been used by the REMUS team. This reinforces the choice to use a 3D model for the Kambara system.



```
REMUS Status:
 Mode:      Mission
 Objective:Transponder Home
 Number:    3 of 6
 Pitch        -2.7
 Roll         -2.5
           Cmd  Pos  Goal Status
 Thrust:    81       1500 0000
 Rudder:   -70  -70 143.5 0000
 Pitch:     70   70 -20.0 0000
 Depth:        0.21 10.5 Alt.:  1.0
 CPU load:  23
 Flags:     2420
 Ports:     F9 77 FF
```

Figure 3: Host software for the ROV named REMUS.

Another project, called NOMAD (at the Field Robotics Center at Carnegie Mellon University) is using Java 2 in its effort to develop robots for autonomous search of Antarctic meteorites. It is also used to demonstrate advanced control, navigation, and search technologies, as a terrestrial analog to robotic exploration of Mars and the Moon. This uses a tabbed pane similar to that used for Kambara, and provided a useful comparison [16]. It is apparent from Figure 4 that for a ground-based vehicle such as the NOMAD, the developers selected a 2D view rather than the 3D view opted for in Kambara's case.

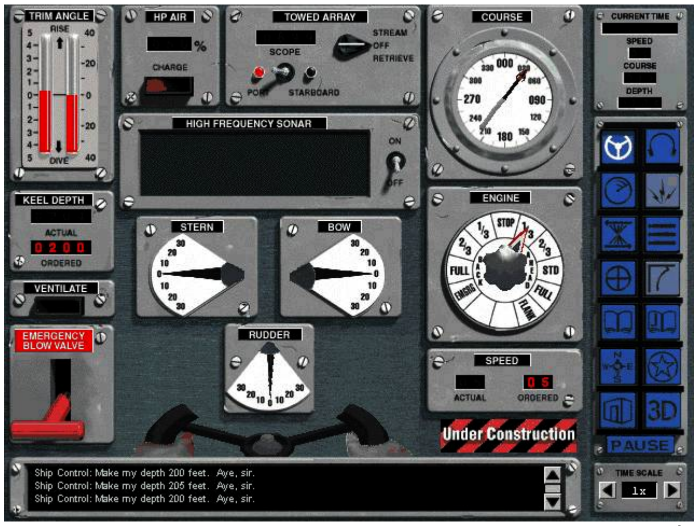


Bogie Angle (Left): 0.0
Bogie Angle (Right): -0.0
0 cm/s AMP
0 cm/s AMP
0 cm/s AMP
0 cm/s AMP
Robot Speed 0 cm/s
Turn Radius 0 m
Mode
Driving Automatic
Science Independent
Temperature
Internal 20.0 C
Odometer
Total (km) <total km>
Trip (km) <trip km>
Time ON
Total <total on>
Current <current>
Watchdogs
☑ Watchdog 1
☑ Watchdog 2
☑ Watchdog 3
N (X) 0.0 m
E (Y) 0.0 m
Up (Z) 0.0 m
Dist 0.0 m
Roll 0.0 d
Pitch 0.0 d
Speed 50 0 -50 0.0 cm/s
Heading 0.0 N

Figure 4: State Information for the robot Nomad.

2.1.2.2. Game and Dashboard Interfaces

Even games that simulate submarines and aircraft were observed to give an idea of the ways in which real-time data can be represented and controlled. The interface below indicates the way submarines are represented in a submarine simulator [15].



Ship Control: Make my depth 200 feet. Aye, sir.
Ship Control: Make my depth 205 feet. Aye, sir.
Ship Control: Make my depth 200 feet. Aye, sir.

Figure 5: Interface for the submarine simulator game 688 Hunter/Killer.

Finally real-life dashboards such as those in cars, submarines and yachts were observed, again providing alternative representations of certain parameters.

2.2 Human-Computer Interaction

Interface design ([12] and [13]) has a large affect on user satisfaction and influences the amount of time an operator spends with the robot. The aim was to design an interface that maintained the users' interest. This meant the interface had to be simple and intuitive but still provided access to all the functions of the robot. To accomplish this two main areas were examined:

1. Reducing the response time;
2. Making the interface easy to read and follow;

2.2.1. Reducing the response time.

In today's society people are often impatient and like to achieve their goals quickly. Reducing the response time means that information can be sent and received quickly, allowing more operations to be completed in less time. Although the response time is mainly dependent on the users' computer and server, over which there is no control, there are a few design aspects that will help with the fast relay of information. These are:

Using a tabbed pane - This means that when the user seeks more information (via a separate pane) the computer has already created all of the objects on it, and it is a relatively fast transition to the new page. This makes the response time shorter encouraging users to test all of the functions of the robot.

Simplifying the representation of data – By using graphical components, a user can easily identify the information on hand. This is compared with an interface consisting purely of numbers, which is quite difficult for the viewer to interpret.

2.2.2. Making the interface easy to read and follow

If an interface design is too complex with a lot of text or instructions then it is highly likely that the user will become quickly discouraged. For this reason it was attempted to make the interface as simple and intuitive as possible. This was done in a number of ways.

Reduced text - People tend to find it easier if the text is reduced to only the essential information; the reader can then scan the screen quicker.

Reducing the need to scroll - This was achieved by making all of the important information, for a particular aspect of Kambara, visible on a tabbed pane. Scrolling is still necessary in the message areas, however this is understandable due to the frequency with which a new message may be reported.

Reducing irrelevant components - Users can become confused if there are too many ideas on the screen. To reduce confusion components were removed if they were:

- Irrelevant to that particular pane,
- Out dated or not fully functional, such as the new control modes planned.

Labeling each component – Every GUI component's title was placed at the top of it's own self-contained frame. This ensures that a user can immediately tell what they are viewing.

Using Tooltips – Many of the GUI components make use of the Java tooltip feature. This means that when the user places the cursor over a component, a small label displays relevant information about that component. Of course, the fact was also considered that users who frequent the interface may find this quite distracting. For this reason, the tooltips can be turned off from the menu bar.

2.2.3. Resulting considerations

By considering the aforementioned factors, this left a simple, easy to follow design.

Consistent designs for each template - To reduce user confusion all of the different screen permutations have a similar design. They all have:

- The menu bar at the top, to allow for completion of general operations from any pane.
- The 3D model and video image constantly available. It was decided that these two items would be useful regardless of the pane the user is viewing.
- The message panel at the bottom. This displays warning messages, as well as allowing a user to track important aspects of their tasks.

2.2.4. Basic design principles

There are some fundamental aspects of a GUI which relate to the way in which a user perceives the information on display to them. Of these, space is the most important, with proximity, alignment and contrast also being important considerations [7].

2.2.4.1. Space

Space is the most effective element that can be used to provide support for the user in their cognitive processing of visual displays. Spatial relationships are perceived precognitively - that is, without conscious effort. They do not have to be decoded and interpreted, as do colour cues, typographical cues, and so on.

While it was difficult to find space within the Kambara GUI, due simply to the number of GUI components to be displayed, some measures were taken. As discussed earlier, each component was contained within it's own bordered frame, with an accompanying title. These frames typically had 5 pixel gaps both inwards to the component and outwards to other frames. This assists with the feeling that the user space is not cluttered.

2.2.4.2. Alignment

Human beings perceive items that are aligned vertically and/or horizontally to be more organised than those that are not, and people process, learn and remember organised information better than unorganised information.

Poor alignment creates too many perceptual "edges" in a display, so better alignment results in fewer features to be processed.

2.2.4.3. Proximity

Elements that are close together in a visual display will be assumed to be related, and conversely, elements that are far apart will not be seen as related to each other. When elements are not clearly differentiated by proximity, the audience has to group them consciously by focusing on them, taking in their meaning, and deciding which ones go together.

It was for this reason that a tabbed pane was used for the Kambara interface. The label on the tab immediately gives the user an idea of what they are viewing, and components have been grouped in to themes where possible.

Equal division of space results in poor proximity, as it is not clear which elements are related. Sometimes more or less space was used between elements to indicate which ones go together by virtue of their relative proximity.

2.2.4.4. Contrast

Contrast exists in several forms, the primary being:

- size,
- colour,
- shape.

Contrast can be used to make elements more or less dominant in the display, influencing the order in which they are processed and their perceived importance or urgency. For this reason, important features of the Kambara interface are both large and colourful. Examples include the 3D model which is very important to the control of the vehicle, and the torque controls which are also of great relevance. Also, the "Stop" button is very apparent, coloured red in the bottom right corner (see Figure 2); in case the user wishes to halt everything.

Insufficient contrast means that more and more treatments are necessary to make important information stand out. By using contrast that is sufficiently strong it is easy to use a few simple, easily-perceptible treatments.

With the context of the project in mind, and having considered how the final result would affect the user, it was necessary to move to the next stage of the interface's development. This meant selecting a programming language that was appropriate to the task at hand, and then entering in to the Software Development Life Cycle (SDLC).

Chapter 3

3. PROGRAMMING LANGUAGE SELECTION AND THE SOFTWARE DEVELOPMENT LIFE CYCLE

The programming language selected for any software development project can either be a tremendous aid or, in the case of poor selection, a profound hindrance. With this in mind the first, and one of the most important, decisions that was made on this project was the choice of the programming language, Java.

3.1 Programming Language Selection

3.1.1. Comparison of Programming Languages

C and C++ are fast and powerful low-level languages. They have a huge installed base, and most of the world's consumer software is written in these two languages, probably because of the importance of speed for the consumer market. C++ was certainly considered an option on this project [9].

Java is the newest of the languages considered. It incorporates many software engineering principles (object-oriented, strongly typed, good exception handling). It is the only language suitable for writing applets that run on top of browsers, and this is a fact that has done more to boost Java's popularity than anything else. Java's cross-platform compatibility and convenient APIs for networking and multi-threading also work in it's favour.

Visual Basic (VB) allows developers to construct programs by pasting various pre-built components into a workspace. VB has been widely adopted by the business world for building front-ends to databases and building prototypes for programs that will be later written in other languages. VB is quite restrictive in its network capabilities compared to other choices. [10]

So after examining a variety of potential programming languages, the two most likely to be used were Java and C++. In the end, Java was selected due to the numerous advantages it offered for developing the Kambara interface.

3.1.2. Java's Advantages

The Java programming language has many advantages over other languages. It is object oriented which allows programmers to design reusable components easily. Java has built in garbage collection which frees memory automatically. Plus, Java includes built in data structures and algorithms for creating GUIs and communicating with other computers over a network. Additionally, the emergence of the API Java 3D was going to make it much easier to develop a 3D model for Kambara. This is as compared to developing in the slightly lower level package OpenGL.

Another advantage Java has over other languages is its portability. When a Java program is compiled, it is not compiled into native machine code; instead it is compiled into byte code which can be interpreted by a Java Virtual Machine. Once a specific computer architecture has a Virtual Machine designed for it, the computer can execute any Java program that has been compiled into byte code. This portability becomes evident in web based applications. Although Java's portability gives it a clear advantage over other languages, this feature also creates one of Java_s biggest disadvantages.

3.1.3. Java's Primary Disadvantage

Although Java's ability for producing portable, architecturally neutral code is desirable, the method used to create this code is inefficient. As mentioned above, once Java code is compiled into byte code, an interpreter called a Java Virtual Machine, specifically designed for a computer architecture, runs the program. Unlike natively compiled code, which is a series of instructions that correlate directly to a microprocessors instruction set, an interpreter must first translate the Java binary code into the equivalent microprocessor instruction. Obviously, this translation takes some amount of time and, no matter how small a length of time this is, it is inherently slower than performing the same operation in machine code.

This is not as much of a problem as it used to be though. Java is being continually developed and optimised, and it's speed disadvantage is slowly but surely becoming less of a problem. It is sufficient to say that Java is fast enough to accomplish the tasks required of the Kambara interface, and it will only get faster.

3.1.4. Summary of Java's Selection

In order to complete the software objectives required as part of the Kambara GUI, Java was selected. This was done for the following reasons. It:

- is portable, that is it has cross-platform compatibility,
- has convenient APIs for networking and 3D programming,
- is easy to use for GUI development, and
- is acceptably fast for the tasks at hand.

The specific software used for this project was the Java Developer's Kit (JDK) version 1.2.1, and the Java 3D API version 1.1. This API is layered on top of OpenGL version 1.1 (or greater).

Having decided upon a language, the software development could commence. The starting point for this can be understood better by understanding the SDLC.

3.2 Software Development Life Cycle (SDLC)

There are a number of different models which are used to describe different approaches to software production (see [17] and [18]). These models are neither rigid, nor prescriptive and are treated chiefly as frameworks. The most widely quoted model is that of the Software Development Life Cycle (SDLC) (see Figure 6).

This model describes the software production process as being divided into a sequence of phases, which in turn can be divided into subphases. A fairly simple generic form of the SDLC is described below, and as can be seen in Figure 6, is in a format that explains the term 'waterfall model' that has sometimes been applied to it.

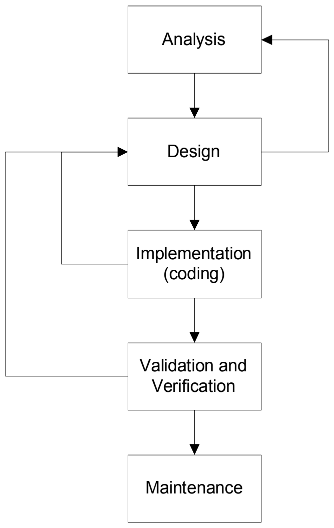


Figure 6: Overview of Software Development Life Cycle.

The major phases of the life-cycle can be identified as:

(1) Analysis, which is concerned with identifying what is needed from a system.
(2) Design, which is concerned with describing how the system is to perform its tasks so as to meet the specification.
(3) Implementation, which elaborates upon the design and translates this into a form that can be used on a computer system
(4) Validation and Verification, which is concerned with performing a validation of the implementation, in order to demonstrate how well it complies with the original requirements and the design.

In order to develop a GUI for Kambara, clearly it is necessary to know about the communication with the vehicle. Knowledge should be gained about the types of data to be output from the vehicle to the interface. Also, the types of control commands that may need to be sent by the interface should be identified. So in order to discover what was needed from the system, it was necessary to enter in to the first phase of the SDLC, the Analysis phase. This phase would be followed by the Design phase, however it should be remembered that due to the continuously iterative nature of the process, these two phases run almost concurrently.

C h a p t e r   4

4. ANALYSIS AND DESIGN

The analysis phase of the SDLC consisted of evaluating the type of data coming from the submarine, known as state information. Following this, the types of control commands to be sent to the vehicle were evaluated, and it was upon this basis that a design was developed.

4.1 Analysis

4.1.1. Definition of State Information

Prior to coding any components it was necessary to find out exactly what information would be required to be transmitted to and from the submarine. These initial analysis specifications are shown in Table 1. Note that an item is considered to be raw data if it comes directly from an onboard sensor on Kambara, whereas derived data is calculated based on the raw data. This data was expanded upon to wind up with the vector of values transmitted by the robot each time (see Appendix B).

The system parameters provided a good idea of the types of data to be represented, and allowed for more concrete ideas about ways to visualise the data. Some initial sketches were also made of possible ways to represent various aspects of Kambara's data and these resulted in some decisions on what was desired in the coding phase.

This led into a searching phase of the project, during which the Internet was scanned for various components which might provide useful building blocks. It is was important at this stage to bear in mind the software goals stated earlier.

Type of Data	Item	Number of item	Byte contribution
Raw	Accelerometer (sensor)	6 (2 x $\dot{u}, \dot{v}, \dot{w}$)	24
Raw	Velocity (sensor)	3 (p, q, r)	12
Raw	Compass position	3 (roll, pitch, yaw)	12
Raw	Compass magnetic disturbances	3 (B_x, B_y, B_z)	4
Raw	Compass Temperature	1	4
Raw	Compass Field Distortion	1	4
Raw	Depth	1	4
Raw	Motor voltage	5	20
Raw	Motor current	5	20
Raw	Battery voltage	1	4
Raw	Camera	3 (pan, tilt, zoom)	12
Raw	Computer status	64	256
Derived	Accelerometer (Kambara)	3	12
Derived	Velocity (Kambara)	3	12
Derived	Position (Kambara)	3	12
Derived	Motor command torque	5	20
Derived	Tracker State	Template(256) + Feature(24) + Peak(16)	296
Derived	Controller State	100 (vector of 100 numbers)	400
			Total: 1128

Table 1 : Evaluation of parameters required for representation.

Once the optic fibre tether is removed from Kambara, an image will also need to be transmitted along with this information. During the testing phase, images will come directly from video streaming to the client.

It should be noted that only some of these features were implemented during this initial development of the GUI. For example, tracker and feature state were not considered further due to the look of an image feed on which to base this information.

4.1.2. Planning for Control Modes

In order to facilitate future planning, it was necessary to define the ways in which a user would be able to use Kambara's GUI. Prior to explaining these modes any further, it is important to remember that only one user at a time would be permitted to actually manipulate the robot.

The four control modes decided upon were:

4.1.2.1. Observer Mode

This is the mode entered by the users who are not in control of the robot; it enables them to watch the activities of the operator. While this is technically not a control mode, it is the state in which a user would be in, if they have not successfully gained control of the vehicle. This mode allows viewing of all of Kambara's state information.

4.1.2.2. Individual Control Mode

This mode would be used when it is necessary to fire individual thrusters. This may be useful if, for example, Kambara is in an awkward position next to a rock. This would also apply to the use of manipulating items such as the pan of the camera.

4.1.2.3. Teleoperation Control Mode

The idea behind this mode is that the user will indicate the desire to perform actions such as move "up", "down", "left" or "right", and all of the thrusters will be coordinated to carry this out.

4.1.2.4. Supervisory Control Mode

This is a control mode similar to that used with the rover that landed on Mars in 1998. This form of control effectively allows the user to plan a route. This stage was planned for but not implemented.

Time-delayed teleoperation is laborious and unpredictable for remote operators. A better mode of operation is supervised teleoperation, or autonomous operation, in which the robot itself is responsible for making many of the decisions necessary to maintain progress and safety. [8].

4.1.3. Analysis Summary

The analysis gave an indication of the data that needed to be represented. A subset of this data was used in order to implement the system, with anticipation of more data being available to the GUI at a later point in time. With this basis in mind, the structure of the Kambara GUI system could then be examined.

4.2 Design

The design phase is concerned with describing how the system is to perform its tasks so as to meet the specification. The project was effectively split in to stages which are:

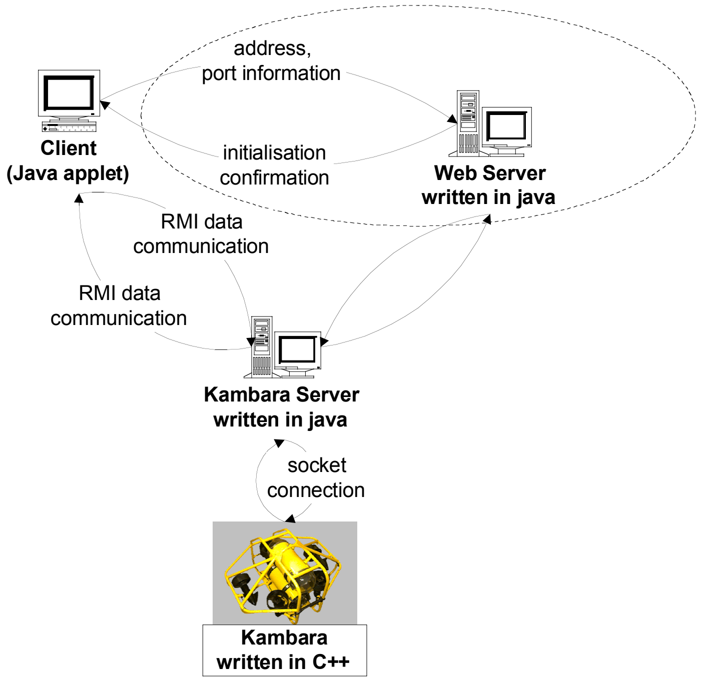
(1) Network Architecture, and
(2) Software Architecture.

This logical separation of tasks was necessary in order to aid the design process by making the tasks more distinct.

4.2.1. Network Architecture

4.2.1.1. Long-term Goal for Network Design

The network architecture of the eventual system was considered so that plans could be made to interface the client to the server, and to have that in turn communicate effectively with the submarine. The network system planned for in the long-term is shown below in Figure 7.



address, port information
Client (Java applet)
initialisation confirmation
Web Server written in java
RMI data communication
RMI data communication
Kambara Server written in java
socket connection
Kambara written in C++

Figure 7: Long-term Network Architecture for the Kambara System.

The dotted circle in Figure 7 refers to the Web server which will be used for initialisation. In the event that a user is running off a Java server other than that which the Web server is based on, the initialisation process will still need to be run once. However, from that point on, all client-server communication will be between the client applet and the Java server via Remote Method Invocation (RMI). A simple socket connection will be used between the Java server and the submarine onboard computer (VxWorks) since RMI only works between two Java objects.

4.2.1.2. Short-term Goal for Network Design

The system depicted in Figure 8 is that which is presently implemented.

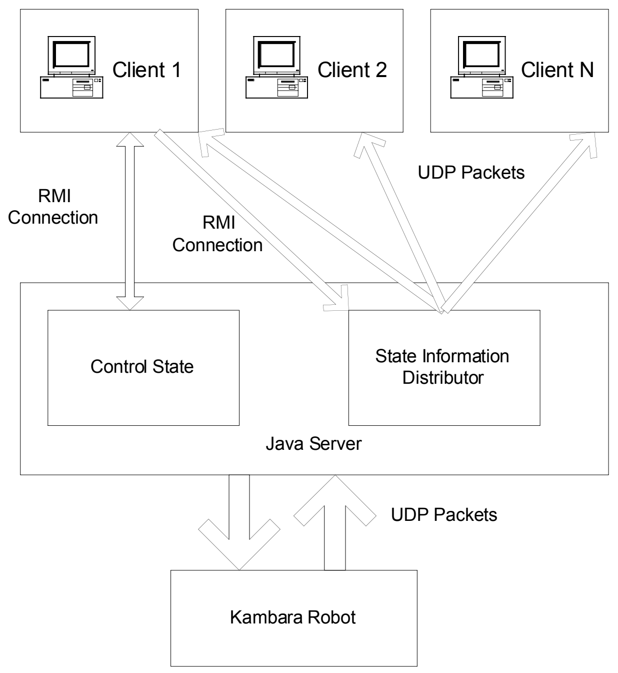


Figure 8: Short-term Network Architecture for Kambara System.

This essentially differs from the long-term goal in that the client is not built as an applet, but rather as an application. An applet will need to be used in order to allow the interface to function over the Internet, and this will bring with it additional security considerations (see section 7.2.2).

The fundamentals of the network are the same in both long and short term cases. There will be multiple clients connecting to a server (also written in Java), which in turn will be receiving state information from the robot Kambara. Each of these components as well as the network connections between them are explained in greater detail below.

Server

The easiest way to understand the network architecture being developed is firstly to examine how the server functions.

In general terms, the server is responsible for keeping track of and distributing the state information from the robot, as well as issuing control commands to the robot from "only one" of the clients.

State Information Distributor

Each client wishing to receive information about Kambara must register with the server. This is done via a RMI connection which can always be used for this purpose. The client opens a connection, and sends a request for state information.

If the client is not already registered then the server will place their address and port number in a queue as depicted below in Figure 9. All clients listed in that queue will then receive UDP packets containing the latest state information.

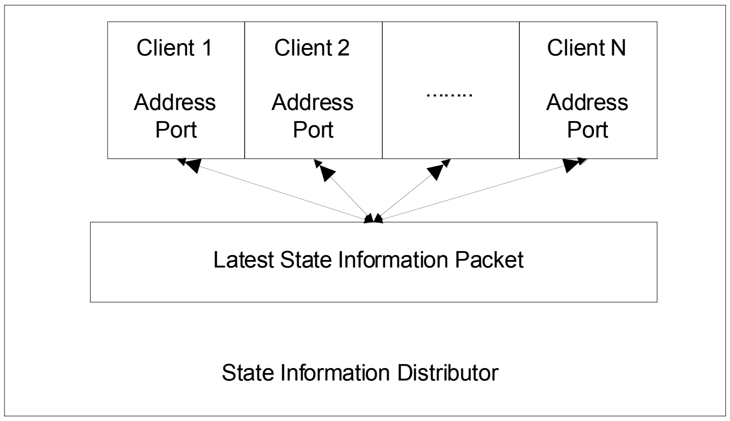


Figure 9: State Information Distributor.

An important point to note here is that it is the latest packet coming from Kambara that is stored at the server. By storing the packet, and not extracting the data from it, the details of the state information itself is abstracted away from the server. The server is simply responsible for storing the latest information, and passing it on to those who request it.

Control State

The focus of this project has been to develop a system which allows multiple users to view the data while allowing only one user to control the robot at a time. This ensures that there are no conflicting commands.

The server is responsible for ensuring that only one client at a time has the ability to control the robots, via the control state stored on the server. Essentially, the methods which allow a user to manipulate Kambara are synchronised such that while one client is controlling them, no other users are allowed to.

In the event that a control request arrives at the server from a client other than the controller, a message of refusal is transmitted to the requesting client's GUI.

Server Threads

In order for the server to function correctly, it must be listening for both control and registering requests from the clients, while continually receiving a stream of packets from Kambara. Also, these packets must be sent out to all registered clients as soon as they are received.

The first thing to discuss is the reception of control requests at the server. The mechanism for either asking for control, or sending a command is through the RMI connection discussed previously. Since the client is directly calling a function from an object on the server, this can happen at any time during the server's actions.

The reception and distribution of state information packets is accomplished by using two threads at the server. The first of these, called the *NetworkServer* thread simply runs through a loop whereby a packet is received, at which point the second thread running is notified. This second thread, called the *StateInfoManager*, is responsible for taking the new state information, and sending it out to each of the registered clients.

Client

An overview of the client's function is, to receive state information packets, and update the GUI with the received information. In the event that the client has control of the robot, the ability is also provided to issue commands to the robot.

The functions discussed above can be separated in to two categories, signals arriving at the client, and signals being sent by the client.

Client Output Signals

The first step for any client wishing to receive information is to register with the server. This is accomplished by using the RMI connection to call the server's method *addStateInfoListener*, which places the client's address and port number on to a queue. This same connection can be used to disconnect when the client's session has finished.

If the client is in any mode other than 'Observer mode', then they will have the ability to issue commands to the server. This is once again done via the RMI connection, and all of the functions called are defined within the server object *ControlState*. Different types of commands can be issued depending on the control mode involved.

Client Input Signals

Once a client is registered with the server it will begin to receive state information. The information comes to the client in the form of datagram packets from the server, via the protocol known as User Datagram Protocol (UDP). This is essentially an unreliable protocol in that the sender never knows whether or not the packet actually reaches it's destination; for this reason packets are occasionally lost. However, this loss of confirmation is acceptable in a system such as Kambara's in which information is being continually sent. This must be weighed up against the increase in speed which is gained by not needing to confirm every packet which is sent.

The UDP packets are each constructed at the server with the client's address and port number, obtained through the registration process discussed earlier. These can then be used to update the GUI components.

Client Threads

The client needs to perform three major tasks:

(1) Send registration, exit or command signals,
(2) Receive state information packets, and
(3) Update the GUI.

The first of these tasks is accomplished using RMI, whose commands can be sent at any time. For the second and third tasks, a thread is required for each.

The thread responsible for receiving packets is known as *KambaraClient*. This simply repeats the process " receive a packet then signal the second thread to update the GUI ".

The second thread is called *OutputKambaraData*, and it is responsible for updating the graphical components of the GUI with the new information. The updating of graphics can be quite time consuming in the context of a system where information is continually streaming in. It is for this reason that a separate thread is used.

Given an explanation of the client and server, it is now useful to consider the big picture, that is, exactly what happens in terms of command flow through the system.

Flow of Data Within The System

A typical flow of commands throughout the system is shown in Figure 10.

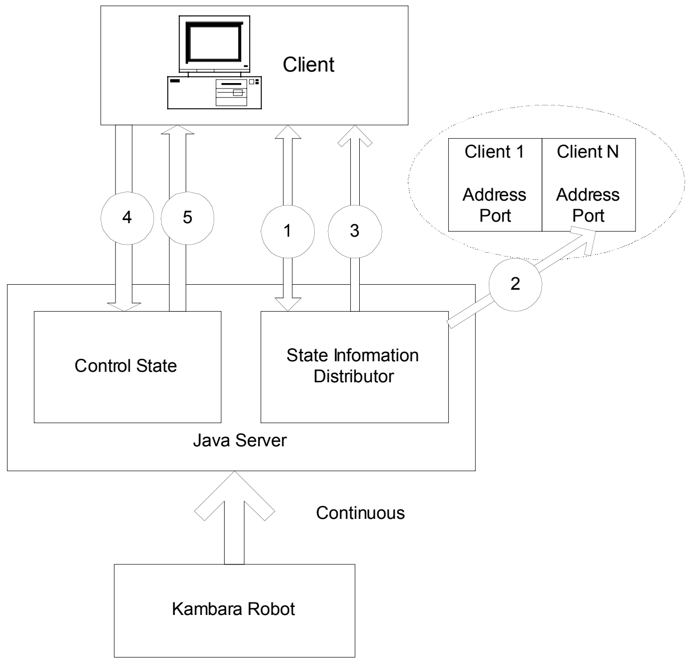


Figure 10: Typical Flow of Data in Kambara's system.

Each stage is discussed below, however the packets being sent from the robot to he server are considered to be continuous.

(1) The client sends a registration command by an RMI connection, indicating that they wish to receive the state information.

(2) The server takes the address and port number of the client, and places it in to the vector of client's to receive information.

(3) Each state information packet received by the server is then sent back to the client via a UDP packet.

(4) A registered client may wish to ask to control the vehicle.

(5) The requesting client is either sent back an error message indicating that the vehicle is presently being used by someone else, or is sent a confirmation, at which point the client's control mode changes as is appropriate. The client may relinquish control by returning to observer mode.

By understanding the different ways in which the server and client operate, it is now easier to gain an understanding of the software structure used to form these two entities.

4.2.2. Software Architecture

The software architecture is different for the primary components of the Kambara GUI system. It is best described by separately considering the client and server.

4.2.2.1. Software Architecture of the Kambara Client

The main module for the client is known as *MainKambaraClient* and is essentially just an object in which the three primary subcomponents are created. An overview of these subcomponents is now given before describing each of them in more detail.

Overview

Firstly there is the object *KambaraClient*, which deals primarily with the network details of the client. It is responsible for communication between the client and the server. This creates those GUI components with network components, and links them with the main GUI panel *KambaraMain*.

KambaraMain is essentially the basis for the GUI components. This is where all of the menus and graphical components are created and stored.

Finally there is a thread object called *OutputKambaraData* which runs when a network connection is established. This is responsible for continually updating the GUI components with any new state information received.

Network details (KambaraClient)

This subcomponent consists of the thread discussed earlier in section 4.2.1.2, Client Threads. This is responsible for setting up the RMI connection with the server, receiving a packet, and signalling a second thread to update the GUI.

GUI components (KambaraMain)

KambaraMain is the panel in which all of the GUI components were built. The way that this component is handled is quite important when considering whether the implementation will be an application or an applet.

By creating everything on a panel, this panel can either be inserted in to a frame in order to crate an application, or alternatively it can be inserted in to an applet when that function is desired.

The initial process taken in this design was to create a frame called *KambaraMainFrame* which was created as one of the three primary subcomponents within the client *MainKambaraClient*. *KambaraMain* is then created as a panel within *KambaraMainFrame's* layout. A visual representation is shown in Figure 11.

Essentially the software design for the GUI, which is based in *KambaraMain,* has been divided in to a tabbed pane, a 3D model, a video feed placeholder, a menu bar and a message area. Within the tabbed pane, the information is divided into separate panels depending on common themes of the various parameters. This grouping is quite logical, based purely on what information a user would want to see concurrently. It should be noted that only a selection of the panels and components are indicated in Figure 11.

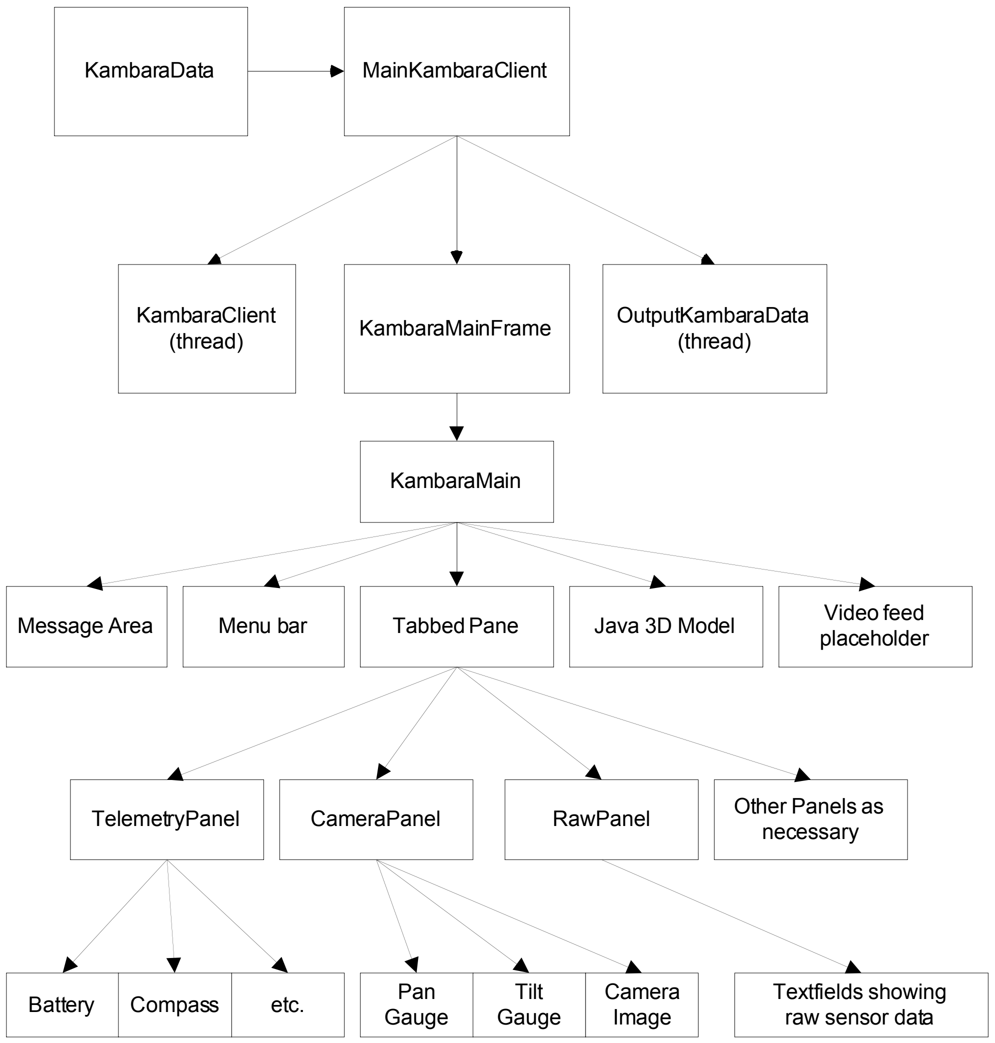


Figure 11: Software Architecture for the client side Kambara GUI.

KambaraMain is essentially the central object of the system, since a variable based on this initial object is passed to all of the subsequent components and panels. Before continuing any further, it is important to consider one other object which is created initially as part of a *MainKambaraClient* object, namely *KambaraData*. *KambaraData* contains the fields to store data each time Kambara sends information.

Throughout development, the structure *KambaraData* was used as an intermediate between the receiving stage and the output stage. Early on, only random values were written to this structure, and this ensured that when real submarine data or simulator data was received, it was simply a matter of writing correct values to *KambaraData* instead of random ones.

The rest of the GUI is set up when a new *KambaraMain* object is made within *KambaraMainFrame*. The idea is that *KambaraMain* generates it's subcomponents (tabbed pane etc.). Then all of the details that fit within an individual tabbed pane are created in separate modules such as *TelemetryPanel* and *CameraPanel*. These modules in turn create appropriate graphical objects based on smaller components.

The KambaraData object which is received by the client GUI is constructed from the packet sent by the Java server, so observing Figure 8 it is apparent how the software for the GUI fits in to the overall network structure.

Thread component (OutputKambaraData)
When a network connection is established via the RMI link, packets will begin to be received by the client. This thread will process the data contained with in these packets, and use it to update the GUI components on screen. This was discussed in more detail in section 4.2.1.2, Client Threads.

4.3. Conclusion

It was very important to define the network and software architecture prior to beginning the implementation phase. As is a continuing theme with the use of the SDLC, some specific details of these phases were modified as the Kambara system came to fruition. However the analysis and design discussed above certainly layed a solid framework upon which to develop this project.

Chapter 5

5. IMPLEMENTATION

The implementation phase elaborates upon the design and translates this into a form that can be used on a computer system. This was a time consuming phase of the project during which many obstacles needed to be overcome; and during which the establishment of a solid design was appreciated.

The implementation is discussed according to the software goals defined in 1.3.1, Software Goals. These are summarised below. The software requirements of the project are to:

1) present state information in a graphical form;
2) develop the interface to have cross-network capabilities; allowing multiple users to view the state of the robot, while having only one user control it;
3) provide a three dimensional model of the robot;
4) develop a control system for the robot which allows simple point-and-click control from the interface, and
5) provide the ability to view footage from the robot's onboard cameras.

5.1. Presenting State Information

The development of the GUI was carried out in a number of steps. Some components were based upon source code found on the Internet, while others were built from scratch.

5.1.1. Division in to Tabbed Panes

The information had to be divided in to a number of different areas in order to be presented successfully on the tabbed pane being used. These were:

(1) Telemetry Data: relating to general information about the vehicle
(2) Camera Data: the pan, tilt and zoom of the camera
(3) Position Data: a two-dimensional indication of position to be used in conjunction with the 3D model
(4) Raw Data: the numerical values coming directly from sensors on the vehicle

(5) Derived Data: the numerical values which have been derived on board the vehicle
(6) Message Data: the error, status or warning messages transmitted to the client.
(7) Navigation Data: the controls used to navigate the robot when in teleoperation mode.

Each of these categories was assigned a tabbed pane, and implemented using the GUI subcomponents developed. Thes screenshots of these can be found in Appendix C.

5.1.2. GUI Layout

One time consuming task while developing the GUI was to get the layout of the components correct. All of the components were built with resizing in mind so that the layout could be as flexible as possible. Despite this, in order to fix the components in to the desired layout, the dimensions of many of the components needed to be fixed. This is one aspect of the layout that could be improved.

The GUI components needed to be updated every time a new packet of state information was received. This can be quite a time consuming task due to the need to paint individual GUI components

The key aspect to be verified for all of these GUI components was the rate at which they could update the information. This is discussed in the next chapter, validation and verification.

5.2. Cross-network capabilities

The implementation of the network design (see Section 4.2.1) was quite difficult due to a number of unforeseen problems.

5.2.1. Socket Connections in Java 2

An initial problem was simply to allow for socket connections within the program. This is a problem simply because of the security arrangements in Java 2 [4]. Indeed, security is paramount to networked programs, in terms not only of the data travelling over the connections, but also of the making and breaking of those connections. Java 2 introduces a much more fine-grained control mechanism than previous versions. Instead of the all or nothing approach of applications versus applets, now everything is treated the same. The VM uses a policy file that is located on the local machine to determine what code is permitted to perform each action. Typically, this file resides in the lib/security directory where the Java Runtime Environment (JRE) is installed, in a file called java.policy. Inside this file is a list of each piece of code defined by location and what it is permitted to do.

For this reason, in order to allow socket connections to listen, connect and accept information, it was necessary to place a line in the java.policy file. This allowed for the use of sockets on the defined ports.

5.2.2. Using RMI on Windows NT

A problem which seemed to drastically affect the performance of the Kambara system came about when the network code was applied to the Windows NT system on which it would be used.

Having developed the networking aspects of the system on a Windows 98 system, there were no anticipated problems in transferring it to a Windows NT machine. This was not the case, as it took approximately twenty times longer to load the *KambaraServer* on the NT system.

It was evaluated that this problem was not a Java issue, but rather a DNS issue. The name of the local host, on which everything was running, was placed in to the Windows NT hosts file (C:\WINNT\System32\driivers\etc\Hosts). The performance then returned to what was expected based on the Windows 98 implementation.

As will be discussed next chapter (Validation and Verification), the passing of packets throughout this network implementation did have delays associated with it.

5.3. Implementation of Java 3D Model

5.3.1. The Evolution of the 3D Model

Prior to making use of the Java 3D API, it was necessary to have a representation of the Kambara submarine which could be manipulated as was necessary.

This was initially done with a very basic model created using AutoCAD seen here in Figure 12.

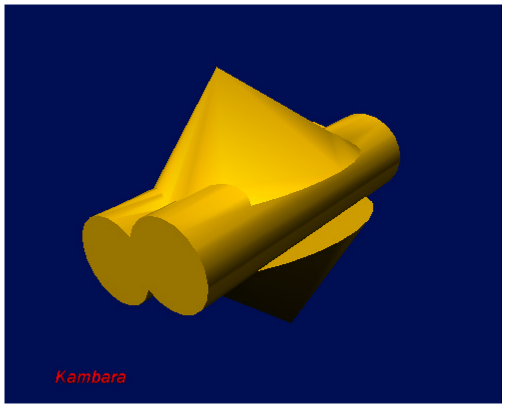


Figure 12: Early prototype of 3D model.

This was useful for determining the ways to load models in to a Java 3D scene, as well as for determining some of the basic rotation methods required.

As the need arrived for a more accurate model, one was provided having been created using the program Inventor. This was saved as a VRML world. It is possible to load a VRML world in to a Java 3D scene, however, the initial work had been done with a Wavefront (.OBJ) model in mind. For this reason, prior to manipulating the model it was desired that it first be converted in to a .OBJ file.

A converting program was found called Crossroads [14] which could convert among various 3D formats. By using a 3D studio format as an intermediate stage the VRML model was successfully transformed in to a OBJ model.

Some problems were found with the converting program Crossroads, in that it would only load one VRML object at a time for conversion. The VRML model provided was composed of several individual components combined together in to a scene. The linking file that creates this scene would not load up in Crossroads, and for this reason, only the submarine frame was initially used as the model (see Figure 13).

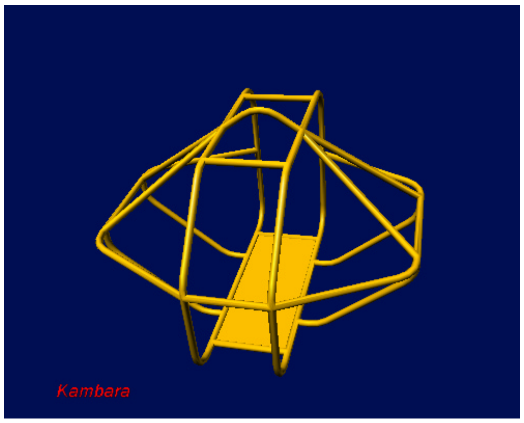


Figure 13: Intermediate 3D model of Kambara.

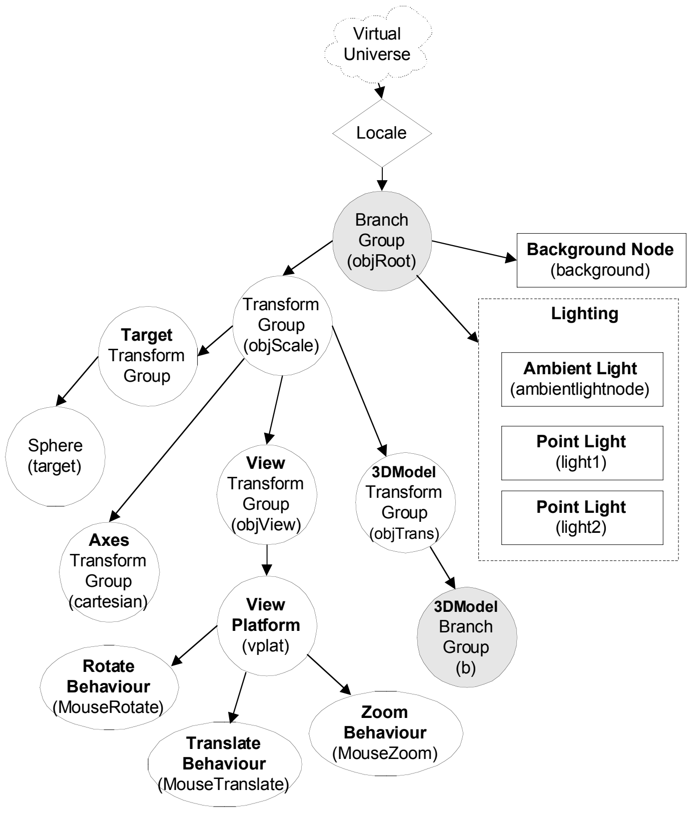
This enabled the development of all of the functionality required by the 3D model. However, by only using the one component the illusion was created of a faster frame rate than would be present when the full model was loaded in. This needed to be kept in mind, and hence the number of polygons in the model was reduced to account for this fact.

5.3.2. Applying the 3D Model

The Java 3D API core framework is based on a scene graph programming model. Pictures rendered with Java 3D are called scenes. There is an underlying object class structure that defines the composition of the scene [6].

The scene, or "virtual universe", is broken into the following components that make up the scene's composition: behaviour, model, object characteristics, 3D coordinate, math, and everything else needed to create a complex world of 3D objects. At the centre stage of the framework is the *SceneGraph* object. It contains a complete description of the scene, including model data, attributes, and viewing information.

To bring the Kambara 3D model to life, the hierarchy planned was as seen in Figure 14.



Virtual Universe
Locale
Branch Group (objRoot)
Background Node (background)
Lighting
Ambient Light (ambientlightnode)
Point Light (light1)
Point Light (light2)
Transform Group (objScale)
Target Transform Group
Sphere (target)
Axes Transform Group (cartesian)
View Transform Group (objView)
3DModel Transform Group (objTrans)
3DModel Branch Group (b)
View Platform (vplat)
Rotate Behaviour (MouseRotate)
Translate Behaviour (MouseTranslate)
Zoom Behaviour (MouseZoom)

Figure 14: Scene Graph hierarchy for Kambara's Java 3D System.

The scene comprises parts that retain their individuality, yet represent the measurable whole called the *virtual universe*. To this a *Locale* object is attached which provides the coordinate system.

A collection of several subgraphs is created and attached to the Locale object. For the Kambara system, the subgraphs required were:

- *Background:* This subgraph simply provides the background colour, indicative of the water in which the submarine is travelling.
- *Lighting:* Ambient light was required to show the vehicle, and two point lights were used to give a more intuitive sense of orientation as the submarine moves.
- *3D Model:* A subgraph was used for the vehicle itself. Essentially at the bottom of this subgraph, the .OBJ file is loaded in to a Java 3D *SceneGraph*

and then extracted to become a *BranchGroup*. The *TransformGroup* associated with this can then be used to move and orient the vehicle.

- *Target:* The target is represented independently so that it can be manipulated to the desired destination of the model.
- *Axes:* The co-ordinate system is represented with 3 lines so that a user can visualise what Kambara is doing. These lines are coloured red, green and blue to correspond respectively with the X, Y and Z axes. This is in line with the convention of representing XYZ axes with the RGB colour scheme.
- *View:* Finally a view platform is attached which gives the user the ability to manipulate the position from which they are viewing the scene. This view has several behaviours associated with it to enable rotating, zooming and translating of the viewpoint.

All of the objects that compose the root *BranchGroup* are contained within a bounding sphere. This was made arbitrarily large, since the behaviours are always required to apply.

5.3.3. Co-ordinate System Problems

Throughout implementation there were problems combining the coordinate system represented in Java 3D with the interpretation of Kambara's co-ordinate system by other group members.

When the Java 3D model created was loaded in to the virtual universe, the model defaults to the orientation shown in Figure 15.

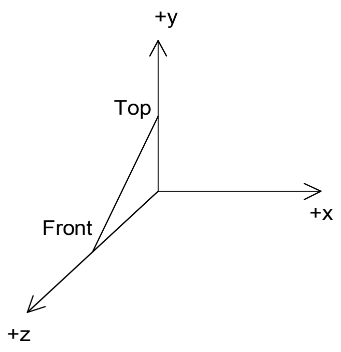


Figure 15: Default 3D orientation.

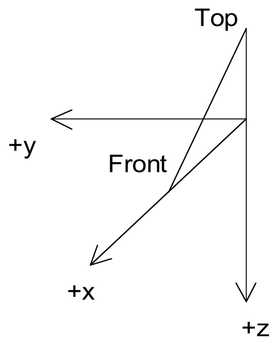


Figure 16: Desired 3D orientation.

This state has the model with it's front facing down the +z axis and it's top facing along the +y axis. However this does not correspond with the way that the submarine's orientation was interpreted in order to use the learning algorithms involved in developing it's autonomy. For this reason, prior to applying the learning algorithm's estimation of orientation, it was first necessary to rotate the submarine model in to the desired position (see Figure 16).

5.4. Control Modes

The basic control modes were successfully implemented in the system. When the user first loads the GUI application, a label in the top right corner indicates that the current control mode is "Disconnected". During this state of operation, the options under the Control Mode pulldown menu are disabled.

Upon successfully connecting to the server, the control mode automatically changes to "Observer Mode", at which stage the user can view the state information coming from Kambara. This also enables the control mode menu, providing further control mode options.

The next issue dealt with was whether the user can gain control of the vehicle. If any of the remaining 3 modes are selected from the control menu (Individual mode, Teleoperation mode or Supervisory Mode), then the user will receive one of two signals in reply to their request, either:

(1) *Request granted*: control is granted to the user since the robot is either not being controlled, or is already in control of the requesting user.
(2) *Request denied*: the robot is presently being controlled by someone else

As discussed throughout this project, only a very basic control system has been implemented.

The Individual control mode has been implemented, and when selected provides a number of buttons on several components. Control of individual motors can be managed by increasing or decreasing the desired torque using the buttons which appear in this mode. This is also the case for the pan, tilt and zoom of the cameras.

The teleoperation mode allows the user to transmit some basic high-level navigation commands. Essentially, the user can choose out of eight directions in its lateral plane, as well as being able to move forwards or backwards (see Appendix C7).

For both of the above modes, the control signals are presently just sent to the server. Future implementation will require the development of a way to transmit these control commands from the server to the robot.

The supervisory mode was not implemented. Essentially, the supervisory mode will allow the user to plan out a route to follow, or to execute a number of commands in sequence. Since the controls implemented so far are at such a low level, the supervisory mode was not developed. This is discussed further in the next chapter (see section 7.2.1).

5.5. Video Feed

It was planned to use a digitiser to continually refresh an image which would be presented in the GUI. Unfortunately, the submarine development had not reached the stage were this could be tested. An image placeholder was used in anticipation of the video stream being present. This image was treated in the same way as the 3D model, in that it is always on screen (never obscured by a tabbed pane selection).

Chapter 6

6. VALIDATION AND VERIFICATION

The next phase was concerned with performing a validation of the implementation, in order to demonstrate how well it complies with the original requirements and the design.

This essentially meant testing the primary software requirements to see if they met their objectives.

6.1. Testing procedure

Firstly a note on the way in which tests were carried out. Problems were encountered in implementing the application on a Solaris machine. This was due to the fact that Java 3D only runs on version of Solaris equal to or higher than 6. As a result, the Kambara system developed was always tested on the one machine (this was done on both a Windows NT and Windows 98 system). Specifically, the client(s) and server were always be running on the local host. This was not too troublesome, however, loading the one machine's processor with all of these programs is bound to lead to poorer performance than if each component of the program was run on a separate machine.

The client-server system would receive packets from one of two inputs. The first of these was a submarine simulator, whose packets contained information giving a good estimation of submarine data, particularly with relevance to position and orientation. The second possible input was simply a random packet generator (*RobotServer*) containing values that were within the appropriate ranges, but were generated randomly. These two inputs both used the same packet format, and hence were interchangeable.

Unfortunately, the sensor data from the submarine itself was not available during the testing phase of this project.

6.2. Basic design principles

The GUI design principles discussed in 2.2.4, were all satisfied during the implementation phase. Space and alignment were used to maximum affect. Additionally, proximity was used to group similar components.

6.3. State information updating

The ability for the client to view the state information coming from the robot was examined in a number of ways.

Firstly the speed at which the client updates the GUI was examined. This was done on two separate systems, Windows NT and Windows 98. To do this, the robot simulator was started up, and the time it took to send a number of packets was recorded. During this same period, the number of packets received by the client was also recorded.

The average results are shown below in Table 2.

Test	Number of packets sent by simulator	Number of packets received by client	Time (s)	Update frequency (Hz)
With server present (Windows 98)	647	198	53.6	3.7
Without server present (Windows 98)	763	440	43.2	10.2
With server present (Windows NT)	698	286	65.3	4.4
Without server present (Windows NT)	703	402	36.8	10.9

Table 2: Results from tests of GUI update rate.

It should be noted that the performance of the client achieved the software objective of an update rate of 10 Hz when the server was not present in the system. However, the performance was not satisfactory with the server present. It is also apparent that performance is greater on the Windows NT system, though this could be attributed to processor power, rather than the operating system itself.

The drop in speed due to the server is one area that should be optimised. It may be necessary to do some buffering at the server, or possibly to optimise the thread handling.

It should be remembered that all three of the programs involved in testing (client, server, simulator

6.4. Network testing

Aside from the network aspects of the GUI updates discussed above, there was another aspect of the network to test. It was examined if it was possible to have multiple clients connected to the server.

The case of two clients connecting to the server was tested, and was implemented successfully. The server acknowledged the registration of both clients, and sent state information updates to each of them.

6.5. Control mode testing

The control modes implemented, observer, individual and teleoperation were all successfully tested.

Firstly, the disconnected state was examined. When in this state, a user can not enable any of the options in the Control Mode menu. This is intuitively correct, since a disconnected user can not control the robot.

The observer mode simply allowed the user to view the state information, while having the ability to select another mode if desired.

When in the individual mode, various components have buttons added to them, which allow for the sending of relevant signals. These buttons all successfully called functions on the server using the RMI connection. These functions simply acknowledged the action of the button, without any further action taking place

This was also the case in teleoperation mode, except that instead of adding new buttons, the Navigation tabbed pane is added for the user's selection.

Finally, the control and network aspects were combined in a test. Two clients logged on to the server, and one gained control of the simulator robot. This client was able to switch between the various control modes, whereas the other client was not able to gain control of the vehicle. Control was relinquished when the controlling client returned to observer mode.

Chapter 7

7. CONCLUSION AND FURTHER WORK

7.1. Conclusion

Many important underwater tasks rely on portraying a submarine's parameters to a controller in an intuitive representation. This project has used Java along with it's associated APIs to communicate an AUV's information to the user in a more convenient manner than is presently used on similar interfaces.

This interface has cross-network capabilities, allowing multiple users to view the state of the robot, while having one user control it. Also, a simple control system has been implemented, allowing control via a point-and-click method on either images or buttons on the interface.

The update rate of the client GUI achieved the software objective of 10 Hz, however this was only without the server present. With the server in the system, the update rate was closer to 4 Hz.

A 3D model is provided to aid in visualising Kambara, and by using Java 3D the possibilities are open for more sophisticated control devices to be used in the future.

The ability to view the live video has not been implemented. This was not investigated more thoroughly due simply to the fact that the video stream was not available during the testing phase. A placeholder has been used on the interface to account for the time when a video feed is used.

At present, input to this system is only via a simulator. However, by building upon this existing GUI system, a user will be able to command an underwater robot to perform useful tasks, while observing the results via real-time updates on the GUI.

7.2. Further Work

This project has focused on ensuring that data coming from the submarine, or in this case, a submarine simulator, can be successfully displayed to clients who request such information. However there are various aspects of the project which can be built upon, the most apparent of which is the control system.

7.2.1. Extending the Control Modes

During the implementation of the Kambara GUI system some basic point and click controls were implemented. These provided the basics of the Individual and Teleoperation control modes. However the control system can become far more advanced than this.

While the basis of the Individual control mode is there, it can still be improved to be more user friendly. For example, presently in order to reach a certain pan angle on the camera, the increase button is pressed until the desired value is reached. It would more useful to be able to drag the needle with the mouse to the desired value. This is just one aspect where improvements could be made.

The teleoperation mode really only consists of a control panel allowing the operator to tell the robot to move in one of ten directions (up, up right, right, forwards etc.) while increasing or decreasing the yaw, pitch and roll. This method of control will only really be useful during the testing stages, to check the thrusters are coordinating as expected. Some useful control methods should be investigated. By using a Java 3D model as part of this system, a number of input devices should be considered options. Some options include a simple joystick, or, a more advanced method, the 6 degree of freedom tracking devices which are now available.

A plan will need to be made for a supervisory mode control system in anticipation of future robot operations when an optic fibre tether is not present. The ability to control in supervisory mode is an important consideration as there is a significant time delay between sending a command and the response of a robot to the command when an optic fibre tether is no longer present. It is desirable to have a system whereby this time delay is accounted for.

One final aspect of control which may be useful is to store a queue of users who are requesting control. Presently, if a user makes such a request and the robot is already being controlled, they are simply denied their request. It might be useful to have a system, in which such a request is placed on a queue in anticipation of the robot becoming free for control.

7.2.2. Implementing long-term network architecture

In section 4.2.1.1, the long-term network goal of this system was discussed. This essentially refers to the objective whereby it is possible to use this interface across the Internet. In order for this to happen, the application presently developed will need to be transformed in to applet form, in order to run in a browser such as Netscape.

This brings with it additional complications. Specific Java plug-ins need to be run with the present versions of Internet browsers in order for them to display Java 3D objects. Also, an applet form has additional security considerations associated with it. This may limit the way in which operations such as logging and storing client details are carried out.

7.2.3. Logging information

Presently, the logging system is in it's primitive stages, with the menu option having been established, but no logging actually commencing. This has several aspects upon which improvements can be made.

File storage is going to be a problem simply due to the amount and frequency of data which is being transmitted. Multiple files are likely to be required, that is, at a certain file size, a new file should be opened and the previous one stored. Additionally, the files could be written in binary rather than ASCII in order to save some space.

The method of logging should also be examined. When this system is running as an applet, the Java security arrangements will not allow for writing on a client's system without their express permission. For this reason, it may be desirable to have a separate logging process running at the server end, whose information can be requested by the client.

7.2.4. GUI improvements

The layout of the GUI is an issue that deserves some further investigation. Presently, information is grouped logically in to individual tabbed panes. However, user testing may show that it is more beneficial to be able to have any individual GUI component (eg compass, battery) on screen at one time. It may be more useful to implement the system as a series of internal frames, or possibly even allow the user to select what components are available to them via checkboxes.

With a change of layout representation, there may be problems with the resizing of components. As discussed in the implementation section, many of the components have fixed dimensions, in order to get them in to an appropriate position within the tabbed pane. This may be a hindrance if trying to implement a more dynamic layout.

7.2.5. 3D model improvements

There are also methods by which the 3D model can be used more effectively. The first improvement would be to implement the more complicated 3D model developed using Inventor. Due to the conversion problems discussed in section 5.3.1, only the submarine frame is used in the model at the moment

Presently this model is used simply to represent the position and orientation of the submarine in 3D space. There are many other ways in which this could be used. For example, the motor torques presently represented by a bar graph may be able to be visualised as a vector coming either forwards or backwards out of each of the 5 submarine motors.

Additionally, the exact position values of the submarine in each of the 3 Cartesian axis directions may be of use. This could be done by encasing the submarine in some sort of transparent sphere, on to which vector projections could be made.

7.2.6. Video feed implementation

The image placeholder on the GUI should be replaced by the streaming video feed. Additionally, plans should be made to send pictures to the GUI when the optic fibre tether is no longer present.

7.2.7. Client implementation

The update speed of the client GUI could also be optimised. Presently, the threads doing the updating are polling one another based upon the value of a boolean variable. This is not the most optimal solution, and alternatives should be investigated. Additionally, the speed reductions due to keeping one socket open the whole time should be examined.

7.2.8. Server implementation

Some features will need to be added to the server in order to provide a more robust system. The assigning of port numbers to registering clients is currently done simply by incrementing the port number counter. This has drawbacks in the case that when a client disconnects, their port number is never again assigned.

Finally, it would be useful if the control status of the Kambara robot could be transmitted to each registered client. This would prevent client's from having to attempt to gain control in order to find out if the robot is available for use.

BIBLIOGRAPHY

[1]
C. Gaskett, D.Wettergreen, A. Zelinsky. Reinforcement Learning for a Visually-guided Autonomous Underwater Vehicle.

[2]
Lemay, Laura and Cadenhead, Rogers. *SAMS Teach Yourself Java 1.2 in 21 Days*. Sams Publishing. 1998.

[3]
Niemeyer, Patrick and Peck, Joshua. *Exploring Java*. O'Reilly and Associates. USA. 1996.

[4]
Couch, Justin. *Java 2 Networking*. McGraw-Hill. New York. 1999. Pg 38-41

[5]
Flanagan, David. *Java in a Nutshell*. O'Reilly and Associates. USA. 1996.

[6]
Brown, Kirk and Peterson, Daniel. Ready-to-Run Java 3D. Wiley Computer Publishing. New York. 1999.

[7]
Visual Design Resources.
http://www.indiana.edu/%7Eiirg/ARTICLES/VIZRES/resource_page.html.
June 1998.

[8]
Carnegie Mellon Robotics Institute. http://robotweb.ri.cmu.edu/lri/lrd/nav-home.html. July 1998.

[9]
Programming Language Comparison. http://odin.bio.sunysb.edu/tedshieh/

[10]
Programming Language Performance.
http://www.cs.colostate.edu/_cs154/PerfComp/index.html

[11]
REMUS software. http://adcp.whoi.edu/REMUS/remus_software.html

[12]
Telerobot at University of Western Australia http://telerobot.mech.uwa.edu.au/.

[13]
Laurel, Brenda. The Art of Human-Computer Interface Design. Addison-Wesley Publishing Company. New York. 1990.

[14]
Crossroads 3D converter.
http://www.europa.com/~keithr/crossroads/download.html

[15]
Submarine simulator 688 Hunter/Killer review
http://www.janes.ea.com/screen/688_02.jpg.

[16]
NOMAD GUI. http://www.frc.ri.cmu.edu/~patrickd/nomad.html.

[17]
Budgen, David. *Software Design*. Addison-Wesley Publishing Company. New York. 1994. Pg 14-16

[18]
Wattam, S.I. *Software Engineering, A Dynamic Approach*. Sigma Press. Wilmslow, England. 1991. Pg 68-70

[19]
NASA Jet Propulsion Laboratory. http://robotics.jpl.nasa.gov/

APPENDIX A: REMOTE METHOD INVOCATION (RMI)

Since the use of RMI is an integral part of the Kambara GUI system, it is worthwhile to explaining in more detail what happens behind the scenes with RMI and why it was selected to be used for control and registration commands across the network.

RMI allows an application to call methods and access variables inside another application, which may be running in a different Java environment or a different system altogether, and to pass objects back and forth over a network connection. RMI bases most of its functionality on serialization to pass classes to and return them from the remote objects. The packaging and passing of method arguments is one of the more interesting aspects of RMI, as objects have to be converted into something that can be passed over the network. This conversion is called serialization.

At the most basic level, object serialization is the ability to write an object instance to a stream and then reconstruct that stream into the exact replica object instance, potentially on another machine [4]. As long as an object can be serialized, RMI can use it as a method parameter or a return value.

The RMI Architecture

The primary goal of RMI was to make interacting with a remote object as easy as interacting with a local one. In addition, however, RMI includes more sophisticated mechanisms for calling methods on remote objects to pass whole objects or parts of objects either by reference or by value, as well as additional exceptions for handling network errors that may occur while a remote operation is occurring.

RMI has several layers in order to accomplish all of these goals and a single method call crosses many of these layers to get where it's going (see Figure 17).

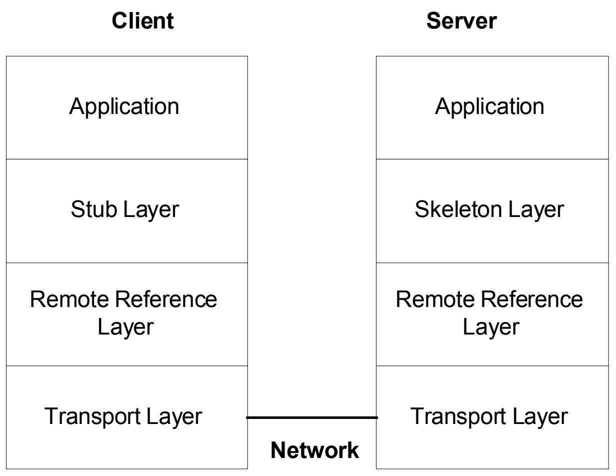


Figure 17: RMI layers.

- The "stub" and "skeleton" layers are on the client and server, respectively. These layers behave as surrogate objects on each side, hiding the "remoteness" of the method call from the actual implementation classes. For example, in the client application it is possible to call remote methods in precisely the same way as calling local methods; the stub object is a local surrogate for the remote object.

- The Remote Reference Layer handles packaging of a method call, its parameters and return values for transport over the network.

- The transport layer is the actual network connection from one system to another.

Having three layers for RMI allows each layer to be independently controlled or implemented. Stubs and skeletons allow the client and server classes to behave as if the objects they were dealing with were local, and to use exactly the same Java language features to access those objects. The Remote Reference Layer separates the remote object processing into its own layer, which can then be optimised or reimplemented independently of the applications that depend on it. Finally the network transport layer is used independently of the other two so that you can use different kinds of socket connections for RMI.

RMI versus Sockets

Remote objects are a way of abstracting the client/server approach so that the programmer needs to know only what functionality and what data need to be passed to the server, without having to know how to get it there and back.

At the very lowest level, any communication between two machines requires the use of the network. Invariably, that involves the use of a socket connection, over which data must flow.

RMI is a more sophisticated mechanism for communicating between distributed Java objects than a simple socket connection would be, because the mechanisms and protocols by which you communicate between objects are defined and standardised. You can talk to another Java program using RMI without having to know beforehand what protocol to speak or how to speak it.

Files Needed for RMI

Building an RMI-based system requires at a minimum three files. First there is the need for an interface that defines what methods are going to be used. Next you will need an implementation of that interface- the server code. Finally you will need something that uses the server code – the client. (see Figure 18)

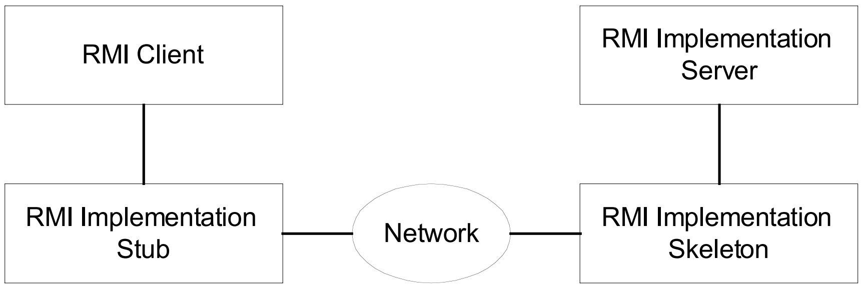


Figure 18: The flow of control from the client to the server and back.

RMI Compiler

The RMI Compiler (RMIC) is used to provide the glue between the implemented code and the networking code. It takes the server-side implementation of the code and produces two extra files- a skeleton and a stub. In these files are hidden all the low-level networking code. The stub is located on the client and the skeleton is located on the server. The stub provides a proxy that generates the actual calls to the skeleton, which then forwards the requests onto the real implementation instance on the server (see Figure 18)

RMI Registry

Having the server code and client code almost completes the package. The final part is something that listens for RMI connections and deals with issues of loading the correct class instances and connection and termination of clients. The RMI Registry is responsible for doing this.

During code development, the registry was run from the command line. The reason for this is that it takes the current class image and serves that to clients. If you change the server code implementation, the changes will not be picked up by the client. It was necessary to stop the registry and restart it with new code. Once the network code on the server for this project was completed, the registry was run as a process. The implemented code dynamically creates the registry and makes use of many other RMI capabilities.

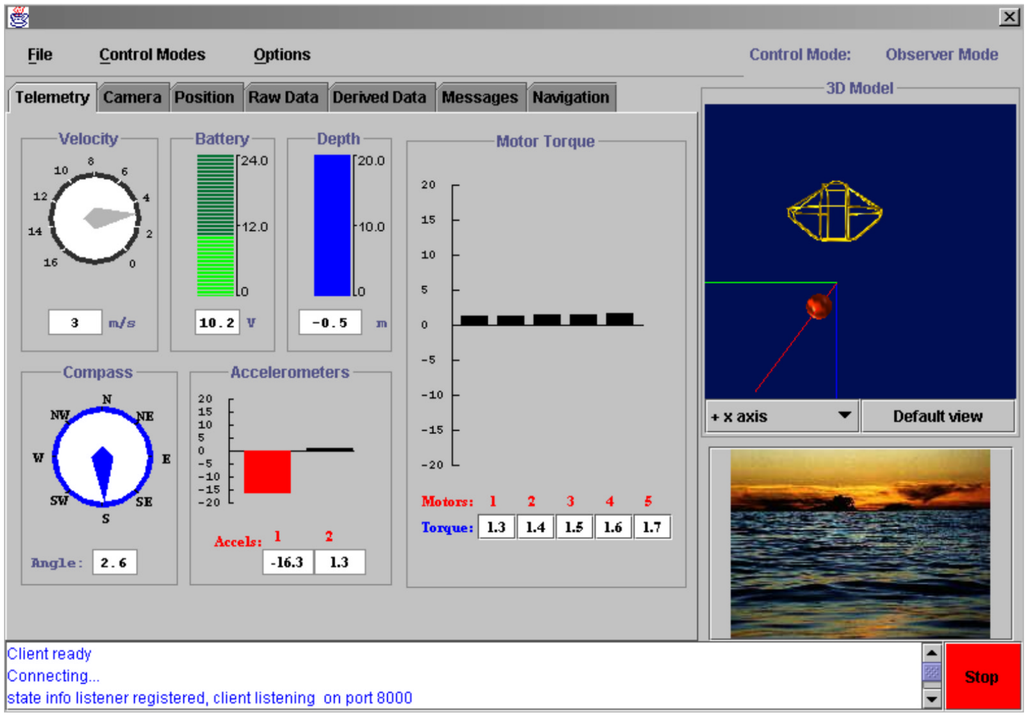
RMI Security Issues

During development it was necessary to enable socket permissions in the JDK file (see 5.2.1). It was necessary to explicitly place the name of the machine on which the code was operating (sometimes localhost) within the permission statement. Depending on the version of the JDK, RMI does not treat localhost and the local machine name as being equivalent.

APPENDIX B: VECTOR OF DATA TRANSMITTED BY SIMULATOR

Field element/range	Data represented
0-2	Vehicle position
3-6	Vehicle orientation (quaternion)
7-9	Vehicle local velocity
10-12	Vehicle angular velocity
13-17	Motor torque
18-20	Vehicle target
21	Battery voltage
22-23	Accelerometer totals
24	Compass angle
25	Camera pan value
26	Camera tilt value
27	Camera zoom value
28-32	Motor current
33-37	Motor voltage
38-40	Derived accelerometer 1 u,v,w values
41-43	Derived accelerometer 2 u,v,w values
44-46	Raw accelerometer 1 u,v,w values
47-49	Raw accelerometer 2 u,v,w values
50-52	Compass magnetic disturbances

APPENDIX C1: SCREENSHOT OF TELEMETRY PANEL



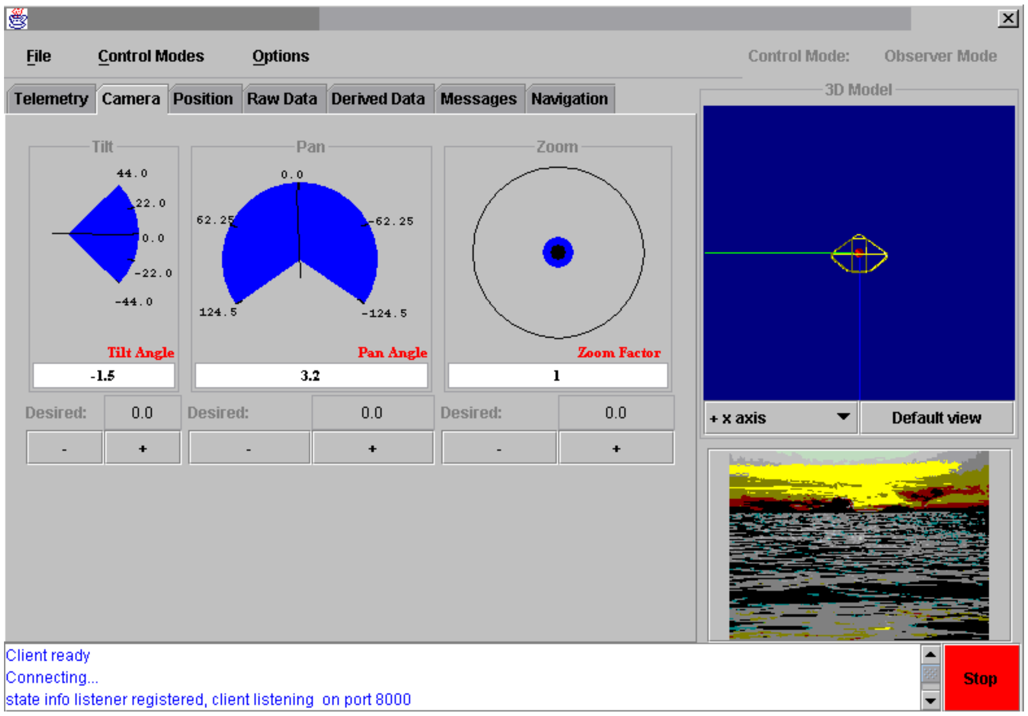
Field	Value
Velocity	3 m/s
Battery	10.2 V
Depth	-0.5 m
Angle:	2.6

Accels:	1	2
	-16.3	1.3

Motors:	1	2	3	4	5
Torque:	1.3	1.4	1.5	1.6	1.7

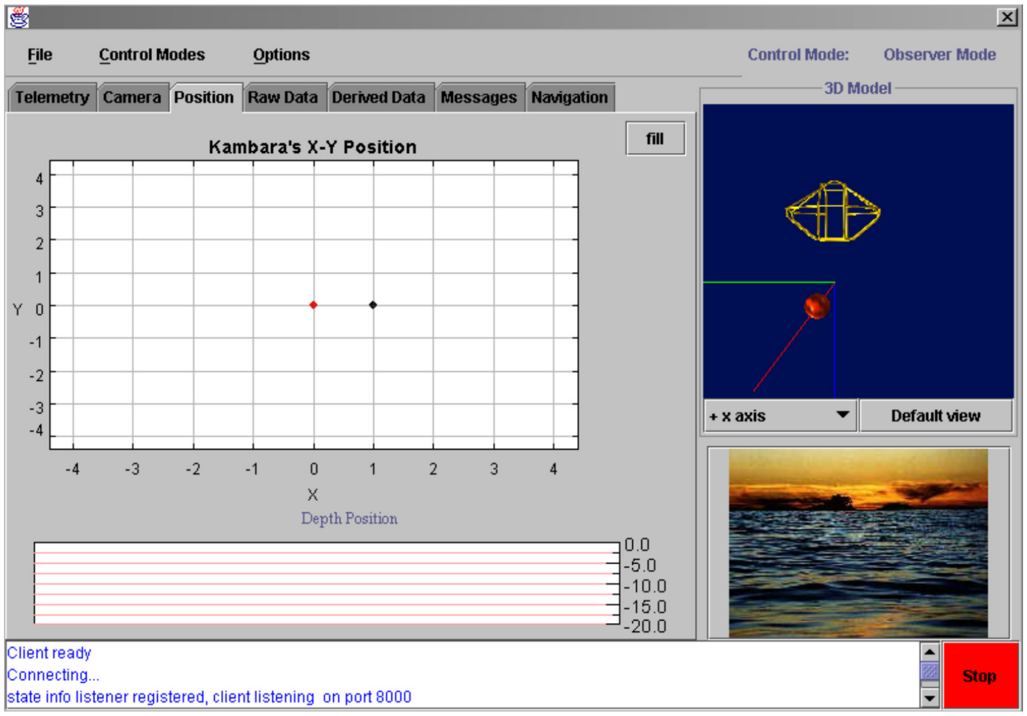
```
Client ready
Connecting...
state info listener registered, client listening  on port 8000
```

APPENDIX C2: SCREENSHOT OF CAMERA PANEL



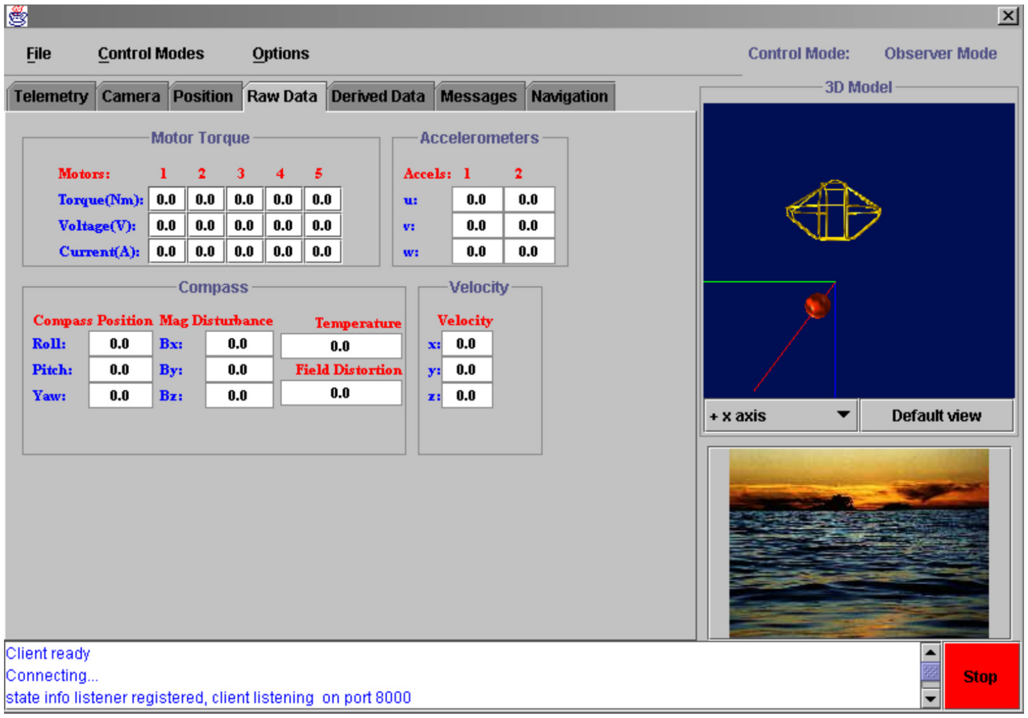
Client ready
Connecting...
state info listener registered, client listening on port 8000

APPENDIX C3: SCREENSHOT OF POSITION PANEL



```
Client ready
Connecting...
state info listener registered, client listening  on port 8000
```

APPENDIX C4: SCREENSHOT OF RAW DATA PANEL



Control Mode: Observer Mode

Motor Torque

Motors:	1	2	3	4	5
Torque(Nm):	0.0	0.0	0.0	0.0	0.0
Voltage(V):	0.0	0.0	0.0	0.0	0.0
Current(A):	0.0	0.0	0.0	0.0	0.0

Accelerometers

Accels:	1	2
u:	0.0	0.0
v:	0.0	0.0
w:	0.0	0.0

Compass

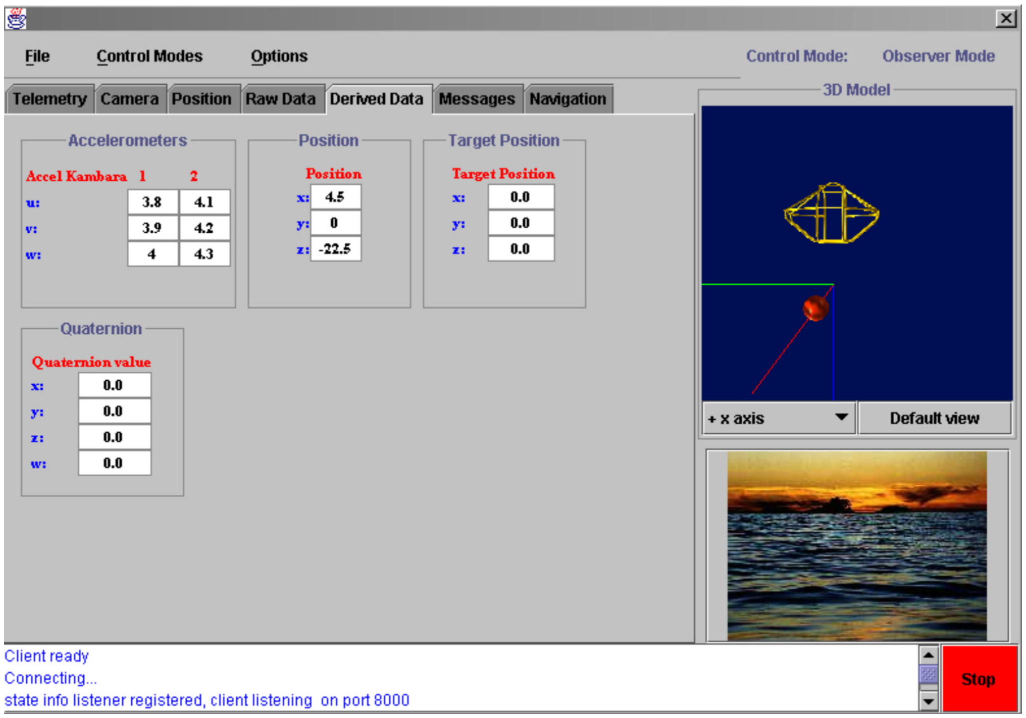
Compass Position		Mag Disturbance		Temperature
Roll:	0.0	Bx:	0.0	0.0
Pitch:	0.0	By:	0.0	Field Distortion
Yaw:	0.0	Bz:	0.0	0.0

Velocity

Velocity	
x:	0.0
y:	0.0
z:	0.0

```
Client ready
Connecting...
state info listener registered, client listening  on port 8000
```

APPENDIX C5: SCREENSHOT OF DERIVED DATA PANEL



Control Mode: Observer Mode

Accelerometers

Accel Kambara	1	2
u:	3.8	4.1
v:	3.9	4.2
w:	4	4.3

Position

Position	
x:	4.5
y:	0
z:	-22.5

Target Position

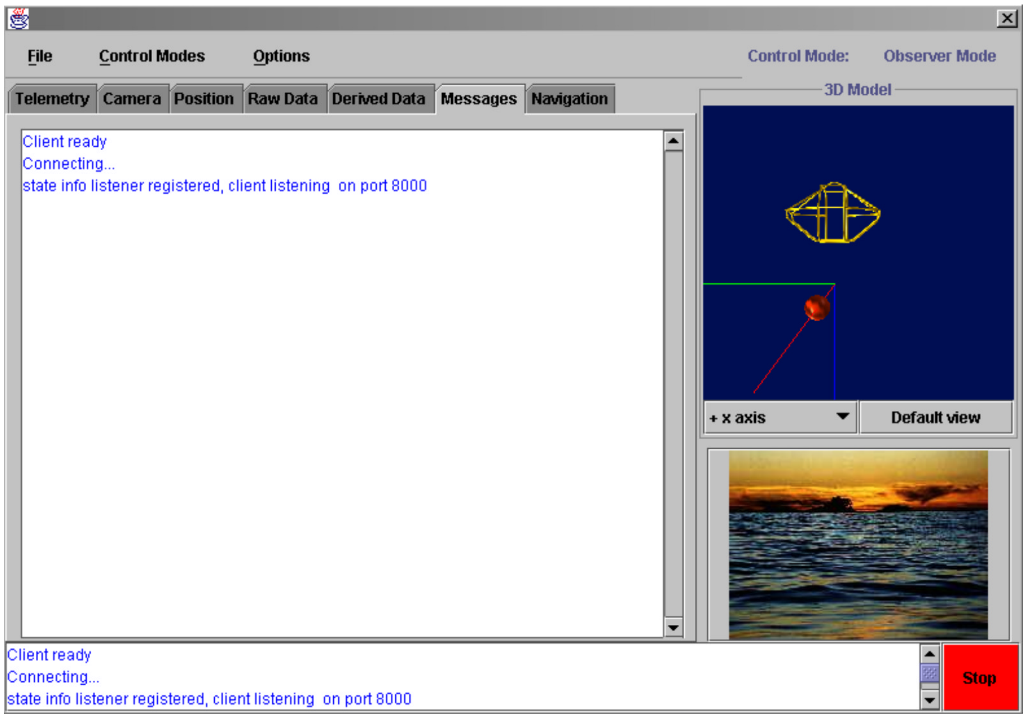
Target Position	
x:	0.0
y:	0.0
z:	0.0

Quaternion

Quaternion value	
x:	0.0
y:	0.0
z:	0.0
w:	0.0

```
Client ready
Connecting...
state info listener registered, client listening  on port 8000
```

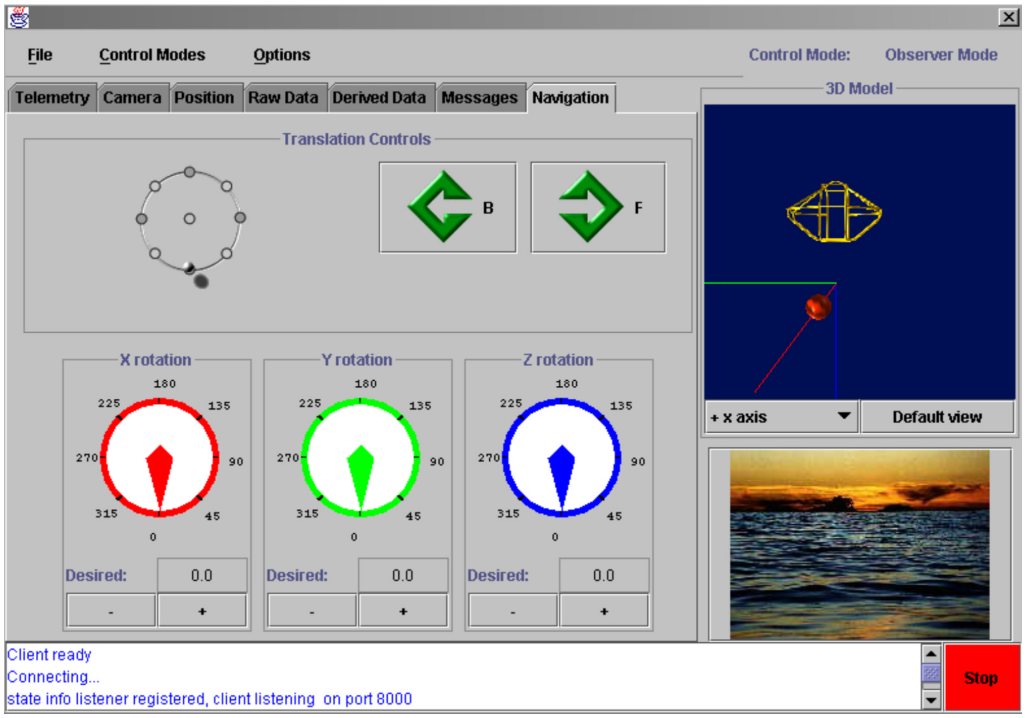
APPENDIX C6: SCREENSHOT OF MESSAGES PANEL



```
Client ready
Connecting...
state info listener registered, client listening  on port 8000
```

```
Client ready
Connecting...
state info listener registered, client listening  on port 8000
```

APPENDIX C7: SCREENSHOT OF NAVIGATION PANEL



Client ready
Connecting...
state info listener registered, client listening on port 8000